Noname manuscript No.
(will be inserted by the editor)

Multiset-trie data structure

Mikita Akulich · Iztok Savnik · Matjaž Krnc · Riste Škrekovski

the date of receipt and acceptance should be inserted later

Abstract This work proposes a new data structure *multiset-trie* that is designed for storage and efficient processing of a set of multisets. Moreover, multiset-trie can operate on a set of sets without efficiency loss. The multiset-trie is a search-tree-based data structure with the properties similar to those of a trie. It implements all standard search tree operations together with the multiset containment operations such as submultiset and supermultiset. Containment operations allow efficient search and retrieval of the individual multisets and also collections of multsets. In particular, it can find and retrieve closest submultisets or supermultisets as well as all submultisets or supermultisets that are parameterized by user input. Multiset-trie is compared to modern solutions that provide containment operations on multisets. The study shows how efficient multiset-trie is by outperforming inverted file in both equality and containment queries by up to 4 orders of magnitude in query execution time.

Keywords multiset-trie, multiset, bag-of-words, trie, data structure, containment operations, inverted file

Mikita Akulich
University of Primorska, FAMNIT, Koper, Slovenia
E-mail: mikita.akulich@gmail.com

Iztok Savnik
University of Primorska, FAMNIT, Koper, Slovenia
E-mail: iztok.savnik@famnit.upr.si

Matjaž Krnc
University of Primorska, FAMNIT, Koper, Slovenia
E-mail: matjaz.krnc@famnit.upr.si

Riste Škrekovski
University of Ljubljana, FMF, Ljubljana, Slovenia
E-mail: riste.skrekovski@famnit.upr.si

1 Introduction

A multiset is a data structure that represents a collection of elements. It generalizes a set data structure by allowing duplicate elements in a collection. Multisets appear in a wide variety of domains and applications. The index structures for storing sets of multisets were studied in the area of object-relational database systems to efficiently store, compress and query multiset-valued attributes [3,6,13,16]. Furthermore, the need to efficiently store and query the multisets appears also in the information retrieval [**?**], the data mining [**?**], and in the area of expert systems [**?**].

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 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 [**?**]. 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 include the inverted file [**?**], signature tree [] or B+ tree [9]. These solutions provide a key-value look-up for elements of a multiset implying that containment operations are still element-wise. (? Is it possible to state more precisely the drawbacks of these solutions? ?)

To improve the efficiency of containment operations, we propose a data structure *multiset-trie* that is designed for storage and processing of a finite bounded set of multisets. It extends the *set-trie* data structure proposed by Savnik [14] that was designed for storage and processing of a finite bounded (?or unbounded?) set of sets. Set-trie is a trie based data structure that provides a fast search and retrieval of sets and implements set-containment operations. Multiset-trie generalizes set-trie and can work with a set of sets (as set-trie) or with a set of multisets providing efficient multiset-containment operations.

The multiset-trie is an n-ary tree based data structure with properties similar to those of a trie. Multisets are associated with a collection of nodes in a tree such that every node represents an element of a multiset with particular multiplicity bounded by a node degree n. Multiset-trie is a kind of search tree. Similarly to a trie, it uses common prefixes for shared data representation. Unlike the compact prefix tree, 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. The multiset-trie handles multisets directly by having access to each of the element without the need to reconstruct them for processing, which allows fast retrieval and containment operations. In particular, it supports submultiset and supermultiset queries. The operations allow to find and retrieve the closest submultisets or supermultisets as well as to find and retrieve all of them.

Empirical studies on real data show, that multiset-trie is sensitive to the context, which can be further used to optimize data structure for particular data. Moreover, knowing the data it is possible to estimate the overall performance of the multiset-trie both time and space related using our mathematical theory that describes multiset-trie. The comparative study shows how efficient multiset-trie is by outperforming inverted file in both equality and containment queries by up to 4 orders of magnitude in the time consumed by query.

In the following Section 2 we present the organization of multiset-trie data structure in detail. Next, in Section 3 we present operations that multiset-trie currently supports. This includes multiset containment and the basic search tree functions. The algorithms in pseudo code are presented as well. The description of multiset-trie functions and procedures 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 time complexity of the algorithms and space complexity of the structure. Further, in Section 5 we present an empirical study of the multiset-trie. Synthetic and real-world data sets are used in experiments to test the performance of the data structure. The experiments also highlight the methods for optimizing a multiset-trie. The related work is reviewed in Section 6. Finally, we conclude with the discussion of future work 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 submultiset and supermultiset that will be discussed in the next section in greater details.

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

The multiset-trie is an n-ary tree based data structure with the properties of 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 two 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* $u \to v$ represents a symbol $s \in \Sigma$ with multiplicity j, such that $\phi^{-1}(i) = s$. Such a transition $u \to v$ 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 $u \to v$ 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 end points at 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.

Let us have an example of a multiset-trie data structure. Let $\sigma = 2$ and $\Sigma = I = \{1, 2\}$ respectively, so the mapping ϕ is an identity mapping. Fix the degree of a node $n = 3$, so the maximal multiplicity of an element in a multiset is $n - 1 = 2$. The figure 1 presents the multiset-trie that contains multisets $\emptyset$, $\{1, 1, 2\}$, $\{1, 2, 2\}$, $\{2\}$, $\{1, 2\}$, $\{2, 2\}$. The *Null* children are omitted on the figure.

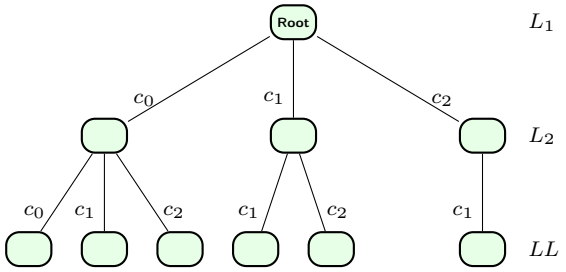

Fig. 1 Example of multiset-trie structure.

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 the figure 1 we can extract inserted multisets as follows:

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

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 $\sigma = 2$:

Multiset		Instance of type Multiset	
$\{1, 1, 2\}$	$\cong$	2	1
		1	2

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 sub tree 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 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

```
1: procedure INSERT(M, m)
2:     currentNode ← M
3:     for i = 1 to σ do
4:         if child c_{m[i]} of currentNode is Null then
5:             create new child c_{m[i]} of currentNode
6:         currentNode ← c_{m[i]}
7:     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

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

3.3 Delete

The 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 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 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.

Algorithm 3 Function DELETE

```
1: function DELETE(M, m)
2:     currentNode ← M
3:     parent ← currentNode
4:     position ← 1
5:     for i = 1 to σ do
6:         if child c_{m[i]} of currentNode is Null then
7:             return False
8:         numChildren ← 0
9:         for j = 0 to n − 1 do
10:            if child c_j of currentNode is not Null then
11:                numChildren ← numChildren + 1
12:        if numChildren is not 1 then
13:            parent ← currentNode
14:            position ← i
15:        currentNode ← c_{m[i]}
16:    child c_{m[position]} of parent ← Null
17:    return True
```

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 submultiset inclusion. If an exact match is not found in $\mathcal{M}$, the function uses multiset-trie to find the closest (the largest) submultiset of m in $\mathcal{M}$ by decreasing multiplicity of 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 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 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

```
1: function submsetExistence(M, m, dev)
2:     currentNode ← M
3:     if currentNode is accepting then
4:         return True
5:     for i = m[1] down to max(0, m[1] − dev) do
6:         if child c_i of currentNode is not Null then
7:             if submsetExistence(c_i, m[2 :], dev) then
8:                 return True
9:     return False
```

3.5 Super-multiset existence

The function SUPERMSETEXISTENCE$(\mathcal{M}, m, dev)$ checks if there exists supermultiset 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) supermultiset 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 function reaches some level where it meets a *Null* node and can not go further using 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 supermultiset $x \supseteq m$ in $\mathcal{M}$. The function SUPERMSETEXISTENCE is presented in Algorithm 5.

Algorithm 5 Function SUPERMSETEXISTENCE

```
1: function supermsetExistence(M, m, dev)
2:     currentNode ← M
3:     if currentNode is accepting then
4:         return True
5:     for i = m[1] to min(n − 1, m[1] + dev) do
6:         if child c_i of currentNode is not Null then
7:             if supermsetExistence(c_i, m[2 :], dev) then
8:                 return True
9:     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/supermultiset, but store the results and continue procedure until all existing sub/supermultisets 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 submultisets 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

```
1: result ← empty container
2: x ← empty array of size σ
3: function getAllSubmsets(M, m, dev)
4:     currentNode ← M
5:     if currentNode is accepting then
6:         add copy of x to result
7:     for i = m[1] down to max(0, m[1] − dev) do
8:         if child c_i of currentNode is not Null then
9:             x[1] ← i
10:            getAllSubmsets(c_i, m[2 :], dev)
```

Algorithm 7 Function GETALLSUPERMSETS

```
1: result ← empty container
2: x ← empty array of size σ
3: function getAllSupermsets(M, m, dev)
4:     currentNode ← M
5:     if currentNode is accepting then
6:         add copy of x to result
7:     for i = m[1] to min(n − 1, m[1] + dev) do
8:         if child c_i of currentNode is not Null then
9:             x[1] ← 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 our mathematical model that we use to describe the distribution of multisets

in the multiset-trie and input data. Using 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 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 presence of the emptyset in the multiset-trie.

Finalizing the mathematical analysis, we present the study of space complexity of the multiset-trie in the 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 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 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 multiset-trie. Let n be the maximal degree of a node in 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 an 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 a 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 $\xi_1, \xi_2, \ldots, \xi_{\sigma+1}$ 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 $\forall\ i \in [1, \sigma]$ 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 existence of any individual node.

Lemma 1 *Any potential node on a fixed level i, where $i \in \{1, 2, \ldots, \sigma + 1\}$ 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 sub tree with the root v and call it v-sub tree. Since the height of the multiset-trie is $\sigma + 1$ we can calculate the height of the v-sub tree. Taking in account that the root node has height 1, the height of the v-sub tree 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 sub tree, i.e. a node on the level $\sigma + 1$ that belongs to v-sub tree. The possible number of nodes on the leaf level of v-sub tree 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-sub tree 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 type.

Lemma 2 *Suppose that a node v exists at level $1 \leq i \leq \sigma$. 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 $k \in \{1, 2, \ldots, n\}$ children. Let $X_0, X_1, \ldots, X_{n-1}$ be random variables, they are defined as follows:

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

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. □

Knowing the probability density and probability generating functions of ξ_{ij} from Lemma 2, we now can 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 [5] 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).

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 $1 \leq i \leq \sigma$.

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).

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

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

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

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 submultisets or all the supermultisets of m. In order to calculate the expected cardinality of such subtrees we need the following definition.

Definition 1 Let $m = \{1^{k_1}, 2^{k_2}, \ldots, \sigma^{k_\sigma}\}$, 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 submultiset or supermultiset 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 (submultiset and supermultiset). □

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 submultisets or supermultisets has worst case running time complexity 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)

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

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 mulitset-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 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 submultiset or supermultiset 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 supermultiset 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.

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

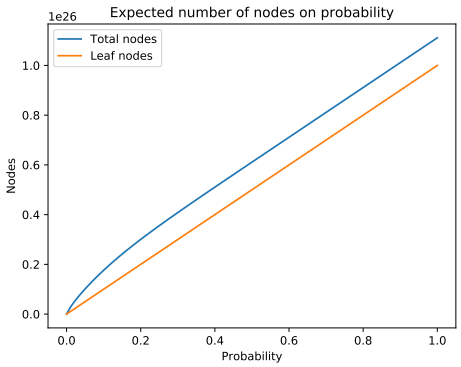

Fig. 2 $\mathbb{E}(|\mathcal{M}|)$ and $\mathbb{E}(|M|)$ on probability.

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.

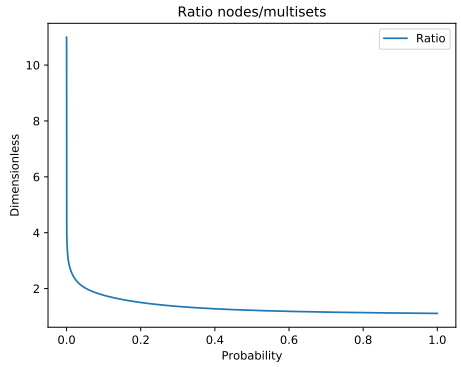

Fig. 3 Ratio $\mathbb{E}(\frac{|\mathcal{M}|}{|M|})$ on p.

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 of 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 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 implementation of multiset-trie is done 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 [1]. The implementation of the program was optimized for testing and therefore, the program operates with files, in order 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 about the parameters that will be varied throughout the experiments and discuss the experimental data that was used.

Let M be a set of multisets that are 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 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 next sections we will present the behavior of the multiset-trie data structure depending on the selected parameters as well as the comparative benchmark of the multiset-trie against B+ tree implementation of inverted index. We start with experiments that are performed on an artificially generated data in order to give a general picture of the multiset-trie performance. 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 maximal 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 non linearity of the performance with respect to the density of the multiset-trie. The next experiment on the multiset-trie uses the real world data. In Experiment 4 the influence of the mapping ϕ is studied. The input data is obtained by mapping of the real words from 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. After all the experiments we present an empirical comparison of multiset-trie data structure with B+ tree based inverted index. We use inverted index to store and retrieve multisets in the same way as it is described in the paper by Helmer and Moerkotte [9] for sets. In the comparison we use three types of queries exact, submultiset and supermultiset retrieval.

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 is 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 a 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

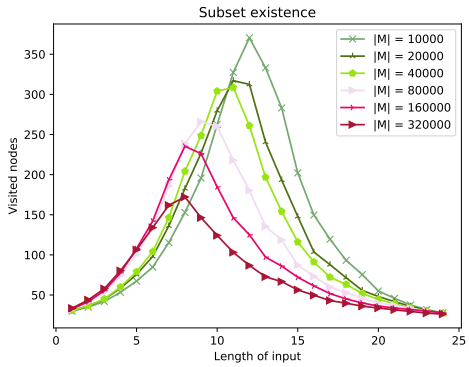

Fig. 4 Experiment 1, submsetExistence function.

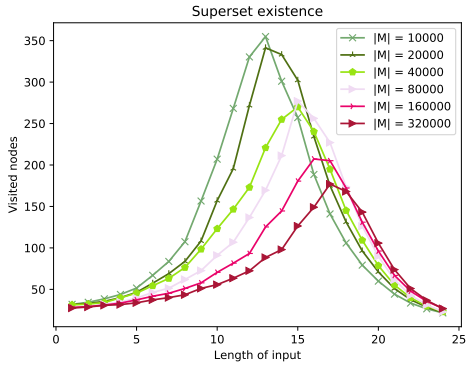

Fig. 5 Experiment 1, supermsetExistence function.

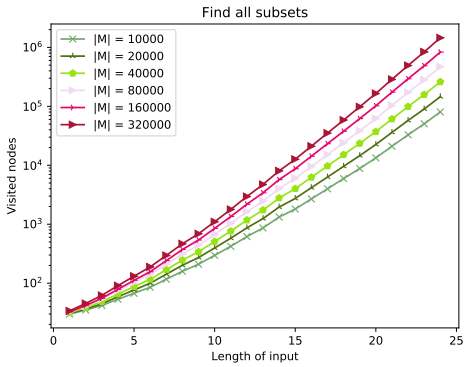

Fig. 6 Experiment 1, getAllSubmsets function.

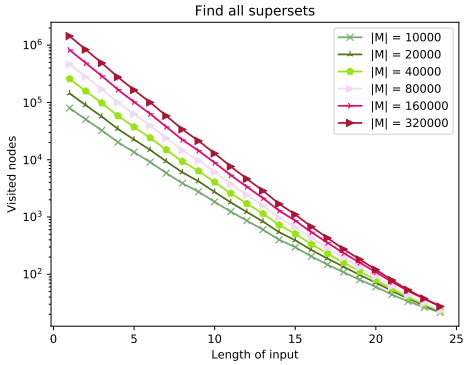

Fig. 7 Experiment 1, getAllSupermsets function.

experiments in order to measure the number of visited nodes for the queries with test multiset of a distinct size. 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 [14]. 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 4 and 5). The results are as expected, because the increase of the density increases the probability of finding submultiset or supermultiset in multiset-trie, which leads to the 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 submultiset or supermultiset, 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 to appear in M, the distribution of set sizes in M is normal. Consequently, with increase of 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 6 and 7). 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 supermultisets. 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 multist-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.

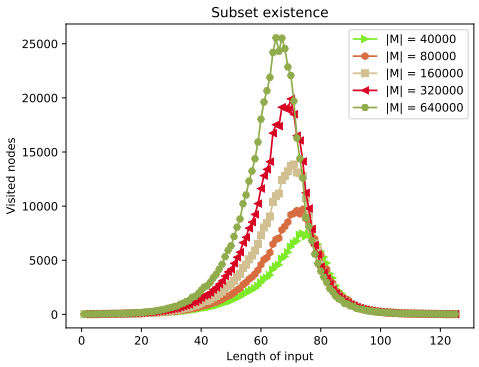

Fig. 8 Experiment 2, submsetExistence function.

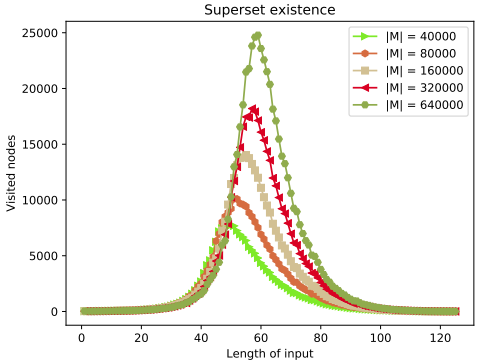

Fig. 9 Experiment 2, supermsetExistence function.

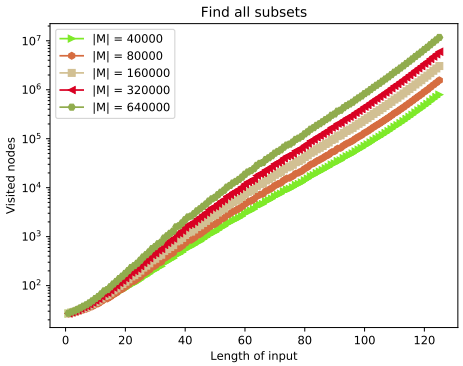

Fig. 10 Experiment 2, getAllSubmsets function.

5.2 Experiment 2

In the Experiment 2 we demonstrate the performance of the unrestricted multiset-trie allowing *multisets* to be inserted into 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 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 $6^{25} = 2.84 \times 10^{19}$.

As we can see from the graphs on figures 8 and 9, 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 become more sparse. Multisets in a sparse multiset-trie differs more, which leads to the larger number of visited nodes.

The maxima for both functions varies from 7500 to 25000 visited nodes. According to those maxima at least 300-1000 multisets were checked in order to find submultiset or supermultiset, 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

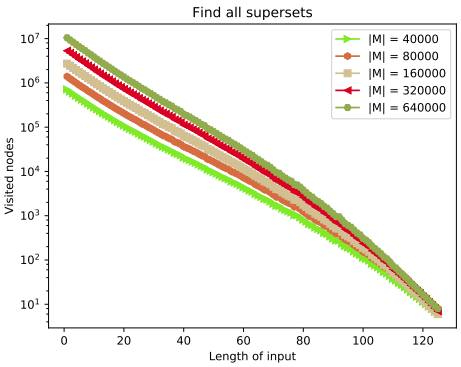

Fig. 11 Experiment 2, getAllSupermsets function.

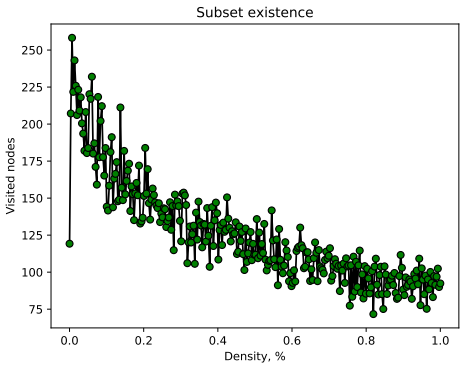

Fig. 12 Experiment 3, submsetExistence function.

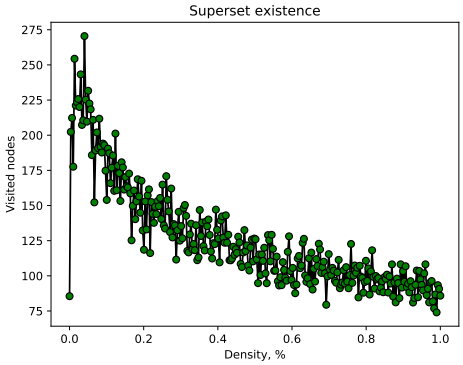

Fig. 13 Experiment 3, supermsetExistence function.

complete multiset-trie, then in 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 a behavior was previously observed in the Experiment 1. The explanation is the same: the input data has uniform distribution, implying that the size of multisets in M is normally distributed. Because of the normal distribution of size of multisets the shift of the peak occurs as the density increases.

It can be also 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 10 and 11). It happens, because the number of multisets increases as the density increases. So there are more nodes have to be visited in order to retrieve all sub- or supermultisets 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 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 of such a behavior is that the dependence of the number of visited nodes on density is not a linear function. It is obvious that the performance of the mentioned above 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, when the density is high, multisets differ less, 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 particular density.

As we see on figures 12 and 13 both functions SUBMSETEXISTENCE and SUPERMSETEXISTENCE have the maximum around $d \approx 7.0 \times 10^{-5}$. The maximum is less

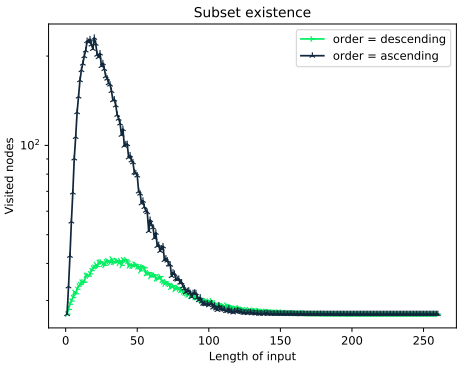

Fig. 14 Experiment 4, submsetExistence function.

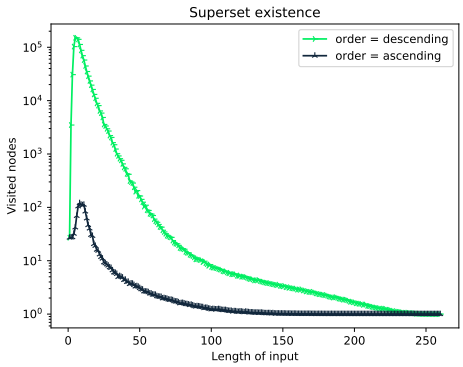

Fig. 15 Experiment 4, supermsetExistence function.

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 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 an input data.

The data is taken from 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 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 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 14 and 15 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. It means that in order to give an answer whether some multiset exists or not one have 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.

5.5 Experiment 5

In this experiment, we demonstrate the performance of multiset-trie data structure compared to the inverted index based on the B+ tree. Both data structures are implemented in the programming language C++, providing in this way an experimental setup for a fair comparison [ref. to impl].

((Presentation of the inverted index implementation? What about the smallest postings first?) – (What do you mean by "smallest postings"?))

The experiment uses the input data for the construction of the given data structure and the test data for the execution of the operations on the given data structure. The input data comprises a set of randomly generated multisets used for the construction of a data structure. The test data includes the set of multisets together with the operations that are evaluated. The input and test data were generated with respect to parameters σ and n as presented in Table 1.

((Is there anything special to be said about the artificially generated data?) – (Not really. We say that

the data is generated with respect to ms-trie parameters and it is random. Other detail, such as how we generated the data (with Python) does not really matter.))

σ	n
5	1
30	1
5	3
15	3
30	3
10	10

Table 1 Configuration of σ and n in benchmark.

We tested all three types of a query on all of the configurations from Table 1 which results in 18 experiments total, i.e., 6 experiments per query type. In each of the experiment, we measured an average time consumed by the data structure to process the query. The results of the exact search, sub-multiset and super-multiset search experiments are presented in tables Table 2, Table 3 and Table 4, respectively.

σ	n	Multiset-trie (μs)	Inverted index (μs)
5	1	3.45	17782.35
30	1	4.18	24865.93
5	3	2.20	1508.81
15	3	4.39	2146.36
30	3	10.67	3639.97
10	10	6.93	384.05

Table 2 Exact search.

σ	n	Multiset-trie (μs)	Inverted index (μs)
5	1	8.96	73500.84
30	1	17.33	547572.74
5	3	117.95	162360.43
15	3	20.74	443321.39
30	3	23.75	947706.14
10	10	55.59	466022.68

Table 3 Sub-multiset search.

σ	n	Multiset-trie (μs)	Inverted index (μs)
5	1	10.63	63073.86
30	1	14.65	449251.68
5	3	171.04	163256.77
15	3	43.42	425733.80
30	3	22.06	729831.34
10	10	58.32	373784.81

Table 4 Super-multiset search.

We can see that multiset-trie outperforms inverted index in all of the experiments by up to 4 orders of magnitude. In an exact search, multiset-trie has to traverse only up to $\sigma + 1$ nodes to get query result. It can be seen from results that with the increase of σ the processing time for multiset-tire also increases. Multiplicity also affects the processing time; however, this happens passively. ((equivalent to $\sigma + 1$?) – (No, much smaller impact. In fact, if we consider the same data set and vary σ and max multiplicity, the multiplicity will have almost no influence. The σ will affect every single lookup of a multiset in ms-trie, when multiplicity will only influence supermultiset queries, where we will check one extra multiplicity.)) Multiplicity, or degree of a node n, defines the shape of multisets that are stored in multiset-trie. Thus, it affects the structure and density of the multiset-trie.

As for the inverted index, all three operations must first fetch all the postings for each particular element of a test multi-set. Afterward, the intersection of postings is computed to answer the query. The operations use more processing time than a simple tree traversal, which we can see from results. Postings are filtered on-the-fly to reduce the cost of the intersection. ((I do not get this: "Note that inverted index will also access at most σ postings during search, but in addition to access some extra processing is required.") – (Inverted index will search for multisets in all of the postings (σ postings) in the worst case. Ms-trie needs to traverse $\sigma + 1$ nodes to lookup a multiset. So the access time would be approximately the same. However, inverted index requires additional processing of postings to search for a multiset when the ms-trie does not.))

Implementations of the sub-multiset and super-multiset are similar in the case of the inverted file. The algorithm consists of the same steps. First, the postings are fetched for each element of the test multiset. Depending on the particular operation, postings are filtered on-the-fly. Finally, the union or the intersection of the filtered set of postings is computed. Note that the processing time increases with the size of the inverted index because of the increased sizes of postings.

((Can you characterize the sub-tree (num of nodes visited) visited by an operation more precisely? Can we say something about this? What is the size of the sub-tree visited by an operation? Why it is not very large? I did not try to finalize the following text.) – (The sub-tree we visit is quite simple to picture. Take a multiset and draw the complete path defined by this multiset in the ms-trie. Everything that is on the left side of this path will be visited by submset query, everything on the right will be visited by supermset query. The size of this sub-tree we actually described in the math

analysis. There is a formula that is parameterized by input multiset. I think I don't say that it is not large anywhere. It can be the whole tree.)) In the case of multiset-trie, only a traversal of the tree is required which is much faster than the processing of postings as we can see from results. In the worst case the whole tree is traversed, but so is for an inverted file. ((Can we see how...?) – (The same way as with ms-trie. Just search for a submsets of some large multiset that would result in retrieving the whole structure.))

6 Related work

[Will be rewritten]

6.1 Multiset

Multiset is a widely used data structure in different areas of mathematics, physics and computer science [15]. The theory of multisets is based entirely on the theory of sets. However, classical mathematics does not deal with multisets directly. Instead, one can define a multiset to be a *family* of sets or the functions on ordered pairs, where the members of a pair are an element and its multiplicity. This means that mathematically the concepts of set such as cardinality, set-containment operation, power set, equivalence classes and others are well defined for multisets in terms of sets [2].

The concept of a multiset can also be referred to the *bag-of-words model*. This model takes its origin from a linguistic context studied by Harris [7]. According to the bag-of-words model, text can be represented as a bag (multiset) of words, where an element is a word and the number of its occurrences in the text is multiplicity. A bag of words does not keep track of grammar and ordering of words.

6.2 Information retrieval

Information retrieval (IR) refers to a problem of finding material of an unstructured nature that satisfies an information need [12]. Usually, one is searching for a specific documents in a significantly large text documents database. The size of a database makes the search a time consuming operation. In order to resolve the issue IR systems pre-process data and create indexes for future use in search operation.

The bag-of-words model is widely used in IR. In particular, such a representation of text documents is used in database indexes when a full text search of a database is required. The full-text search problem refers to indexing techniques for full-text databases. The most efficient index nowadays uses the concept of an inverted index [17].

The proposed data structure multiset-trie can be used as an alternative implementation of the search structure of an inverted index. It represents words as multisets and stores them into data structure. The query processing is achieved using boolean retrieval model [12] and multiset containment operations. Multiset containment operations of the multiset-trie implement the nearest neighbor search queries which retrieve not only exact but also the most relevant results to a user.

6.3 Generalized search tree

The properties and operations of the multiset-trie makes it a competitor to the most efficient implementation of a search tree the *Generalized Search Tree (GiST)* [4,8, 10] that is used in inverted index. GiST is a very flexible data structure that can be customized in order to behave like B+-tree, R-tree or RD-tree. It also provides support for an extensible set of queries and data types that B+-tree, R-tree or RD-tree do not support originally. GiST supports all the basic search tree operations such as insert, delete and search, and in addition provides such extensions as the nearest-neighbor search and multiset containment operations. The extensions provided by GiST are native in the multiset-trie. The multiset-trie also has a fixed height while GiST is a self-balanced tree and has to use additional methods in order to preserve its balance.

6.4 Set-trie

The multiset containment queries are well studied in the area of relational databases. The queries are well-defined in the context of relational algebra [11]. This problem was previously studied for a restricted case of queries. In particular, the storage and fast retrieval of sets were previously accomplished in data structure *set-trie* proposed by Savnik [14].

The set-trie data structure is based on a trie data structure. It supports set containment operations such as retrieval of the nearest sub- and supersets and retrieval of all sub- and supersets from the data structure.

The data structure multiset-trie adapts the properties of the set-trie implementing the functions SUBMSETEXISTENCE, SUPERMSETEXISTENCE, GETALLSUBMSETS and GETALLSUPERMSETS together with the basic tree functions such as INSERT, DELETE and SEARCH. Moreover, multiset-trie extends the abilities of the set-trie allowing to store and retrieve multisets. The down-

side of such an extension is that multiset-trie no longer supports path compression that was obtained in set-trie. However, the design of multiset-trie provides a constant worst case time complexity of search function independently of user input.

7 Conclusions and future work

[Will be rewritten] 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 the Experiment 4. Pre-processing determines the optimal encoding for input data and ensures the best performance of the multiset-trie on particular input data. In case of storing words in the multiset-trie, the search queries can be always optimized based on the frequencies of letters in a specific language. We also see from Experiments 1 and 2 that 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 it is 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 of the space complexity shows that multiset-trie requires only $O(|M|)$ space, which is the minimal possible space that is required by any data structure for storage of $|M|$ objects. As for the running time complexity of the 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).

It can also be concluded that the multiset-trie is an input sensitive data structure, because the size of multiset-trie $|\mathcal{M}|$ depends on the distribution of multisets in M. Our mathematical model assumes that multisets m in M are distributed uniformly. However, in a real world models such an assumption is not true in a lot of the cases. Specifically, the probability $P(m \in M)$ may vary dramatically and can be even equal to 0. For example, if words are mapped to multisets, then the sample space contains very large multisets. Nonetheless, most of them will have zero probability to appear in M, because a word that would correspond to such a multiset simply does not exist.

The above results have opened even more interesting questions for the future research. 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 the types of sub and supermultisets can be specified. As an example, the multiplicity of an element in a multiset can be bounded in operations getAllSubmsets and getAllSupermsets. Such functionality would allow more specific queries of multisets. The second line of research is to investigate the multiset-trie as an index data structure in detail. It will be very interesting to study the comparison of the multiset-trie with other existing index data structures.

References

1. M. Akulich. Mstrie repository. `https://github.com/nick-ak96/mstrie`, 2019.
2. W. D. Blizard et al. Multiset theory. *Notre Dame Journal of formal logic*, 30(1):36–66, 1988.
3. P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, 49(1):375–402, 2016.
4. A. Z. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *International Conference on Extending Database Technology*, pages 313–330. Springer, 2006.
5. C. W. Gardiner. *Stochastic methods*. Springer-Verlag, Berlin–Heidelberg–New York–Tokyo, 1985.
6. V. Gripon, M. Rabbat, V. Skachek, and W. J. Gross. Compressing multisets using tries. In *Information Theory Workshop (ITW), 2012 IEEE*, pages 642–646. IEEE, 2012.
7. Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.
8. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. *Generalized search trees for database systems*. September, 1995.
9. S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, Oct 2003.
10. M. Kornacker. High-performance extensible indexing. In *VLDB*, volume 99, pages 699–708, 1999.
11. G. Lamperti, M. Melchiori, and M. Zanella. On multisets in database systems. In *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pages 147–216, London, UK, UK, 2001. Springer-Verlag.
12. C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.
13. K. A. Ross and J. Stoyanovich. Symmetric relations and cardinality-bounded multisets in database systems. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 912–923. VLDB Endowment, 2004.
14. I. Savnik. Index data structure for fast subset and superset queries. In *International Conference on Availability, Reliability, and Security*, pages 134–148. Springer, 2013.

15. D. Singh, A. Ibrahim, T. Yohanna, and J. Singh. An overview of the applications of multisets. *Novi Sad Journal of Mathematics*, 37(3):73–92, 2007.
16. C. Steinruecken. Compressing sets and multisets of sequences. *IEEE Transactions on Information Theory*, 61(3):1485–1490, 2015.
17. J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 352–352. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1992.