Article

Multiset-trie data structure

Mikita Akulich [1,†], Iztok Savnik [1,2,†], Matjaž Krnc [1,2,3,†], and Riste Škrekovski [1,2,3,†,*]

[1] Faculty of mathematics, natural sciences and information technologies, University of Primorska, Slovenia;
[2] Faculty of Information Studies, Novo Mesto, Slovenia
[3] Department of Mathematics, University of Ljubljana, Slovenia
* Correspondence: skrekovski@gmail.com
† These authors contributed equally to this work.

Abstract: This paper proposes a new data structure *multiset-trie* that is designed for storing and efficiently processing a set of multisets. Moreover, multiset-trie can operate on a set of sets without efficiency loss. The multiset-trie is a search tree with properties similar to those of a trie. It implements all standard search tree operations together with the multiset containment operations such as sub-multiset and super-multiset. Suppose we have a set of multisets S and a multiset X. The multiset containment operations retrieve multisets from S that are either sub-multisets or super-multisets of X. We present the mathematical analysis of a multiset-trie that gives the time complexity of the algorithms and the space complexity of the data structure. Further, the empirical analysis of the data structure is implemented in a series of experiments. The experiments illuminate the time complexity space of the multiset containment operations.

Keywords: trie data structure; multiset; containment queries

Citation: Akulich, M.; Savnik, I.; Krnc, K.; Škrekovski, R. Multiset-trie data structure. *Algorithms* **2022**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

1. Introduction

A multiset is a collection of elements where a particular element can have more than one instance. As in the case of ordinary sets, the ordering of the elements in multisets is not relevant. For example, multisets $\{1, 1, 2\}$ and $\{1, 2, 1\}$ represent the same multiset.

Multisets appear in a wide variety of domains and applications [1]. The index structures for storing sets of multisets were studied in the area of object-relational database systems to store, compress and query multiset-valued attributes efficiently [2–5]. The need to efficiently manage multisets also appears in the information retrieval [6–8] where texts are represented as multisets. In data mining, sets are often used to represent and efficiently search hypotheses in the knowledge discovery process [9,10]. In the area of expert systems, multisets are used for the representation and querying of the preconditions of rules [11]. Finally, in recent internet applications, efficient representation and search of multisets (such as user requests and object features) have become essential [12–14] for data cleaning, information integration, community mining, and entity resolution.

In this paper, we address the problems of storing, indexing, and querying the sets of multisets. In particular, we deal with the design of an index data structure that provides an efficient implementation of the multiset containment queries. Let S be an index storing a set of multisets. For a given input multiset m, a *containment query* searches for either sub-multisets or super-multisets of m in S.

Existent indexes for storing a set of multisets are rooted in the search trees [15]. The elements of a search tree can be accessed through keys. This approach is efficient for checking the membership of individual multisets m in S. However, it is not as efficient for containment queries. The search based on the containment relation requires access to the collections $C \subseteq S$ of multisets that are related to a multiset m either by a sub-multiset or a super-multisets relationship.

The existing solutions to the implementation of the containment queries use the inverted indexes [6,16–20], the signature trees [16,21–23] and, the B+ trees [20,24]. These solutions provide a key-value look-up, where a key is an element of a multiset and a value is a corresponding multiset. The containment operations require multiple key-value index accesses and also additional processing of partial results such as intersection or union of multisets.

To improve the efficiency of containment operations, we propose a data structure *multiset-trie* that is designed for storing and processing a finite bounded set of multisets. A multiset-trie generalizes the *set-trie* data structure proposed by Savnik [1,25] that was designed for storage and processing of a finite bounded set of sets. The set-trie is an extension of a trie data structure to provide, besides fast search and retrieval of sets, also the efficient implementation of the set containment operations. The set-trie is also a form of the binomial tree [15].

Multiset-trie provides a space-efficient representation of a set of multisets and efficient multiset containment operations. As in the case of set-trie, the ordering of the elements in multiset-trie is not relevant for the representation of multisets. As a consequence, the efficiency of the multiset containment operations is obtained by selecting the *specific* ordering of multiset elements. This ordering can be exploited for the efficient search in a multiset-trie.

The multiset-trie is an n-ary tree data structure. Each multiset in a multiset-trie is represented by a path from the root to a leaf node. Multiset elements are symbols from an alphabet. Each symbol from the alphabet is represented by a node at a certain level of the multiset-trie. The node stores the multiplicity of an element in a multiset.

A multiset-trie is also a kind of search tree. Similar to a trie, it uses common prefixes for a shared data representation. Unlike the compact prefix tree [26], the multiset-trie does not support path compression. However, the absence of path compression makes the multiset-trie a perfectly height-balanced tree. Moreover, when multiset-trie is full, it forms a complete n-ary tree.

Contributions. The main contributions of this paper are as follows. First, a multiset-trie is a novel data structure for storing and querying a set of multisets that provides efficient multiset containment operations.

Second, a mathematical model is developed to analyze the complexity of multiset containment operations. In particular, we estimate the size of a sub-tree traversed by a containment query and give an insight into the time complexity of containment queries.

In addition to the mathematical analysis, the size of a sub-tree visited by a multiset containment query is also the central focus of the empirical analysis. We carefully designed the experiments to unravel the main features of the time complexity space.

Finally, the mathematical, as well as empirical analyses, show an influence of ordering the elements of multisets on the efficiency of storing and processing multisets. We show that the ordering, which is based on the frequencies of the multiset elements, can speed up the multiset containment queries by orders of magnitude.

Paper organization. The paper is organized as follows. In the following Section 2, we present the multiset-trie data structure in detail. Section 3 presents the operations of the multiset-trie. These include the basic operations of search trees and multiset containment operations. The algorithms are presented in detail using pseudo code.

The description of multiset-trie operations is followed by the mathematical analysis of their complexity in Section 4. The main assumption is that multisets are constructed uniformly at random with bounded cardinality. By using probabilistic tools, we describe the time complexity of the algorithms and the space complexity of the structure.

In Section 5 we present an empirical study of the multiset-trie. Synthetic and real-world data sets are used in experiments that are designed to study the performance of multiset containment operations. The experiments highlight the methods for optimizing a multiset-trie.

The related work is reviewed in Section 6. This section describes a set-trie data structure, the inverted indexes, the signature indexes, and the multisets in relational database systems. Finally, the concluding remarks and the future work are presented in Section 7.

2. Multiset-trie data structure

Let Σ be a set of distinct symbols that define an alphabet, and let σ be the cardinality of Σ. The *multiset-trie* data structure stores multisets that are composed of symbols from the alphabet Σ. It provides the basic tree data structure operations such as insert, delete, and search, together with multiset containment and membership operations such as sub-multiset and super-multiset that will be discussed in the next section in greater detail.

Multiset ignores the ordering of its elements by definition, which allows us to define a bijective mapping $\phi : \Sigma \rightarrow I$, where I is the set of integers $\{1, 2, 3, \ldots, \sigma\}$. In this way, we obtain indexing of elements from the alphabet Σ, so we can work directly with integers rather than with specific symbols from Σ.

The multiset-trie is an n-ary tree-based data structure with the properties of the trie. A node in multiset-trie always has degree n, i.e., n children. Some of the children may be *Null* (non-existing), but the number of *Null* children can be at most $n-1$. All the children of a node, including the *Null* children, are labeled from left to right with labels c_j, where $j \in \{0, 1, \ldots, n-1\}$. Every pair of child nodes u and v that share the same parent node have different labels.

Nodes that have equal height in a multiset-trie form a level. The height of a multiset-trie is always $\sigma + 1$ if at least one multiset is in structure. The height of the root node (the first level) is defined to be 1. Levels in multiset-trie are enumerated by their height, i.e., a level L_i has height i. The connection between level height in a multiset-trie and symbols from alphabet Σ is defined as follows. A level L_i, where $i \in \{1, 2, \ldots, \sigma\}$ represents a symbol $s \in \Sigma$, such that $\phi^{-1}(i) = s$. The last level $L_{\sigma+1}$ does not represent any symbol and is named *leaf level* (*LL* for short).

Since every level, except LL represents a symbol from Σ, we can define a transition between nodes that are located at different levels in a multiset-trie. Consider two nodes u, v in a multiset-trie at levels L_i, L_{i+1} respectively, where $i \in \{1, 2, \ldots, \sigma\}$. Let a node u be a parent node of a node v and consequently a node v be a child node of a node u. Suppose that a child node v is not *Null* and has a label c_j, where $j \in \{0, 1, \ldots, n-1\}$. Then the *path* $u \rightarrow v$ represents a symbol $s \in \Sigma$ with multiplicity j, such that $\phi^{-1}(i) = s$. Such a transition $u \rightarrow v$ is called a *path of length* 1 and is allowed if and only if a node v is not *Null* and u is a parent node of a node v. If a node v has label c_0, then the path $u \rightarrow v$ represents a symbol with the multiplicity 0 respectively i.e. an empty symbol.

We define a *complete path* to be the path of length σ in a multiset-trie with the endpoints at the root node (the 1st level) and LL. Thus, a multiset m is inserted into a multiset-trie if and only if there exists a complete path in a multiset-trie that corresponds to m. Note that every complete path in a multiset-trie is unique. Therefore, the multisets that share a common prefix in a multiset-trie can have a common path of length at most $\sigma - 1$. The complete path that passes through nodes labeled by c_0 on all levels represents an empty multiset or an empty set. Thus, any multiset m that is composed of symbols from Σ with maximum multiplicity not greater than $n-1$ can be represented by a complete path in a multiset-trie.

An example of a multiset-trie data structure with $\sigma = 2$ and $\Sigma = I = \{1, 2\}$ (i.e. the mapping ϕ is an identity mapping) is shown on Figure 1. On the figure, which stores elements of $\{\emptyset, \{1, 1, 2\}, \{1, 2, 2\}, \{2\}, \{1, 2\}, \{2, 2\}\}$, the degree of a node is set to be $n = 3$, so the maximal multiplicity of an element in a multiset is $n - 1 = 2$.

Let a pair (L_i, c_j) represents a node with label c_j at a level L_i. The pair (L_1, c_j) is equivalent to $(L_1, root)$, since the first level has the root node only. According to Figure 1 we can extract inserted multisets as follows:

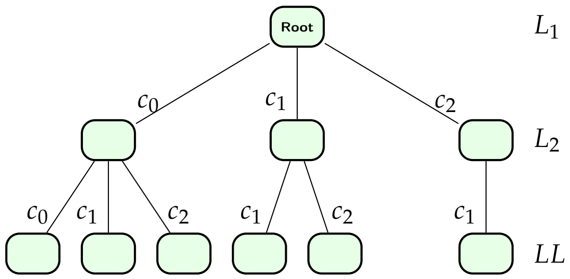


Figure 1. Example of multiset-trie structure containing multisets multisets $\emptyset$, $\{1,1,2\}$, $\{1,2,2\}$, $\{2\}$, $\{1,2\}$, $\{2,2\}$. The *Null* children are omitted.

$$
\begin{aligned}
(L_1, root) \to (L_2, c_0) \to (LL, c_0) &= \{1^0, 2^0\} = \emptyset \\
(L_1, root) \to (L_2, c_0) \to (LL, c_1) &= \{1^0, 2^1\} = \{2\} \\
(L_1, root) \to (L_2, c_0) \to (LL, c_2) &= \{1^0, 2^2\} = \{2,2\} \\
(L_1, root) \to (L_2, c_1) \to (LL, c_1) &= \{1^1, 2^1\} = \{1,2\} \\
(L_1, root) \to (L_2, c_1) \to (LL, c_2) &= \{1^1, 2^2\} = \{1,2,2\} \\
(L_1, root) \to (L_2, c_2) \to (LL, c_1) &= \{1^2, 2^1\} = \{1,1,2\}
\end{aligned}
$$

where e^k represents an element e with multiplicity k.

3. Multiset-trie operations

Let $\mathcal{M}$ be a multiset-trie and let M be a set of multisets that are inserted into the multiset-trie $\mathcal{M}$. We define a type *Multiset* in order to use it as a representation of a multiset. The type *Multiset* is an array m of constant length σ, where i-th cell represents the element $\phi^{-1}(i)$ from Σ with multiplicity $m[i]$. From now on, we agree that the first cell of an array has index 1. Let us have an example of a *Multiset* instance with $\sigma = 2$:

$$
\begin{array}{ccc}
\text{Multiset} & & \text{Instance of type Multiset} \\
\{1,1,2\} & \cong & \begin{array}{|c|c|} \hline 2 & 1 \\ \hline \end{array} \\
 & & \scriptstyle 1 \quad\; 2
\end{array}
$$

The operations supported by the multiset-trie data structure are as follows.

1. INSERT($\mathcal{M}$, m): inserts a multiset m into $\mathcal{M}$ if $m \notin M$;
2. SEARCH($\mathcal{M}$, m): returns true if a multiset $m \in M$ for a given $\mathcal{M}$, and returns false otherwise;
3. DELETE($\mathcal{M}$, m): returns true if a multiset m was successfully deleted from $\mathcal{M}$, and returns false otherwise (in case $m \notin M$);
4. SUBMSETEXISTENCE($\mathcal{M}$, m, dev): returns true if there exists a $x \in M$ for a given $\mathcal{M}$ such that $x \subseteq m$ and $|x[i] - m[i]| \leq dev$ for $1 \leq i \leq \sigma$, and returns false otherwise;
5. SUPERMSETEXISTENCE($\mathcal{M}$, m, dev): returns true if there exists a $x \in M$ for a given $\mathcal{M}$ such that $x \supseteq m$ and $|x[i] - m[i]| \leq dev$ for $1 \leq i \leq \sigma$, and returns false otherwise;
6. GETALLSUBMSETS($\mathcal{M}$, m, dev): returns the set of multisets $\{x \in M : x \subseteq m \wedge |x[i] - m[i]| \leq dev\}$ for a given $\mathcal{M}$, where $1 \leq i \leq \sigma$;
7. GETALLSUPERMSETS($\mathcal{M}$, m, dev): returns the set of multisets $\{x \in M : x \supseteq m \wedge |x[i] - m[i]| \leq dev\}$ for a given $\mathcal{M}$, where $1 \leq i \leq \sigma$.

In the following subsections, we will present each operation of the multiset-trie data structure separately.

Firstly we would like to describe some notations that will be used. The multiset-trie data structure is a recursive data structure. Hence, any subtree of a multiset-trie $\mathcal{M}$ is again a multiset-trie. This fact allows us to use the root node of a multiset-trie as its representative. Thus, the notation $\mathcal{M}$ will be used instead of $\mathcal{M}.root$ to refer to the root node of $\mathcal{M}$. Non-existing or *Null* nodes in multiset-trie will be marked as *Null* and existing nodes at the level

LL will be marked as *accepting* nodes. The array slicing operation will be used as follows. For a given array a, $a[i:]$ represents the array obtained from a by taking only the cells from index i until the last cell.

3.1. Insert

The procedure INSERT($\mathcal{M}$, m) inserts a new instance m of type Multiset into multiset-trie $\mathcal{M}$. If the complete path already exists, then the procedure leaves the structure unchanged. Otherwise, it extends partially existing or creates a new complete path. The procedure does not return any result. The pseudocode for procedure INSERT is presented in Algorithm 1.

Algorithm 1 Procedure INSERT

```
procedure INSERT(M, m)
    currentNode ← M
    for i = 1 to σ do
        if child c_{m[i]} of currentNode is Null then
            create new child c_{m[i]} of currentNode
        currentNode ← c_{m[i]}
    mark currentNode as accepting
```

3.2. Search

The function SEARCH($\mathcal{M}$, m) checks if the complete path corresponding to a given multiset m exists in the structure $\mathcal{M}$. The function returns true if the multiset m exists in $\mathcal{M}$, and returns false otherwise. The function SEARCH is presented in Algorithm 2.

Algorithm 2 Function SEARCH

```
function SEARCH(M, m)
    currentNode ← M
    for i = 1 to σ do
        if child c_{m[i]} of currentNode is Null then
            return False
        currentNode ← c_{m[i]}
    return True
```

3.3. Delete

Function DELETE($\mathcal{M}$, m) searches for the complete path that corresponds to m in order to remove it. If the path can not be found, the function immediately returns false. During the search, the function keeps track of the number of children for every node. It marks the nodes that have more than one child as *parent nodes* and remembers the label of the child, which is a potential node where the sub-tree will be cut to remove the multiset. The parent node is needed to perform a removal because the multiset-tire is an explicit data structure. When the search is completed, the function removes the sub-tree of the last found parent node and returns true. In such a way, after deletion, all the prefixes for other multisets are preserved in $\mathcal{M}$ and m is removed. The function DELETE is presented in Algorithm 3.

3.4. Sub-multiset existence

The function SUBMSETEXISTENCE($\mathcal{M}, m, dev$) checks if there exists a multiset x in $\mathcal{M}$, that satisfies the condition $x \subseteq m$ and $|x[i] - m[i]| \leq dev$, where $1 \leq i \leq \sigma$. The function starts with searching for an exact match $x = m$ in $\mathcal{M}$, since $m \subseteq m$ by definition of sub-multiset inclusion. If an exact match is not found in $\mathcal{M}$, the function uses multiset-trie to find the closest (the largest) sub-multiset of m in $\mathcal{M}$ by decreasing the multiplicity of

Algorithm 3 Function DELETE

function DELETE($\mathcal{M}$, m)
  $currentNode \leftarrow \mathcal{M}$
  $parent \leftarrow currentNode$
  $position \leftarrow 1$
  for $i = 1$ to σ **do**
    if child $c_{m[i]}$ of $currentNode$ is $Null$ **then**
      return False
    $numChildren \leftarrow 0$
    for $j = 0$ to $n - 1$ **do**
      if child c_j of $currentNode$ is not $Null$ **then**
        $numChildren \leftarrow numChildren + 1$
    if $numChildren$ is not 1 **then**
      $parent \leftarrow currentNode$
      $position \leftarrow i$
    $currentNode \leftarrow c_{m[i]}$
  child $c_{m[position]}$ of $parent \leftarrow Null$
  return True

elements in m. The parameter dev is used to limit a maximal deviation of multiplicity for a particular element in x with respect to m. At every level, the function tries to proceed with the largest possible multiplicity of an element that is provided by m. However, when the function reaches some level where it meets a *Null* node and can not go further using the path provided by m, it decreases the multiplicity of an element that corresponds to a current level with respect to the specified maximal deviation. Thus, the function can decrease the multiplicity of an element or eventually skip it in order to find the closest $x \subseteq m$. The function SUBMSETEXISTENCE is presented in Algorithm 4.

Algorithm 4 Function SUBMSETEXISTENCE

function SUBMSETEXISTENCE($\mathcal{M}, m, dev$)
  $currentNode \leftarrow \mathcal{M}$
  if $currentNode$ is $accepting$ **then**
    return True
  for $i = m[1]$ down to $\max(0, m[1] - dev)$ **do**
    if child c_i of $currentNode$ is not $Null$ **then**
      if SUBMSETEXISTENCE($c_i, m[2:], dev$) **then**
        return True
  return False

3.5. Super-multiset existence

The function SUPERMSETEXISTENCE($\mathcal{M}, m, dev$) checks if there exists super-multiset x of a given multiset m in $\mathcal{M}$, such that condition $|x[i] - m[i]| \leq dev$ is satisfied, where $1 \leq i \leq \sigma$. By analogy to the function SUBMSETEXISTENCE, the function SUPERMSETEXISTENCE starts by searching for an exact match $x = m$ in $\mathcal{M}$. If an exact match is not found in $\mathcal{M}$, the function searches for the closest (the smallest) super-multiset x of m in $\mathcal{M}$ by increasing multiplicity of elements in m. At every level, the function tries to proceed with the smallest possible multiplicity of an element that is provided by m. However, when the function reaches some level where it meets a *Null* node and can not go further using the path provided by m, it increases the multiplicity of an element that corresponds to a current level according to the deviation parameter dev. Thus, the function SUPERMSETEXISTENCE can increase multiplicity of an element up to $n - 1$, where n is the degree of a node in $\mathcal{M}$, to find the closest super-multiset $x \supseteq m$ in $\mathcal{M}$. The function SUPERMSETEXISTENCE is presented in Algorithm 5.

Algorithm 5 Function SUPERMSETEXISTENCE

function SUPERMSETEXISTENCE($\mathcal{M}, m, dev$)
  $currentNode \leftarrow \mathcal{M}$
  if $currentNode$ is $accepting$ **then**
    return True
  for $i = m[1]$ to $\min(n-1, m[1]+dev)$ **do**
    if child c_i of $currentNode$ is not $Null$ **then**
      if SUPERMSETEXISTENCE(c_i, $m[2:]$, dev) **then**
        return True
  return False

3.6. Get all sub-multisets and get all super-multisets

The algorithms for functions GETALLSUBMSETS and GETALLSUPERMSETS are based entirely on algorithms for SUBMSETEXISTENCE and SUPERMSETEXISTENCE functions that do not terminate on the first existing sub/super-multiset, but store the results and continue the procedure until all existing sub/super-multisets in $\mathcal{M}$ are found and stored. The functions GETALLSUBMSETS and GETALLSUPERMSETS are presented in Algorithm 6 and Algorithm 7 respectively.

In order to record a multiset during multiset-trie traversal, we use the variable x in the algorithms. It is an empty array of size σ where we store multiplicities of elements at each level as we traverse the tree. The variable *result* is used as a container for storing sub-multisets of m found during traversal. Both variables x and *result* are presented as global, however, they could be passed to the recursive function as parameters.

Algorithm 6 Function GETALLSUBMSETS

$result \leftarrow$ empty container
$x \leftarrow$ empty array of size σ
function GETALLSUBMSETS($\mathcal{M}, m, dev$)
  $currentNode \leftarrow \mathcal{M}$
  if $currentNode$ is $accepting$ **then**
    add copy of x to $result$
  for $i = m[1]$ down to $\max(0, m[1]-dev)$ **do**
    if child c_i of $currentNode$ is not $Null$ **then**
      $x[1] \leftarrow i$
      GETALLSUBMSETS($c_i, m[2:], dev$)

Algorithm 7 Function GETALLSUPERMSETS

$result \leftarrow$ empty container
$x \leftarrow$ empty array of size σ
function GETALLSUPERMSETS($\mathcal{M}, m, dev$)
  $currentNode \leftarrow \mathcal{M}$
  if $currentNode$ is $accepting$ **then**
    add copy of x to $result$
  for $i = m[1]$ to $\min(n-1, m[1]+dev)$ **do**
    if child c_i of $currentNode$ is not $Null$ **then**
      $x[1] \leftarrow i$ GETALLSUPERMSETS(c_i, $m[2:]$, dev)

4. Mathematical analysis of the structure

In this chapter, we present theoretical results of time and space complexity of the multiset-trie data structure. In the following Section 4.1 we discuss the running time complexity of the presented algorithms. First, in Section 4.1.1, we present the mathematical model that we use to describe the distribution of multisets in the multiset-trie and input data. Using a probabilistic approach and tools from a Galton-Watson process, we measure the expected cardinality of the multiset-trie in Theorem 2. Further, we derive the expected cardinality of the searched subtree of the multiset-trie parametrized by an input multiset in Corollary 1.

In Section 4.1.2 we discuss the running time complexity of the functions GETALLSUBMSETS and GETALLSUPERMSETS. We observe that the complexity of functions is exponential. Moreover, the worst-case running time complexity is the same for both functions, and its upper bound is the cardinality of the multiset-trie.

The remaining "existence" functions are discussed in the Section 4.1.3. We observe that out of the scope of our mathematical model, unlike in functions GETALLSUBMSETS and GETALLSUPERMSETS the mapping ϕ has an impact on performance of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE. In particular, the frequency analysis of the symbols from Σ in input data determines such a ϕ that gives a boost in performance.

We find that the performance of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE in the worst-case scenario is also exponential and does not depend on the outcome of the functions. We give a quite precise upper bound for the worst-case running time complexity, which appears to be the same for both functions. However, it must be stressed that for the positive outcome, an exponential behavior holds only on specific cases, such as the presence of the empty set in the multiset-trie.

Finalizing the mathematical analysis, we present the study of space complexity of the multiset-trie in Section 4.2. We show that the space used for the storage is asymptotically equal to the size of the input data.

4.1. Time complexity of the algorithms

The performance of the functions will be measured by the number of visited nodes in a multiset-trie during the execution of a particular query by the functions SEARCH, DELETE, SUBMSETEXISTENCE, SUPERMSETEXISTENCE, GETALLSUBMSETS, GETALLSUPERMSETS and the procedure INSERT.

By the design of the multiset-trie, it is easy to see that the functions SEARCH, DELETE and the procedure INSERT have complexity of $O(\sigma)$. Because σ is defined when the structure is initialized and does not depend on the user input afterwards, the asymptotic complexity of the functions SEARCH, DELETE and the procedure INSERT is $O(1)$. Nonetheless, in the general case, the complexity is $O(\sigma)$.

In what follows, we focus on the analysis of the more involved functions: SUBMSETEXISTENCE, SUPERMSETEXISTENCE, GETALLSUBMSETS and GETALLSUPERMSETS.

4.1.1. Mathematical model

We start with the basics of our mathematical model. Let Σ be an alphabet of cardinality σ, such that $\Sigma = \{1, 2, \ldots, \sigma\}$. Define N to be the set of all possible multisets that can be inserted in a multiset-trie. Let n be the maximal degree of a node in a multiset-trie. Then the maximal multiplicity of an element in a multiset is equal to $n-1$. Thus, the number of multisets in a complete multiset-trie is $|N| = n^\sigma$. Let M be a collection of multisets inserted into multiset-trie $\mathcal{M}$. All the multisets in M are constructed from the alphabet Σ according to the parameters σ and n. Hence, any multiset $m \in M$, has at most σ distinct elements that are members of Σ and every distinct element in m has multiplicity strictly less than n. Because a multiset does not distinguish different orderings, it is assumed, for simplicity, that all elements are ordered in ascending order. A multiset m is represented as $\{1^{k_1}, 2^{k_2}, \ldots, \sigma^{k_\sigma}\}$, where e^{k_e} represents an element $e \in \Sigma$ with multiplicity k_e.

Denote the nodes of multiset-trie on all levels but on $\sigma+1$ as *internal* and nodes on leaf level as *leaf* nodes. Observe that every internal non-root node has a degree at least 1. Indeed an insertion of a multiset requires construction of a path of length $\sigma+1$, meaning that if an internal node exists in a multiset-trie, it must have a degree at least 1. It also follows that the height of a multiset-trie is always $\sigma + 1$ as soon as at least one multiset is inserted into the data structure.

Our model assumes that all the inserted multisets are chosen with the same probability, meaning that for some $p \in (0, 1)$, the following holds:

$$P(m \in M) = p, \quad \forall\, m \in N.$$

Let $\xi_1, \xi_2, \dots, \xi_{\sigma+1}$ be random variables such that ξ_i represents the number of nodes in a multiset-trie on i-th level. For every node j on i-th level we assign a random variable ξ_{ij} to be the number of its children, such that $j \in [1, \xi_i]$. Then for every $i \in [1, \sigma]$ the following holds:

$$\xi_{i+1} = \sum_{j=1}^{\xi_i} \xi_{ij}, \tag{1}$$

where $\xi_1 = 1$. It is easy to see that the variable ξ_{i+1} can have values in the interval $[\xi_i, n^i]$ and the value of the variable ξ_{ij} is within the interval $[1, n]$. Without conditioning on the existence of any node in multiset-trie, it is easy to describe the probability of the existence of any individual node.

Lemma 1. *Any potential node on a fixed level i, where $i \in \{1, 2, \dots, \sigma + 1\}$ exists, with probability*

$$p_i = 1 - (1 - p)^{n^{\sigma+1-i}}. \tag{2}$$

Proof. Let v be an arbitrary node in a multiset-trie on an arbitrary level i. Consider the subtree with the root v and call it v-subtree. Since the height of the multiset-trie is $\sigma + 1$ we can calculate the height of the v-subtree. Taking into account that the root node has height 1, the height of the v-subtree is

$$h_v = \sigma + 1 - i.$$

A node in a multiset-trie exists if at least one node exists on the leaf level of its subtree, i.e. a node on the level $\sigma + 1$ that belongs to v-subtree. The possible number of nodes on the leaf level of v-subtree can be easily calculated knowing its height. It is equal to

$$n^{\sigma+1-i}.$$

A node at level $\sigma + 1$ exists with probability p, where $p = P(m \in M)$. Thus, the probability that there are no nodes on leaf level in v-subtree is

$$(1 - p)^{n^{\sigma+1-i}}.$$

The claim follows by taking the complement probability of the above result.
□

However, in order to determine the distribution of ξ_{ij}, one needs a lemma of a different kind.

Lemma 2. *Suppose that a node v exists at level $1 \le i \le \sigma$. Then the number of its children ξ_{iv} is modeled by a zero-truncated binomially distributed random variable on parameters n and p_{i+1}. In particular, the probability of node v having k children equals to*

$$P(\xi_{iv} = k) = \frac{\binom{n}{k}(1 - p_{i+1})^{n-k}}{1 - (1 - p_{i+1})^n} \tag{3}$$

and the corresponding probability generating function equals to

$$G_i(z) = \frac{(1 + p_{i+1}(z - 1))^n - (1 - p_{i+1})^n}{1 - (1 - p_{i+1})^n}. \tag{4}$$

Proof. In order to prove the lemma, we have to show that $\xi_{iv} \sim \mathcal{B}_0(n, p_{i+1})$. Consider an arbitrary node v on level $1 \leq i \leq \sigma$. According to the definition of the multiset-trie, a node exists at level i if and only if it has at least one child. Note that this is not true for the nodes on the leaf level $\sigma + 1$. Implies, a node on level i can have $k \in \{1, 2, \dots, n\}$ children. Let $X_0, X_1, \dots, X_{n-1}$ be random variables, they are defined as follows:

$$X_k = \begin{cases} 0 & \text{child } k \text{ of node } v \text{ does not exist} \\ 1 & \text{child } k \text{ of node } v \text{ exists} \end{cases}$$

As it was shown in previous Lemma 2, the distribution of X_k is $X_k \sim Bernoulli(p_{i+1})$. Since our model assumes that all the multisets in M are chosen uniformly at random, the variables X_k, X_l are independent for $k \neq l$. But in our case the node v can not have 0 children, so the sum $\sum_{k=1}^{n} X_k$ has a zero-truncated binomial distribution:

$$\sum_{k=1}^{n} X_k \sim \mathcal{B}_0(n, p_{i+1})$$

which completes the proof. 299

□ 300

Knowing the probability density and probability generating functions of ξ_{ij} from Lemma 2, we can now estimate the number of nodes in a randomly generated multiset-trie as follows:

$$\mathbb{E}(|\mathcal{M}|) = \mathbb{E}\left[\sum_{i=1}^{\sigma+1} \xi_i\right]. \tag{5}$$

In order to evaluate (5) we will use some of the tools from a Galton-Watson process, see Gardiner [27] for an introduction. Using the equations (1) and (4) we can derive the probability generating function for the random variable ξ_{i+1} as

$$G_{\xi_{i+1}}(z) = G_{\xi_i}(G_i(z)). \tag{6}$$

Since there is always precisely one node at the root level, we have $P(\xi_1 = 1) = 1$. Hence, the probability generating function for the random variable ξ_1 is

$$G_{\xi_1}(z) = z^1 = z \tag{7}$$

which is the initial condition for the recursive equation (6). 301

Proposition 1. *The expectation of the random variable ξ_{i+1} can be expressed as follows.*

$$\mathbb{E}(\xi_{i+1}) = \mathbb{E}(\xi_i)\mathbb{E}(\mathcal{B}_0(n, p_{i+1}))$$

for $1 \leq i \leq \sigma$. 302

Proof. Using the following property of probability generating function

$$G'_X(1^-) = \mathbb{E}(X) \tag{8}$$

the expectation for the random variable ξ_{i+1} can be derived in terms of the equation (6). 303

$$\begin{aligned} \mathbb{E}(\xi_{i+1}) &= G'_{\xi_{i+1}}(1^-) \\ &= G'_{\xi_i}(G_i(1^-))G'_i(1^-). \end{aligned} \tag{9}$$

According to (3) and (4) the value of $G_i(z)$ at 1 is 1 and the value of its derivative at 1 is $\mathbb{E}(\mathcal{B}_0(n, p_{i+1}))$. Substituting the values of $G_i(1^-)$ and $G_i'(1^-)$, and applying the property (8) we complete the proof.
□

From the Proposition 1 above and Lemma 2 we can conclude that

$$\begin{aligned} \mathbb{E}(\xi_i) &= \mathbb{E}(\xi_{i-1})\mathbb{E}(\mathcal{B}_0(n, p_i)) \\ &= \mathbb{E}(\xi_{i-1})\frac{np_i}{1-(1-p_i)^n}. \end{aligned} \tag{10}$$

Theorem 1. *Let $\mathcal{M}$ be a multiset-trie defined with parameters n, σ, and denote the number of nodes on every level i by a random variable ξ_i. Furthermore, let all multisets appear in $\mathcal{M}$ with equal probability $p \in (0,1)$. Then the expected number of nodes on every level of $\mathcal{M}$, i.e. $\mathbb{E}(\xi_i)$ is defined as*

$$\mathbb{E}(\xi_i) = n^{i-1}\frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^\sigma}}. \tag{11}$$

Proof. According to (7) the expected number of nodes on the first level is 1.
Using $\mathbb{E}(\xi_1) = 1$ and the result from Proposition 1 we get

$$\begin{aligned} \mathbb{E}(\xi_i) &= \prod_{j=2}^{i}\frac{np_j}{1-(1-p_j)^n} = \prod_{j=2}^{i} n\frac{1-(1-p)^{n^{\sigma+1-j}}}{1-(1-p)^{n^{\sigma+2-j}}} \\ &= n^{i-1}\frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^\sigma}} \end{aligned}$$

□

Having derived the expected number of nodes on every level of multiset-trie, the expected value of the total number of nodes in a multiset-trie can be calculated with respect to the parameters n, σ and p. This result is obtained in the next theorem.

Theorem 2. *The expected cardinality of a multiset-trie defined on parameters n, σ and p can be computed as*

$$\mathbb{E}(|\mathcal{M}|) = \sum_{i=1}^{\sigma+1} n^{i-1}\frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^\sigma}}, \tag{12}$$

where $r = (1-p)^n$, so $r \in (0,1)$.

Proof. Using the results obtained from Theorem 1 we compute

$$\begin{aligned} \mathbb{E}(|\mathcal{M}|) &= \mathbb{E}\left[\sum_{i=1}^{\sigma+1}\xi_i\right] \\ &= \sum_{i=1}^{\sigma+1} n^{i-1}\frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^\sigma}} \end{aligned}$$

□

With the expected number of nodes in a multiset-trie $\mathcal{M}$ obtained from Theorem 2, we can now generalize the result for a subtree in $\mathcal{M}$ parametrized by an input multiset m. The subtrees that we are interested in are the ones that contain all the sub-multisets or all the super-multisets of m. In order to calculate the expected cardinality of such subtrees, we need the following definition.

Definition 1. *Let* $m = \{1^{k_1}, 2^{k_2}, \dots, \sigma^{k_\sigma}\}$*, where* e^{k_e} *is an element* e *with multiplicity* k_e*. Let* M_1, M_2 *be the subsets of the set* M*, such that* $M_1 = \{x \in M : x \subseteq m\}$ *and* $M_2 = \{x \in M : x \supseteq m\}$*. Define* α_i *and* β_i *as follows*

$$\alpha_i = \begin{cases} 1, & i = 0 \\ \prod_{j=1}^{i}(k_j + 1), & 1 \le i \le \sigma \end{cases}$$

and

$$\beta_i = \begin{cases} 1, & i = 0 \\ \prod_{j=1}^{i}(n - k_j - 1), & 1 \le i \le \sigma \end{cases}.$$

The expected cardinality of the subtrees containing the multisets from M_1 or M_2 is defined in the following corollary.

Corollary 1. *Let* M_1, M_2, α_i *and* β_i *be defined as in previous Definition 1, then the expected cardinality of a multiset-trie subtree* $\mathcal{M}_{M_1}$ *that contains all the multisets from the set* M_1 *is equal to*

$$\mathbb{E}(|\mathcal{M}_{M_1}|) = \sum_{i=1}^{\sigma+1} \alpha_{i-1} \frac{1 - (1-p)^{\alpha_{i-1}}}{1 - (1-p)^{\alpha_\sigma}}. \tag{13}$$

The expected cardinality of a multiset-trie subtree $\mathcal{M}_{M_2}$ *that contains all the multisets from the set* M_2 *is equal to*

$$\mathbb{E}(|\mathcal{M}_{M_2}|) = \sum_{i=1}^{\sigma+1} \beta_{i-1} \frac{1 - (1-p)^{\beta_{i-1}}}{1 - (1-p)^{\beta_\sigma}}. \tag{14}$$

Proof. Using the results from Theorem 1 and Theorem 2 we derive the formulas (13) and (14) by specifying the possible number of nodes on every level in the multiset-trie according to the multiset m. Note that the formula (11) assumes that on every level but the first one, there are n possible nodes. Given sub-multiset or super-multiset query and an input multiset m the number of nodes that will be traversed on level i is defined by the number $k_{i-1} + 1$ or $n - k_{i-1} - 1$ for $i \ge 2$. On level $i = 1$ there is only one root node in any multiset-trie $\mathcal{M}$, which always exists if $M \neq \emptyset$ and is traversed for any type of query (sub-multiset and super-multiset).
□

4.1.2. GetAllSubmsets and GetAllSupermsets

In this subsection we discuss the running time complexity of the functions GETALLSUBMSETS and GETALLSUPERMSETS. It is obvious that any other algorithm for retrieving all the sub-multisets or super-multisets has worst-case running time complexity of at least $O(|M|)$. Hence, the functions GETALLSUBMSETS and GETALLSUPERMSETS have the worst-case running time complexity $O(|\mathcal{M}|)$. Indeed, the case when the algorithms retrieve all the multisets stored in a multiset-trie by traversing the whole structure can be easily constructed.

Consider the function GETALLSUBMSETS. The function takes some multiset m as an input argument. Then it returns a set of multisets $\{x \in M : x \subseteq m\}$ from the multiset-trie $\mathcal{M}$. Having a multiset m set to the largest possible multiset in N (it can also be larger)

$$m = \{1^{n-1}, 2^{n-1}, \dots, \sigma^{n-1}\}$$

the whole multiset-trie is traversed during the GETALLSUBMSETS query.

Now let us consider the function GETALLSUPERMSETS. Similarly, the function takes a multiset m as an input argument. However, in this case, it returns the set of multisets

$\{x \in M : x \supseteq m\}$ from the multiset-trie $\mathcal{M}$. In order to obtain a traversing of all the multiset-trie, one must set m to the smallest possible multiset, i.e., an empty multiset

$$m = \{\emptyset\} = \{1^0, 2^0, \ldots, \sigma^0\}.$$

Thus, we can conclude that the worst-case running time complexity of the functions GETALLSUBMSETS and GETALLSUPERMSETS is $O(\mathbb{E}(|\mathcal{M}|))$. According to the Theorem 2 the expected number of visited nodes in the worst case is

$$O(\sum_{i=1}^{\sigma+1} n^{i-1} \frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^{\sigma}}}).$$

According to the Theorem 1 the worst-case running time complexity given an input multiset m for the function GETALLSUBMSETS is

$$O(\sum_{i=1}^{\sigma+1} \alpha_{i-1} \frac{1-(1-p)^{\alpha_{i-1}}}{1-(1-p)^{\alpha_{\sigma}}})$$

and for the function GETALLSUPERMSETS is

$$O(\sum_{i=1}^{\sigma+1} \beta_{i-1} \frac{1-(1-p)^{\beta_{i-1}}}{1-(1-p)^{\beta_{\sigma}}}).$$

4.1.3. SubmsetExistence and SupermsetExistence

We start the analysis of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE with an observation. Our theoretical model assumes that all the multisets are inserted into multiset-trie at random. It was already concluded that the probability distribution function $P(m \in M)$ has an impact on the size of multiset-trie $\mathcal{M}$. Moreover, this distribution influences on the performance of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE even more.

For a real-world model, such that $P(m \in M) \neq const$ the performance of the search algorithms directly depends on the number of nodes on every level ξ_i. When the search functions check if a multiset is in multiset-trie, the complete path that corresponds to that multiset is checked. Knowing that fact, the search can be optimized during the construction of a multiset-trie.

Recall that a multiset-trie is defined on parameters n, Σ, $\sigma = |\Sigma|$ and ϕ. Let the frequency of an element e in a multiset m be the multiplicity of e in m, denoted by $mult_m(e)$. Then, the frequency of an element e can be defined as a sum $\sum_{m \in M} mult_m(e)$. According to the frequencies of elements in Σ, the performance of the multiset-trie can be optimized by the mapping $\phi : \Sigma \to I$. Indeed, the ordering of elements by their frequencies has an influence on the performance. The frequency of an element $e \in \Sigma$ affects the distribution of $\xi_{\phi(e)}$ as follows. The larger the frequency of e, the larger the number of nodes on $\phi(e)$ level. So, if the number of nodes on lower levels is greater than on higher levels, then the search functions will discard complete paths that do not satisfy the query faster. Hence, the closest match will be found faster.

Let us now switch back to our mathematical model and note that the influence of the mapping function ϕ in our model has an inessential impact on performance, because all the multisets are equally likely, and the whole domain N is used for sampling multisets.

Consider both functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE. Whenever the result is *false*, i.e. no multiset in M is a sub-multiset or super-multiset of an input multiset m, both functions in the worst case visit all the nodes in $\mathcal{M}$ but the nodes on leaf level. Of course, such a case would be very rare, assuming a random input model, but it can be constructed as follows.

Consider the function SUBMSETEXISTENCE. Then given an input multiset $m = \{1^{k_1}, 2^{k_2}, \ldots, \sigma^{k_\sigma}\}$, the collection of inserted multisets M must be equal to $M = \{x \in$

$M : k_{x,\sigma} > k_{m,\sigma}\}$. Analogically for the function SUPERMSETEXISTENCE with an input multiset $m = \{1^{k_1}, 2^{k_2}, \ldots, \sigma^{k_\sigma}\}$, the collection of inserted multisets M must be equal to $M = \{x \in M : k_{x,\sigma} < k_{m,\sigma}\}$.

Thus, the worst-case running time complexity of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE is $O(|\mathcal{M}| - |M|)$. According to Theorem 2, this value is

$$O(\sum_{i=1}^{\sigma} n^{i-1} \frac{1-(1-p)^{n^{\sigma+1-i}}}{1-(1-p)^{n^\sigma}}).$$

According to Theorem 1 the worst-case running time given an input multiset m for the function SUBMSETEXISTENCE is

$$O(\sum_{i=1}^{\sigma} \alpha_{i-1} \frac{1-(1-p)^{\alpha_{i-1}}}{1-(1-p)^{\alpha_\sigma}})$$

and for the function SUPERMSETEXISTENCE is

$$O(\sum_{i=1}^{\sigma} \beta_{i-1} \frac{1-(1-p)^{\beta_{i-1}}}{1-(1-p)^{\beta_\sigma}}).$$

Note that the summation goes only up to σ and not up to $\sigma + 1$ as in the Theorem 2 or in the Theorem 1.

As for the case when the outcome of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE is *true* one has to guarantee the termination of the algorithm at some node on the leaf level. The worst-case scenario can be constructed in the same way as for the *false* outcome but with two more multisets in M. The first multiset is the empty multiset. With the empty multiset the function SUBMSETEXISTENCE will visit the same amount of nodes as for the *false* case plus one more for the empty multiset. The second multiset is the maximal possible multiset from N. In this case the function SUPERMSETEXISTENCE will also visit the same amount of nodes as for the *false* case plus one more for the maximal multiset. Hence, the worst-case running time complexity for both outcomes (*true* and *false*) is the same.

4.2. Space complexity

As in any efficient algorithm, there is always some trade-off between space and time complexity. While offering efficient sub- and super-multiset queries, an additional space must be provided for multisets storage. Clearly, the cardinality of the set M is smaller than the size of $\mathcal{M}$, because the number of multisets in $\mathcal{M}$ is equal to the number of nodes only on the leaf level. The figure 2 demonstrates the relation between the number of multisets stored and the number of nodes needed for storage, where parameters σ and n are 26 and 10, respectively.

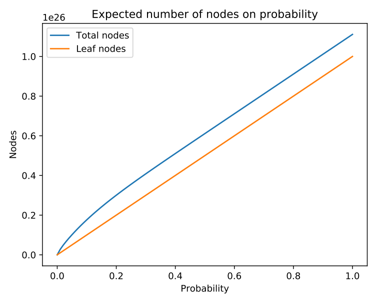


Figure 2. $\mathbb{E}(|\mathcal{M}|)$ and $\mathbb{E}(|M|)$ on probability.

As we see on the figure 2 the value of $|\mathcal{M}|$ is slightly shifted with respect to the value of $|M|$.

Now we demonstrate a more descriptive comparison between $|\mathcal{M}|$ and $|M|$. Figure 3 shows the ratio between the expected cardinality of a multiset-trie $|\mathcal{M}|$ and the actual number of multisets stored $|M|$ for parameters n and σ being 10 and 26 respectively.

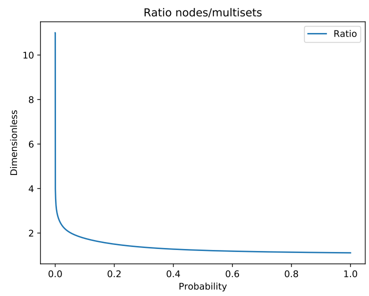


Figure 3. Ratio $\mathbb{E}(\frac{|\mathcal{M}|}{|M|})$ on p.

Note that analyzing the graph on figure 3 we can safely say that the upper bound for the ratio is $\sigma + 1$. The argument holds, because of the limit

$$\lim_{p \to 0^+} \mathbb{E}(\xi_i) = 1, \tag{15}$$

where ξ_i is the number of nodes on i-th level and $1 \leq i \leq \sigma + 1$.

However, the ratio $\sigma + 1$ can be obtained only with a very small cardinality of the set M, in particular $|M| = 1$. In order to obtain such a case the probability p must be at most $\frac{1}{n^\sigma}$.

The lower bound for the ratio is obviously at $p = 1$ and is equal to 1

$$\lim_{n,\sigma \to \infty} \frac{n^{\sigma+1} - 1}{n^\sigma (n-1)} = 1. \tag{16}$$

Since the ratio $\sigma + 1$ can be obtained for a very specific case only and with a small increase in probability, the ratio drops rapidly it can be concluded that the space complexity of the multiset-trie is $O(|M|)$.

5. Experiments

This section contains the results of experiments that were performed on the multiset-trie data structure. In particular, we will test the functions: SUBMSETEXISTENCE, SUPERMSETEXISTENCE, GETALLSUBMSETS and GETALLSUPERMSETS.

The multiset-trie is implemented in the C++ programming language. The current implementation uses only the standard library of C++14 version of the standard and has a command line interface [28]. The implementation of the program was optimized for testing, and therefore, the program operates with files to process queries. After processing all the queries, the results are stored in files for further analysis.

Before we start, we will give a few definitions of the parameters that will be varied throughout the experiments and discuss the experimental data that was used.

Let M be a set of multisets inserted to multiset-trie and let n be the maximal node degree. Let N be the power multiset of Σ, where the multiplicity of each element is bounded from above by $n - 1$. We define the *density* of a multiset-trie to be the ratio $\frac{|M|}{|N|}$, where $|\cdot|$ denotes cardinality.

The selected parameters of the data structure that will be varied in the experiments are as follows:

- σ - the cardinality of the alphabet Σ;
- n - the maximal degree of a node, which explicitly defines the maximal multiplicity of elements in a multiset;
- ϕ - mapping of letters from Σ into a set of consecutive integers;

- d - density of a multiset-trie.

The cardinality of a power multiset N is equal to n^{σ}, which means that density d of a multiset-trie depends on parameters $|M|$, σ and n. Because parameters σ and n are set when a multiset-trie is initialized, the parameter $|M|$ will be varied to change the density in experiments. As we mentioned in Section 2, the mapping ϕ determines the correspondence of letters to levels in multiset-trie, i.e., it defines the ordering of levels in multiset-trie. It is also true that ϕ defines the ordering in multisets.

In the following sections, we will present the behavior of the multiset-trie data structure in four experiments. The first three experiments use artificially generated data, and the fourth experiment uses real-world data. In the Experiment 1 a special case of the multiset-trie is considered. Only sets are allowed to be stored in the data structure, i.e., the maximally allowed multiplicity is set to 1. The performance is measured with respect to the density of the multiset-trie.

The Experiment 2 is an extension of the previous one. Here, we also measure the performance of the multiset-trie depending on its density. The difference is that the allowed multiplicity of an element is raised, i.e. the data structure is populated with multisets.

Summarizing the tests of performance depending on the density, we present the Experiment 3. It shows a nonlinearity of the performance with respect to the density of the multiset-trie.

Finally, the fourth experiment on the multiset-trie uses real-world data. In Experiment 4 the influence of the mapping ϕ is studied. The input data is obtained by mapping the real words from the English dictionary to the set of consecutive integers using the function ϕ. The experiment shows that the performance of the multiset-trie is noticeably influenced by different mappings ϕ. It also shows the usability of the multiset-trie in terms of real data demonstrating the high performance of search queries.

Data generation

We denote by *input data* the data that is used to fill the structure prior to testing and by *test data* the set of queries that are used to test the performance of the functions.

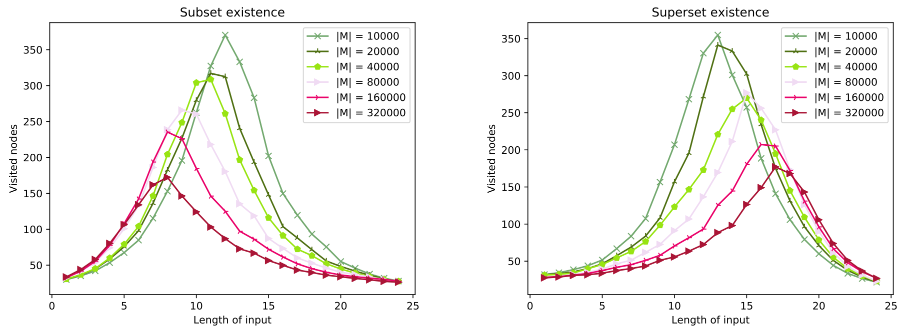
The artificially generated input data is obtained by sampling $|M|$ multisets from N. All the multisets in N are constructed according to parameters σ and n and represent the power multiset of the alphabet Σ. Every multiset in M is chosen from N with equal probability p. Thus, the probability p gives a collection M of multisets that are sampled from N with uniform distribution. Uniform distribution is chosen in order to simulate random user input.

The test data is generated artificially and constructed as follows. Given the parameters σ and n, the possible size of a multiset varies from 1 to σn. The number of randomly generated test multisets for every value of multiset size is 1500. In other words, we perform 1500 experiments in order to measure the number of visited nodes for the queries with a test multiset of distinct sizes. The final value of visited nodes is calculated by taking an arithmetic mean among all 1500 measurements.

5.1. Experiment 1

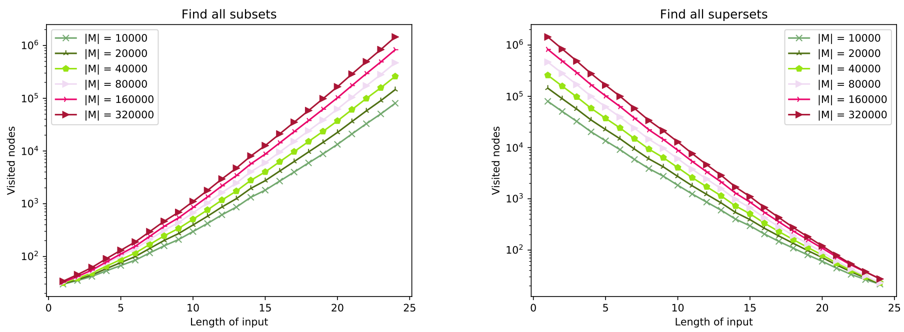
This experiment shows the performance of multiset-trie being used for storing and retrieving *sets* instead of *multisets*. We restrict multiset-trie in order to make a closer comparison with the *set-trie* data structure [25]. In this case, we set the maximal node degree n to be 2 and σ to be 25. The mapping ϕ does not have an influence in this particular experiment because the input data is generated artificially with uniform distribution. On average, the results will be the same for any ϕ, since all the multisets are equally likely to appear in M. The parameter $|M|$ varies from 10000 sets up to 320000 sets. According to the parameters n and σ, the cardinality of N is 33554432 $\approx 3.36 \times 10^{7}$. Thus, the calculated density of the multiset-trie with respect to $|M|$ varies from 0.3×10^{-3} to 9.5×10^{-3}.

The performance of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE increases as the density increases (see figures 4a and 4b). The results are as expected



(a) submsetExistence **(b)** supermsetExistence

Figure 4. Existence functions of Experiment 1.



(a) getAllSubmsets **(b)** getAllSupermsets

Figure 5. Exhaustive functions of Experiment 1.

because the increase of the density increases the probability of finding sub-multiset or super-multiset in multiset-trie, which leads to a lower number of visited nodes.

The maxima are located between 175 and 375 for SUBMSETEXISTENCE and between 175 and 350 for SUPERMSETEXISTENCE. According to those maxima we can deduce that at least 7-15 multisets were checked in order to find sub-multiset or super-multiset, which is from 0.02×10^{-3} to 1.5×10^{-3} of the multiset-trie and from 1.9×10^{-7} to 4.5×10^{-7} of the complete multiset-trie.

As the density increases, the peaks shift from the center to the left or to the right, for SUBMSETEXISTENCE and SUPERMSETEXISTENCE respectively. The shifts are the consequence of the uniform distribution of sets in M. Since every set has the same probability of appearing in M, the distribution of set sizes in M is normal. Consequently, with the increase in the density of the multiset-trie the number of sets in M with cardinality $\frac{1}{2}\sigma$ will be larger than the number of sets with cardinality $\frac{1}{2}\sigma \pm \epsilon$, for $\frac{1}{2}\sigma > \epsilon > 0$. So the function SUBMSETEXISTENCE needs to visit less nodes for test sets of size $\frac{1}{2}\sigma$ than for test sets of size $\frac{1}{2}\sigma \pm \epsilon$. The function decreases the multiplicity of some elements (in some cases skips them) in order to find the closest subset. Hence, the peak shifts to the left. Oppositely the function SUPERMSETEXISTENCE increases the multiplicity of some elements (in this case, adding new elements) in order to find the closest superset. Thus, the peak shifts to the right.

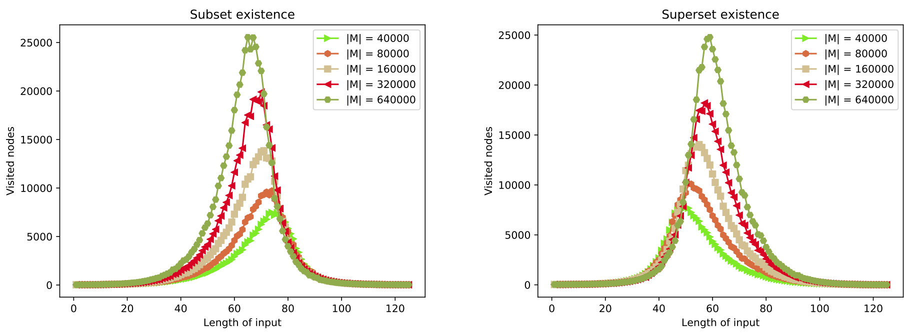
Note that despite the peak shifts both functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE have approximately the same worst-case performance.

The performance of the functions GETALLSUBMSETS and GETALLSUPERMSETS decreases as the density increases (see figures 5a and 5b). This happens because the number of multisets in multiset-trie increases, which means that any multiset in the data structure will have more sub- and super-multisets. The maxima for both functions varies from 8.0×10^4 to 1.5×10^6 visited nodes. We can notice that local maxima for the functions GETALLSUBMSETS and GETALLSUPERMSETS differs with respect to the length of input. The explanation is very simple. In order to find all submultisets of a small set the function has to traverse a small part of the multiset-trie. As the size of a set increases, the part of a multiset-trie where all the submultisets of a given set are stored also increases. The opposite holds for the function GETALLSUPERMSETS.

Despite the fact that for a lookup of any set/multiset σ nodes must be visited in multiset-trie on average case, the data structure has a very similar performance results in comparison to the *set-trie* data structure.

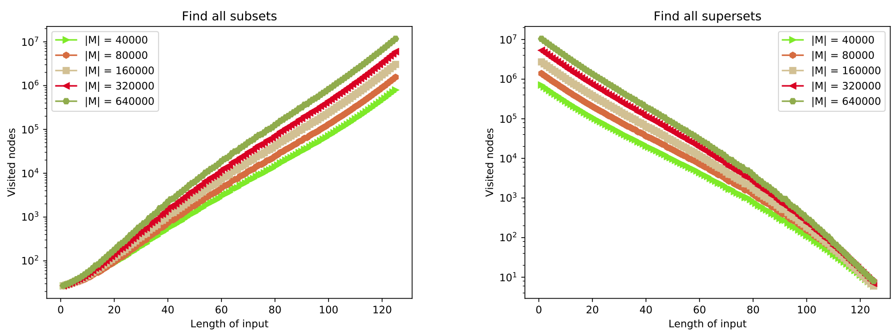
5.2. Experiment 2

In the Experiment 2 we demonstrate the performance of the unrestricted multiset-trie allowing *multisets* to be inserted into the data structure. We set n to be 6 and retain $\sigma = 25$ as it was in Experiment 1. The mapping ϕ does not have an influence on the results, since the input data is generated artificially with uniform distribution. The cardinality of M varies from 40000 to 640000 multisets. Thus, the calculated density d varies from 1.4×10^{-15} to 2.25×10^{-14}. The density is much smaller than in Experiment 1, because now we allow multisets to be stored in the data structure and according to the parameters n and σ the cardinality of N is $6^{25} = 2.84 \times 10^{19}$.



(a) submsetExistence **(b)** supermsetExistence

Figure 6. Existence functions in Experiment 2.



(a) Experiment 2, getAllSubmsets function. **(b)** Experiment 2, getAllSupermsets function.

Figure 7. Exhaustive functions in Experiment 2.

As we can see from the graphs on figures 6a and 6b, the performance of the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE becomes worse as the density increases. In this case, the number $|M|$ is slightly larger than in the Experiment 1, but the density is very small. Consequently, multiset-trie becomes more sparse. Multisets in a sparse multiset-trie differ more, which leads to a larger number of visited nodes.

The maxima for both functions vary from 7500 to 25000 visited nodes. According to those maxima, at least 300-1000 multisets were checked in order to find sub-multiset or super-multiset, which is from 1.5×10^{-3} to 7.5×10^{-3} of the entire multiset-trie and from

1.1×10^{-17} to 3.4×10^{-17} of the complete multiset-trie. The percentage of visited multisets with respect to $|M|$ is larger than in the Experiment 1. However, if one would compare the percentage of visited multiset with respect to complete multiset-trie, then in the case of Experiment 2 it is less by 10 orders than in the Experiment 1.

The peaks are shifted from the center to the left and right for SUBMSETEXISTENCE and SUPERMSETEXISTENCE respectively. Such behavior was previously observed in the Experiment 1. The explanation is the same: the input data has a uniform distribution, implying that the size of multisets in M is normally distributed. Because of the normal distribution of the size of multisets, the shift of the peak occurs as the density increases.

It can also be observed that, as in previous Experiment 1, both functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE have similar worst-case performance.

The functions GETALLSUBMSETS and GETALLSUPERMSETS decrease their performance as the density increases (see figures 7a and 7b). This happens because the number of multisets increases as the density increases. So there are more nodes that have to be visited in order to retrieve all sub- or super-multisets of some multiset. The maximum for both functions varies from 0.9×10^5 to 1.5×10^7 visited nodes. As it was observed in Experiment 1, the maxima occur at the opposite points. For the function GETALLSUBMSETS it will always be at the largest size of the multiset, which is 125 in our case. Conversely the maximum for the GETALLSUPERMSETS is at the smallest size of multiset, which is 0 (an empty set).

The results of the Experiment 1 show that the performance of functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE increases as the density increases. However, we observe the opposite behavior in the Experiment 2. We explain the reason of such a contradiction in the next Experiment 3

5.3. Experiment 3

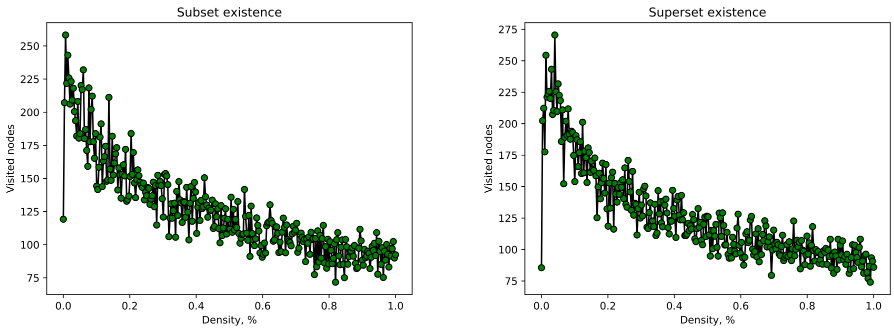
The results of the Experiment 1 and Experiment 2 have shown that as the density of a multiset-trie increases the performance of functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE can both get better and worse. The reason for such a behavior is that the dependence of the number of visited nodes on density is not a linear function. The performance of the abovementioned functions is maximal when multiset-trie is complete. As multiset-trie becomes more sparse (the density is small), multisets differ more, and the number of visited nodes increases. However, multisets differ less when the density is high, so the number of visited nodes decreases. Since the dependence of the number of visited nodes on the density of multiset-trie is a continuous function on the interval $[0, 1]$, there exists a global maximum. In other words, there exists such a value of density where the number of visited nodes is maximal.

In this experiment, we empirically find the extremum of density for functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE. The parameters σ and n are set to 12 and 5, respectively. The density varies from 1.0×10^{-6} to 1.0×10^{-2}. The number of visited nodes was chosen to be maximal for each value of a particular density.

As we see on figures 8a and 8b both functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE have the maximum around $d \approx 7.0 \times 10^{-5}$. The maximum is less than 0.3×10^{-3} and greater than 1.4×10^{-15}, which explains the behavior of multiset-trie in Experiment 1 and Experiment 2. It is safe to say that the maximum may vary depending on parameters n and σ, but such a maximum always exists. Therefore, we omit the experiments with different parameters n and σ.

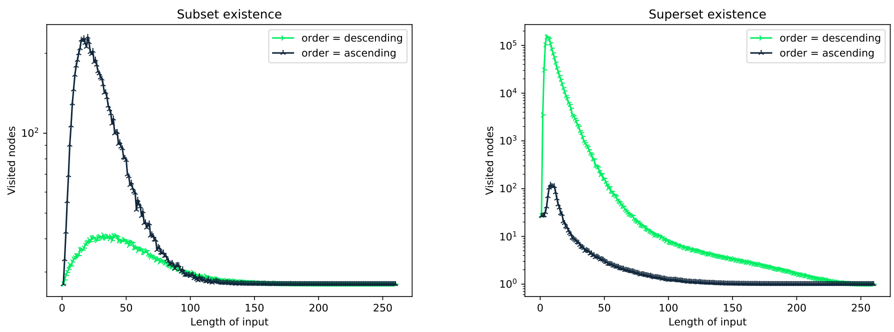
5.4. Experiment 4

In previous experiments, the input was generated artificially with uniform distribution, so there was no influence of the mapping function ϕ on the performance of tested functions. This experiment shows the influence of the mapping ϕ from alphabet Σ to a set of consecutive integers. We obtain the influence by taking the real-world data as input data.



(a) submsetExistence **(b)** supermsetExistence

Figure 8. Exsitence functions in Experiment 3.



(a) submsetExistence **(b)** supermsetExistence

Figure 9. Existence functions in Experiment 4.

The data is taken from the English dictionary, which contains 235883 different words. Those words are mapped to multisets of integers according to the ϕ. In particular, we are interested in cases when $\phi(\Sigma)$ enumerates letters by their relative frequency in the English language. We say that $\phi(\Sigma)$ maps letters in *ascending order* if the most frequent letter is mapped to number σ. Conversely, in *descending order* this letter is mapped to the number 1. The size of the alphabet σ is set to the size of the English alphabet 26. The degree of a node n is set to 10. On average, the multiplicity of letters is, of course, less than 10. We choose such a large node degree allowing the multiplicity to be up to 10 because the dictionary contains such words.

The results on figures 9a and 9b are more balanced when letters are ordered by frequency in ascending order. The maxima for the functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE are at 250 visited nodes.

According to the design of the data structure multiset-trie, we can say something about multiset only if we try to reach it, i.e., to find the complete path that corresponds to a particular multiset. This means that in order to give an answer about the existance of some multiset, one has to check the leaf level in multiset-trie.

Letters that have the least frequencies are now located at the top of multiset-trie according to ascending order of letters by frequency. This means that the search becomes narrower because a lot of invalid paths will be discarded on top most levels. Thus, multiset-trie can be traversed faster.

As you may have noticed the functions GETALLSUBMSETS and GETALLSUPERMSETS were not tested in this experiment. Those functions are not affected by variations of the mapping ϕ, because for any multiset, they retrieve all sub/supermultisets. This means that the number of visited nodes will not be changed as ϕ varies.

6. Related work

The data structure multiset-trie is related to the data structures and indexes designed to store and manage sets and multisets. We mainly focus on the related data structures and indexes that efficiently support the set and multiset containment queries. Firstly, we summarize our previous work on the data structure for managing sets in Section 6.1. Next, we present in Section 6.2 the related work on the inverted files, i.e., the index structure that serves as a central data structure in the area of Information Retrieval (abbr. IR) but also for storing sets and multisets in database management systems. The alternative to the inverted file is the signature tree that is presented in Section 6.3. Finally, we describe the related work in the area of database management systems in Section 6.4. We review the novel index structures used for the containment queries and the proposed containment join algorithms.

6.1. Set-trie

The multiset-trie is closely related to the set-trie data structure introduced by Savnik in [1,25]. A set-trie is a trie data structure that is adapted for the efficient storage and retrieval of sets instead of the sequences of symbols. The set-trie provides the set containment operations such as retrieval of the *nearest* subset or supersets as well as retrieval of *all* subsets and supersets from the sets of sets.

Since we are storing sets where each element of the set can appear only once, and the ordering of elements is not important, the ordering of the elements from the alphabet can be used for guiding the search in set containment operations. Each set is represented in a set-trie by a path including the increasing elements of a set represented by set-trie nodes. Since all sets from a set-trie are ordered by the increasing value of the set elements, the children of each set-trie node n can only be the elements larger than the element n. For a given set s and a set-trie S, the set containment operations search solely the sub-tree of S that includes all the sets (paths from a root to a set-trie node) that are the possible subsets or supersets of s.

The data structure multiset-trie generalizes the set-trie by providing storage for the set of multisets. When the multiset-trie is restricted to store a set of sets, the underlying data structure becomes a simple binary tree. Moreover, all the operations of the set-trie are also supported by the multiset-trie. The generalization comes with a small penalty in performance if we compare the multiset-trie with the set-trie in the performance of the set containment operations. The downside of such a generalization is that multiset-trie no longer supports path compression that was obtained in set-trie. However, the design of multiset-trie provides storage of multisets with constant worst-case time complexity of the set containment operations.

6.2. Inverted file

The inverted file [6,7,16] is the most common data structure used to represent a collection of (multi)sets. In the area of IR [8] the inverted files are used for searching documents that contain a given set of words. It is composed of two parts: a dictionary and the postings. The dictionary maps each word to a list of document identifiers together with the locations of words in documents. The dictionary is most often implemented by a variant of a search tree, such as a B+ tree. The postings are implemented as a list of positions that are stored on the disk because of the huge amount of documents usually indexed by the inverted file. Since we can have a large number of postings for one word, the postings are compressed. Furthermore, several possible optimizations exist in the representation and implementation of postings [7], such as sorting of postings, a technique called skipping, and others.

The empirical analyses [16,24] show that the inverted file is the most efficient data structure for containment queries among the data structures: the sequential signature file, the signature tree, the extendible signature hashing, and the inverted file.

6.3. Signature trees

A dynamically balanced signature tree [29,30], or S-tree, is an alternative data structure for the representation of multisets. An S-tree stores objects on the basis of their attributes represented in the form of signatures. A signature of an object is formed by the discretization of object attributes. Each attribute is discretized by mapping the attribute values to a sequence of bits. The bit sequences are the abstractions of the values of object attributes. They are glued together to form a signature of an object. The mappings from attribute values to sequences of bits are defined in such a way that allows superimposing a set of signatures by a single signature. Such a signature is often formed by using the operation OR. This property of signatures provides the means for the construction of the hierarchy of signatures that is utilized for efficient search. The multisets can be effectively represented using signatures, and the superposition operation can be implemented by the operation OR. The use of the signature tree for the containment operations was studied by Tousidou et al. [22]. They show that S-tree that uses linear hash partitioning can be used to implement the containment operations efficiently.

6.4. Multisets in relational databases

The index structures for the efficient implementation of the (multi)set containment queries were studied in the frame of the relational DBMS as well as the object-relational DBMS, where we can use multivalued attributes, including sets, multisets (bags), and lists. Zhang et al. [31] compared the performance of the containment queries implemented in a standard relational DBMS (abbr. RDBMS) to an Information retrieval engine. The results show that, in general, the IR engine performs better than an RDBMS on containment queries. They have identified the problems that reason for the poor performance of the containment operations in an RDBMS and showed that with some modifications, an RDBMS could perform this class of queries more efficiently.

The joins in an object-relational DBMS can be defined by means of the containment operations. A number of *containment join* methods have been proposed [32–35]. Ramasamy et al. propose the use of the partitioning set join that relies on the representation of sets by using signatures [32]. The signature-based representation allows efficient implementation of the set comparison operations. The partitioning set join was further improved by Melnik at al. [33] to handle large sets and to speed up the partitioning phase of the algorithm. Further, Jampani et al. introduce the PRETTI join algorithm that combines an inverted file with a prefix tree for the efficient implementation of the containment joins. The algorithm for $R \bowtie T$ recursively computes record identifiers from T while traversing a prefix tree storing the sets from R. The algorithm uses a single intersection of two lists to enumerate the matching pairs of rid-s. The PRETTI algorithm was improved by Luo at al. [35] by replacing the prefix tree with the Patricia tree.

7. Conclusions and future work

One of the conclusions of studying the multiset-trie both theoretically and empirically is that our data structure is input sensitive. Input sensitivity implies a non-consistent performance on different input data. However, our argument that the performance can be optimized by pre-processing the input data is confirmed in Experiment 4. Pre-processing determines the optimal encoding for input data and ensures the best performance of the multiset-trie on particular input data. For example, in the case of storing words in the multiset-trie, the search queries can always be optimized based on the frequencies of letters in a specific language. We also see from Experiments 1 and 2 that the dependence of the multiset-trie performance on the density is not a linear function. Yet the function is continuous, and the point of inflection is unique on the whole domain, as shown in Experiment 3. This allows us to predict whether multiset-trie can be used for some particular application, serving a high performance.

The mathematical analysis section provides a non-trivial insight regarding the behavior of multiset-trie datastructure when used in randomized data. It is estimated that the space complexity of multiset-trie is of order $O(|M|)$, which is the minimal possible space required by any data structure for storage of $|M|$ objects. As for the running time complexity of algorithms, the basic tree functions such as INSERT, SEARCH, and DELETE all have a constant complexity once the multiset-trie is defined. The "getAll" multiset containment functions have worst-case running time complexity of $O(|\mathcal{M}|)$, where $|\mathcal{M}|$ is the cardinality of the multiset-trie data structure. The "existence" multiset containment functions have the worst-case running time complexity of $O(|\mathcal{M}| - |M|)$, where $|\mathcal{M}|$ is the cardinality of the multiset-trie and $|M|$ is the number of inserted multisets (nodes on leaf level).

The implementation of the multiset-trie is not optimized. For example, the main reason for space-inefficiency is in implementing the links from a node to its children. Our implementation uses an array data structure to link a node to its children where each element of an array, indexed by the multiplicity of the element of the next level, includes a link to a sub-tree. A custom-implemented small and extendable hash table would significantly decrease the amount the space needed to represent a multiset-trie.

Further steps in our research will be to extend the functionality of the multiset-trie. We are interested in more flexible multiset containment queries where additional conditions constrain the sub and super-multisets. For example, the multiplicity of an element in a multiset can be bounded in operations getAllSubmsets and getAllSupermsets. Furthermore, the similarity search on multisets can be implemented by modifying the algorithms for searching the sub and super-multisets. The second line of research is to investigate the multiset-trie as a database index data structure. A disk-based index data structure allows storing and managing a huge amount of multisets. The mapping from a multiset-trie, i.e., a n-ary search tree, to a block-based index can be easily defined because of the regularity of multiset-trie. It will be interesting to compare the multiset-trie with other existing disk-based index data structures.

Funding information.

Authors acknowledge the support in part by the Slovenian Research Agency, research program P1-0383.

References

1. Savnik, I.; Akulich, M.; Krnc, M.; Škrekovski, R. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. *PLoS ONE* **2021**, *2*.
2. Bouros, P.; Mamoulis, N.; Ge, S.; Terrovitis, M. Set containment join revisited. *Knowledge and Information Systems* **2016**, *49*, 375–402.
3. Gripon, V.; Rabbat, M.; Skachek, V.; Gross, W.J. Compressing multisets using tries. In Proceedings of the Information Theory Workshop (ITW), 2012 IEEE. IEEE, 2012, pp. 642–646.
4. Ross, K.A.; Stoyanovich, J. Symmetric relations and cardinality-bounded multisets in database systems. In Proceedings of the Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 2004, pp. 912–923.
5. Steinruecken, C. Compressing sets and multisets of sequences. *IEEE Transactions on Information Theory* **2015**, *61*, 1485–1490.
6. Zobel, J.; Moffat, A.; Sacks-Davis, R. An efficient indexing technique for full-text database systems. In Proceedings of the Proceedings of VLDB. IEEE, 1992, pp. 352–352.
7. Zobel, J.; Moffat, A. Inverted files for text search engines. *ACM computing surveys (CSUR)* **2006**, *38*, 6.
8. Manning, C.D.; Raghavan, P.; Schütze, H.; et al. *Introduction to information retrieval*; Vol. 1, Cambridge university press, Cambridge, 2008.
9. Mannila, H.; Toivonen, H. Levelwise Search and Borders of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery* **1997**, *1*, 241–258.
10. Flach, P.A.; Savnik, I. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.* **1999**, *12*, 139–160.
11. Forgy, C. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligences* **1982**, *19*, 17–37.
12. Bayardo, R.J.; Ma, Y.; Srikant, R. Scaling up All Pairs Similarity Search. In Proceedings of the Proceedings of the 16th International Conference on World Wide Web; Association for Computing Machinery: New York, NY, USA, 2007; WWW '07, p. 131–140.
13. Xiao, C.; Wang, W.; Lin, X.; Yu, J.X.; Wang, G. Efficient Similarity Joins for Near-Duplicate Detection. *ACM Trans. Database Syst.* **2011**, *36*.
14. Wang, X.; Qin, L.; Lin, X.; Zhang, Y.; Chang, L. Leveraging set relations in exact set similarity join. *Proceedings of the VLDB Endowment* **2017**, *10*, 925–936.
15. H. Cormen, T.; Leiserson, C.; L. Rivest, R.; Stein, C. *Introduction to Algorithms, Second Edition*; 2001.
16. Zobel, J.; Moffat, A.; Ramamohanarao, K. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)* **1998**, *23*, 453–490.
17. Zobel, J.; Moffat, A. Inverted files for text search engines. *ACM computing surveys (CSUR)* **2006**, *38*, 6.
18. Broder, A.Z.; Eiron, N.; Fontoura, M.; Herscovici, M.; Lempel, R.; McPherson, J.; Qi, R.; Shekita, E. Indexing shared content in information retrieval systems. In Proceedings of the International Conference on Extending Database Technology. Springer, 2006, pp. 313–330.
19. Terrovitis, M.; Passas, S.; Vassiliadis, P.; Sellis, T. A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes. In Proceedings of the Proceedings of the 15th ACM International Conference on Information and Knowledge Management; ACM: New York, NY, USA, 2006; CIKM '06, pp. 728–737.
20. Terrovitis, M.; Bouros, P.; Vassiliadis, P.; Sellis, T.; Mamoulis, N. Efficient Answering of Set Containment Queries for Skewed Item Distributions. In Proceedings of the Proceedings of the 14th International Conference on Extending Database Technology; ACM: New York, NY, USA, 2011; EDBT/ICDT '11, pp. 225–236.
21. Deppisch, U. S-tree: A Dynamic Balanced Signature Index for Office Retrieval. In Proceedings of the Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval; ACM: New York, NY, USA, 1986; SIGIR '86, pp. 77–87.
22. Tousidou, E.; Bozanis, P.; Manolopoulos, Y. Signature-based Structures for Objects with Set-valued Attributes. *Inf. Syst.* **2002**, *27*, 93–121. https://doi.org/10.1016/S0306-4379(01)00047-3.
23. Chen, Y.; Shi, Y. Signature Files and Signature File Construction. *Encyclopedia of Database Technologies and Applications* **2005**. https://doi.org/10.4018/978-1-59140-560-3.ch105.
24. Helmer, S.; Moerkotte, G. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal* **2003**, *12*, 244–261. https://doi.org/10.1007/s00778-003-0106-0.
25. Savnik, I. Index data structure for fast subset and superset queries. In Proceedings of the International Conference on Availability, Reliability, and Security. Springer, 2013, pp. 134–148.
26. Sedgewick, R.; Wayne, K. *Algorithms*, 4th ed.; Addison-Wesley Professional, 2011.
27. Gardiner, C.W. *Stochastic methods*; Springer-Verlag, Berlin–Heidelberg–New York–Tokyo, 1985.
28. Akulich, M. Mstrie repository. https://github.com/nick-ak96/mstrie, 2019.
29. Deppisch, U. S-tree: A Dynamic Balanced Signature Index for Office Retrieval. In Proceedings of the Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval; ACM: New York, NY, USA, 1986; SIGIR '86, pp. 77–87. https://doi.org/10.1145/253168.253189.
30. Pfaltz, J.L.; Berman, W.J.; Cagley, E.M. Partial-match Retrieval Using Indexed Descriptor Files. *Commun. ACM* **1980**, *23*, 522–528. https://doi.org/10.1145/359007.359013.

31. Zhang, C.; Naughton, J.; DeWitt, D.; Luo, Q.; Luo, Q.; Lohman, G. On Supporting Containment Queries in Relational Database Management Systems. In Proceedings of the Proceedings of SIGMOD; ACM: New York, NY, USA, 2001; pp. 425–436. https://doi.org/10.1145/375663.375722.
32. Ramasamy, K.; Patel, J.M.; Naughton, J.F.; Kaushik, R. Set containment joins: The good, the bad and the ugly. In Proceedings of the Proc. of VLDB, 2000.
33. Melnik, S.; Garcia-Molina, H. Adaptive Algorithms for Set Containment Joins. *ACM Trans. Database Syst.* **2003**, *28*, 56–99. https://doi.org/10.1145/762471.762474.
34. Jampani, R.; Pudi, V. Using Prefix-Trees for Efficiently Computing Set Joins. In Proceedings of the Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings, 2005, pp. 761–772. https://doi.org/10.1007/11408079_69.
35. Luo, Y.; Fletcher, G.H.L.; Hidders, J.; Bra, P.D. Efficient and scalable trie-based algorithms for computing set containment relations. In Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, 2015, pp. 303–314. https://doi.org/10.1109/ICDE.2015.7113293.