Data structure for set containment queries: theoretical and empirical analysis

Iztok Savnik[a,*], Matjaž Krnc[b,a], Riste Škrekovski[c,d]

[a] *Faculty of mathematics, natural sciences and information technologies, University of Primorska, Slovenia*
[b] *Department of Computer Sciences, University of Salzburg, Austria*
[c] *Department of Mathematics, University of Ljubljana, Slovenia*
[d] *Faculty of Information Studies, Novo Mesto, Slovenia*

Abstract

Set containment operations are an important tool in various fields, such as the datamining tools, the object-relational databases, the rule-based expert systems, and the AI planning systems. In the paper, a set-trie data structure for storing sets is presented, along with the efficient algorithms for the corresponding set containment operations. In addition, we present the mathematical and empirical study of the set-trie. In particular, the expected performance of the data structure is analyzed by a probabilistic model, and some relevant upper-bounds on its efficiency are determined. The empirical study was designed to give insight in the time-complexity space of the set containment operations. The experimental results confirm the mathematical analysis.

Keywords: subset queries, set containment queries, partial matching, access methods, database index

1. Introduction

Set containment queries are frequent in various systems including the datamining tools, the object-relational databases, the rule-based expert systems, and the AI planning systems. Let us first present some insight into the use of set containment operations in these fields.

The enumeration of subsets of a given universal set Σ is very common in *data mining* algorithms [11] where sets are used as the basis for the representation of hypotheses and the search space forms a lattice. Often we have to see if a given hypothesis has already been considered by the algorithm. This can be checked by searching the set of hypotheses (sets) that have already been processed. Furthermore, in some cases the hypotheses can be easily overthrown if a superset hypothesis has already been shown not valid. Such problems include the discovery of association rules, functional dependencies as well as some forms of propositional logic [11, 19, 5, 8].

In the *object-relational database management systems* tables can have set-valued attributes, i.e., the attributes that range over sets. The set containment queries can express either the selection or join operation based on the set containment condition. Efficient access to the relation records based on the conditions that involve set operations are vital for the fast implementation of such queries [13, 21, 7].

The rule-based expert systems use the set containment queries to implement fast pattern-matching algorithms that determine which rules are fired in each cycle of the expert system execution. Here the sets form pre-conditions of rules composed of the elementary conditions. Given a set of valid conditions the set of fired rules includes those with the pre-condition included in this set [6, 4].

Finally, in the *AI planning systems*, the goal sets are used to store the goals to be achieved from a given initial state. Planning modules use the subset queries in the procedure that examines if a given goal set is satisfiable.

*Corresponding author
Email addresses: iztok.savnik@upr.si (Iztok Savnik), matjaz.krnc@upr.si (Matjaž Krnc), skrekovski@gmail.com (Riste Škrekovski)

A part of the procedure represents querying the goal sets that were previously shown to be unsatisfiable. Here also the sets are used to form the basic structure of hypothesis space [2].

In this paper we present a novel index data structure *set-trie* that implements efficiently the basic two types of the set containment queries: the *subset* and *superset queries*. The initial implementation of the set-trie has been done in the frame of the data-mining system FDEP [17]. The preliminary study of set-trie based on the multi-sets has been published in [20]. In this paper we present the theoretical analysis of set-trie, and, the comprehensive empirical analysis based on the efficient implementation of set-trie [16].

The set-trie is a tree data structure derived from the *trie* [14]—while tries are used to store and search words, the set-trie is used to store and query sets. The possibility to extend the performance of usual trie from the membership operation to the set containment operations comes from the fact that we are storing *sets* where the ordering of elements is not important, while ordinary tries are used to store the *sequences* of symbols where the ordering of symbols is important. As it will be presented in the paper, the ordering of set elements is used as the basis for the definition of the efficient algorithms for the set containment operations.

In the mathematical analysis, the expected performance of the data structure is analyzed by a probabilistic model. Standard tools such as Bernoulli distribution and Galton-Watson branching stochastic process are used. As a result, some relevant upper-bounds on the time-complexity are determined. While listing all subsets in the function GETALLSUBSETS can never have subexponential guarantee, we managed to determine an additional upper-bound for the EXISTSUBSET function. Furthermore, we observe that, for the method GETALLSUPERSETS, the input sets with low-ranked members perform faster than these with high-ranked ones. For the function EXISTSUPERSET, we observe that in most cases, the expected number of visited nodes will not be large.

We analyze all four set containment operations empirically on the artificially generated data. In the experiments we observe the influence of several parameters on the performance of the set containment operations: the size of sets, the size of input set-trie, the ordering of symbols in the alphabet, and the shape of test-sets. In Experiment 1 we investigate the influence of the size of sets on the performance of the set containment operations. We show the basic shapes and the experimental upper-bounds of the curves presenting the number of visited nodes for all operations. Furthermore, we notice that the results of the operations EXISTSSUBSET and EXISTSSUPERSET can be seen as dual. Experiment 2 gives some insight into the influence of the size of the alphabet to the behavior of operations. In brief, the curves for all four operations scale almost linearly with the increasing size of alphabet. Finally, Experiment 3 presents the effects of the shape of input sets to the performance of the set containment operations. In particular, for the various set-tries and the chosen input-lengths, we generate four fundamentally different inputs (with respect to the choice of its symbols), and study the influence between the structure of the input-word and the performance of operations. The subset and superset operations exhibit the dual behavior again, but in a stronger way then previously observed. The operation EXISTSSUPERSET works much faster for the test-sets with lower indexes than the operation EXISTSSUBSET. On the contrary, the operation EXISTSSUBSET works much faster for the test-sets that include higher indexes than the operation EXISTSSUPERSET.

Let us now state the contributions of this paper. While the novel data structure set-trie was originally proposed in [20], this paper presents the thorough mathematical analysis and the detailed empirical analysis of the set-trie data structure. Firstly, the contributions of the mathematical analysis are the estimations of the upper-bounds of the time-complexity for the set containment operations. Secondly, the presented experiments give the detailed insight into the structure of the search space of set containment operations as well as the experimental upper-bounds for all operations. Finally, one of the results of the empirical analysis is the demonstration of the duality of subset and superset operations in the context of the data structure set-trie.

The paper is organized as follows. Section 2 presents the data structure set-trie together with the operations for searching the subsets and supersets. We give the precise presentation of the operations INSERT, EXISTSSUBSET, EXISTSSUPERSET, GETALLSUBSETS and GETALLSUPERSTS. Section 3 presents the mathematical analysis of set-trie. The description of the mathematical model is given in Section 3.1. The estimation of the number of visited nodes in the subset operations is studied in Section 3.2. The estimation of the number of visited nodes of the superset operations is presented in Section 3.3. Section 4 describes an empirical analysis of set-trie. The subsequent sections give detailed commentaries of the three experiments including the explanations of the behavior of particular operations and the characterization of the search space for all operations of the data structure set-trie. In Section 5 we presents the related work. The data structures from the research area

of the algorithms and data structures that have the functionality similar to set-trie are discussed in Section 5.1. AI systems that use the sets for querying hypotheses and states are presented and compared to the set-trie data structure in Section 5.2. The indexes of the object-relational database systems that are used to access set-valued attributes are presented in Section 5.3. Finally, the conclusions and the directions of our further work are given in Section 6.

2. Data structure set-trie

Let suppose we have a set of symbols $\{s_1, s_2, \ldots, s_\sigma\}$ that represents an alphabet Σ of the size σ. We would like to store efficiently a set of sets of the elements from Σ. The data structure for storing the set of sets should provide efficient algorithms for the operation insert, the membership operation and the set containment operations.

A syntactical order of symbols from Σ can be defined by assigning each symbol a unique index. The indexes are the elements of the set $\{1, 2, \ldots, \sigma\}$. The assignment of the indexes to the symbols from Σ is used to obtain the unique representation of a set of symbols. It can be represented by means of a set of integer numbers. Furthermore, it turns out that careful definition of the syntactical order of Σ can contribute to the efficient implementation of the set containment operations.

To simplify the presentation of the data structure set-trie, but without any loss of generality, we assume that $\Sigma = \{1, \ldots, \sigma\}$. Since the elements of Σ are totally ordered, we can exploit this ordering for the efficient representation of a set of sets, as well as, for the design of efficient algorithms for the set containment operations. Therefore, a set can be represented by an ordered sequence of integer numbers from Σ. This is the representation of sets that we use for storing sets in the data structure set-trie.

The data structure *set-trie* is a tree composed of nodes labeled with indexes from 1 to σ. It is structured as follows:

- The root node is labeled with $\emptyset$ and its children can be the nodes labeled from 1 to σ. A root node alone represents an empty set.
- A node labeled i can have children labeled with the indexes that are greater than i.
- Each node can have a flag *last_flag* denoting the last element in the set.

A set is represented by a path from the root node to a node with *last_flag* set to true. Note that the path is composed of nodes labeled by the indexes that are increasing along the path. Let us give an example of a set-trie. Figure 2 presents a set-trie containing the sets $\{1, 3\}, \{1, 3, 5\}$, $\{1, 4\}, \{1, 2, 4\}$, $\{2, 4\}$ and $\{2, 3, 5\}$. Note that flagged nodes are represented with circles.

Let S be a set-trie, X a set of indexes from Σ, and, L an ordered list of indexes from X such that $L[1]$ stores the smallest index and $L[|X|]$ holds the highest index from X. A set X is in a set-trie S represented by a path p from the root of S to some node that has the flag *last_node* set to true. The path p is composed of the nodes labeled by the indexes that correspond to the elements of the ordered list L. As with the ordinary tries, the prefixes that overlap are represented by a common path from the root to an internal vertex of set-trie tree.

The operations for searching the subsets and supersets of a set X in a set-trie S use the ordering of indexes. The algorithms do not need to consider the tree branches for which we know they do not lead to the results. The search space for a given X and a set-trie S can be seen as a sub-tree, or, a strip of the set-trie S of the size that depends on the set X and the set-trie S.

2.1. Operations on sets

Let $X \subseteq \Sigma$ and S be a set-trie that represents a collection of sets $W = \{s_i \mid s_i \subseteq \Sigma\}$. To simplify the presentation, we write $X \in S$ if $X \in W$. We are interested in the following operations:

1. INSERT(S, X) inserts the set X into the set-trie S;

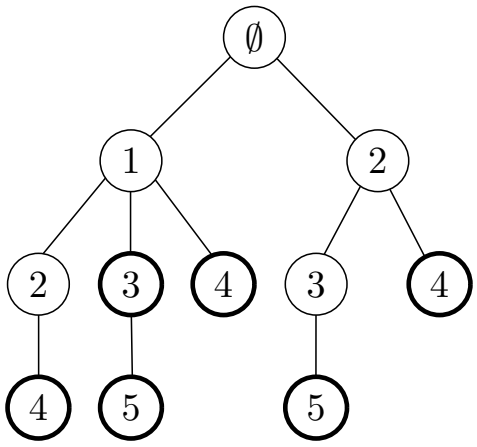

Figure 1: An example of a set-trie.

2. SEARCH(S, X) returns *true* if $X \in S$ and *false* otherwise;
3. EXISTSSUBSET(S, X) returns *true* if $\exists Y \in S : Y \subseteq X$ and *false* otherwise;
4. EXISTSSUPERSET(S, X) returns *true* if $\exists Y \in S : X \subseteq Y$ and *false* otherwise;
5. GETALLSUBSETS(S, X) returns all sets Y such that $Y \in S \wedge Y \subseteq X$; and
6. GETALLSUPERSETS(S, X) returns all sets Y such that $Y \in S \wedge X \subseteq Y$.

Let us now present the data structure for storing sets that will be used for the presentation of the operations. Since the sets of indexes from Σ, that we use in the algorithms, rely on the ordering of indexes we need a special data structure for the representation of sets. The data structure *Set* is represented as a class in an object-oriented programming language: it has a state, and, a set of operations.

The state of the *Set* instances includes the *current element* of the set. The current element is a concept similar to file pointer that points to some position in a file. Analogously, the current element of a set is an element of the set that we currently observe. Let $X \subseteq \Sigma$ and $c \in X$ is the current element of a set X. The *next element* of c is the smallest n such that $n \in X$ and $c < n$ with respect to the ordering of Σ.

Let X denote an instance of the class *Set*. The class *Set* has the following methods. The method X.first() sets the current element of the set to the element of X that has the smallest value. The operation X.existsCurrent() checks if there exists the current element in the set X. The method X.current() returns the current element of the set X if it exists, and, returns an error otherwise. The method X.next() returns the next element of the current element, and, $\sigma + 1$ if there is no such element. The current element is not defined after the method next() is applied to the set X, where the current element is the last (the greatest) element of X.

Algorithm 1 Procedure INSERT(V, X)

```
1: procedure INSERT(V, X)
2:     if X.existsCurrent() then
3:         if exists child of V labeled X.current() then
4:             U ← retrieve child of V labeled X.current();
5:         else
6:             U ← create child of V labeled X.current();
7:         INSERT(U, X.next())
8:     else
9:         V.last_flag = true
```

2.2. *Operation* INSERT

The first operation of set-trie data structure is *insertion*. The operation INSERT(V, X) enters an instance X of the type *Set* into the set-trie S referenced by a root node V. The operation INSERT is presented in Algorithm 1.

Each invocation of the operation INSERT either traverses through the existing tree nodes, or, creates the new nodes to construct a path from the root to the flagged node corresponding to the last element of the set X.

Algorithm 2 Function SEARCH(V, X)

```
1: function SEARCH(V, X)
2:     if X.existsCurrent() then
3:         if exists child of V labeled X.current() then
4:             U ← child of V labeled X.current()
5:             SEARCH(U, X.next())
6:         else return false
7:     else return V.last_flag
```

2.3. *Operation* SEARCH

The operation SEARCH(V, X) searches for a given X in a set-trie S represented by a tree with the root V. It returns *true* if X is an element of the set-trie represented by V, and, *false* otherwise. The operation SEARCH is presented in Algorithm 2. As in the case of ordinary *trie* data structure, operation SEARCH checks if there exists a path from the root of tree V, labeled with the elements (indexes) of X, to some node flagged as *last_node*.

Let us give some more details about the algorithm of the operation SEARCH. The operation has to be invoked with the call SEARCH(V, X.first()) so that V is the root of the set-trie tree and the current element of X is the smallest index of X. Each activation of SEARCH tries to match the current element of X with the child of V. If the match is not successful it returns *false* otherwise it proceeds with the following element of X.

2.4. *Operations* EXISTSSUBSET *and* GETALLSUBSETS

The operation EXISTSSUBSET(V, X) checks if there exists a subset of X in the given set-trie S with the root V. The subset that we search in S can have fewer elements than X. Therefore, besides that we search for the exact match we can also skip one or more elements in X and find a subset that matches the rest of the elements of X. The operation is presented in Algorithm 3.

Algorithm 3 Function EXISTSSUBSET(V, X)

```
1: function EXISTSSUBSET(V, X)
2:     if V.last_flag then
3:         return true
4:     if not X.existsCurrent() then
5:         return false
6:     found ← false
7:     if exists child of V labeled X.current() then
8:         U ← child of V labeled X.current()
9:         found ← EXISTSSUBSET(U, X.next())
10:    if found then
11:        return true
12:    else
13:        return EXISTSSUBSET(V, X.next())
```

In the initial state of the algorithm, the parameter X has current value set to the first element of the set, and, the parameter V references the root of set-trie. The operation EXISTSSUBSET tries to match elements of X with the child nodes of the set-trie V. In each step, either the current element of the X can be matched with a child

of V, or, the current element of X is skipped and the operation tries to match the next element of X with the same set-trie V.

The first if statement in line 2 checks if a subset of X is found in the tree, i.e., the current node of a tree is the last element of subset and last_flag=*true*. The second if statement in line 4 checks if X has no more elements, and, we did not find the subset in V. The third if statement in line 7 verifies if the parallel descend in X and in the tree V is possible. In the positive case, the algorithm calls EXISTSSUBSET with the next element of X and the child of V corresponding to the matched symbol. Finally, if match did not succeed, current element of X is skipped and EXISTSSUBSET is called with the same V and the next element of X in line 13.

The operation EXISTSSUBSET can be easily extended to find all subsets of a given set X in a tree with the root V. After finding the subset in line 3, the subset must be stored and the search can continue. In addition, instead of checking if a subset has already been found in lines 10-13, the operation GETALLSUBSETS(V, X.next()) would be called to collect all the results. The experimental results with the operation GETALLSUBSETS(V, X) are presented in Section 4.

Algorithm 4 Function EXISTSSUPERSET(V, X)

```
1:  function EXISTSSUPERSET(V, X)
2:      if not X.existsCurrent() then
3:          return true
4:      found ← false
5:      element ← X.current()+1
6:      nextElement ← X.next()
7:      while element ≤ nextElement and not found do
8:          if exists child of V labeled element then
9:              U ← child of V labeled element
10:             if element = nextElement then
11:                 found ← EXISTSSUPERSET(U, X.next())
12:             else
13:                 found ← EXISTSSUPERSET(U, X)
14:         element ← element + 1
15:     return found
```

2.5. *Operations* EXISTSSUPERSET *and* GETALLSUPERSETS

The operation EXISTSSUPERSET(V, X) checks if there exists a superset of X in the set-trie S referenced by the node V. In operation EXISTSSUBSET, we could skip some elements from X to match the sets from the set-trie S. In operation EXISTSSUPERSET we can do the opposite: we can omit some elements in the supersets from S to match the parameter set X. This operation is presented in Algorithm 4.

Let us present the algorithm of the operation EXISTSSUPERSET in more detail. The initial state of the operation is: the current element of the parameter set X is set to the first element, and, the parameter V stores the reference to the root of set-trie S. In each recursive step, the algorithm can either descend to the next current element of X and to the matched child node of set-trie V, or, it can leave the set X unchanged and descend only in the set-trie referenced by V to the selected child node.

The first if statement in line 2 checks if we are already at the end of X. If this is the case, then the parameter X is covered completely with a superset from the set-trie referenced by root V. The code in the lines 6-7 sets the interval of elements (lower and upper bounds) that can be used in the search for the supersets from S. Note that the first element of the interval is the current element of X plus 1, and, the last element of the interval is the next element of X. In each pass of the while loop in the line 7, we either descend in parallel in line 11 on both X and the set-trie referenced by V, in the case that we reach the upper bound of the interval, or, we take the current child and call EXISTSSUPERSET on unchanged X in the line 13. Of course, we can descend to the child node of V only in the case that the child exists which is checked in the line 8.

As in the case of the operation EXISTSSUBSET, the operation EXISTSSUPERSET can be extended to retrieve all supersets of a given set X in a tree with the root V. After X is matched completely in line 3, there remains a subtree of trailers corresponding to a set of supersets that subsume X. This subtree is rooted in a tree node, let say U_k, that corresponds to the last element of X. Therefore, after the U_k is matched against the last element of the set X in line 3, the complete subtree has to be traversed to retrieve all supersets that go through the node U_k.

2.6. Some properties of set-trie

Given the size of the alphabet σ, let us define *a complete set-trie* to be a set-trie corresponding to the whole power-set $\mathcal{P}([0, \sigma - 1])$ and denote the corresponding tree with $\mathcal{T}_\sigma$. Clearly, any set-trie with the alphabet-size σ will correspond to some subtree of $\mathcal{T}_\sigma$. Note that the tree $\mathcal{T}_\sigma$ is a binomial tree. We now give some basic properties of the complete set-tries.

Observation 1. *Let $\mathcal{T}_\sigma$ be a complete set-trie on alphabet of size σ. Then the following holds:*

1. *The tree $\mathcal{T}_\sigma$ contains 2^σ vertices.*
2. *All σ neighbors of the root induce subtrees that are isomorphic to $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\sigma-1}$.*
3. *There is precisely 2^{i-1} nodes with label i, for $1 \leq i \leq N$.*

For better illustration of a complete set-trie structure, see Figure 2 depicting the rooted tree on that corresponds to a maximal set-trie, with $\sigma = 4$.

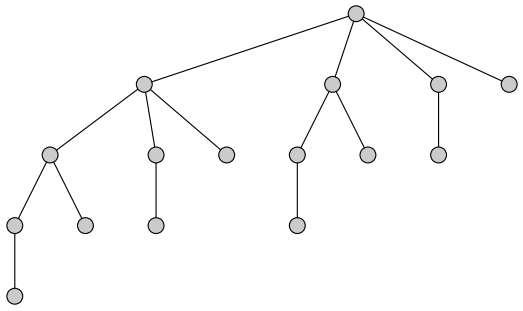

Figure 2: A complete trie for $\sigma = 4$.

3. Mathematical analysis of running-time

In this section we try to provide a mathematical analysis of the running time of the described algorithm. In Section 3.1, we start by describing the basic notations and initial assumptions of our model on the type and distribution of the set-trie, as well as for the input. In particular, our analysis is based on a strong assumption that all words are present in our collection of sets with uniform probability.

The core building block which plays important role in the expected performance of all four functions is the initial cardinality of our set-trie, which we describe in Section 3.1.1. In the process, we use the idea of Galton-Watson branching stochastic process, where we grow our population by consecutively attaching letters of our alphabet to our set-trie. We upper-bound the growth function for any member of the alphabet in Lemma 2, and later construct a probability generating function that majorizes the probability distribution of the cardinality of our set-trie. Finally, we measure the expectation of this probability generating function by the standard derivation approach an obtain the desired expected cardinality in Proposition 5.

In Section 3.2, we observe the exponential behavior of the GETALLSUBSETS function, while determining an additional upper-bound for the EXISTSUBSET function. The result of Section 3.1.1 is not only an answer for GETALLSUBSETS (with different parameters, see Section 3.2.1), but is also an important upper-bound for the

later analysis of EXISTSUBSET. In particular, we determine that in our model, the distribution of the running time of EXISTSUBSET is a majorization of a geometric distribution with parameter p.

In the Section 3.3, we observe that, for method GETALLSUPERSETS, input sets with low-ranked members perform faster then these with high-ranked ones. The nature of the algorithm shows that the length of an input may be as important as the label of its last character - hence the problem is translated to the bound from Section 3.1.1 (again, with different parameter, see Section 3.3). For the function EXISTSUPERSET we observe that, as long as p and α_k are not very small and fairly large, respectively, the expected number of visited nodes will not be large. The bound is attained by carefully constructing an appropriate Bernoulli variables which allow us to construct geometric distribution. Such geometric distribution will again minorize the actual running-time distribution of EXISTSUPERSET. Since this last analysis may be a bit abstract to get a proper performance feeling, we conclude the section by a short discussion on the results.

3.1. The description of the model

Let Σ be the alphabet of size σ and let W be a collection of sets from 2^{Σ} (i.e. subsets of of Σ) stored in the corresponding set-trie S. By the implementation of the set-trie data structure, note that S can be interpreted as a subtree of $\mathcal{T}_\sigma$, where each node contains a non-unique label $l \in \Sigma$. In this sense, observe that $|V(S)| \geq |W|$, as for a given $w \in W$, the set-trie S also includes all prefixes of w. For a fixed probability $p \in (0,1)$ and for $q = 1-p$, our model assumes that for each set $w \subseteq \Sigma$ we have $P[w \in W] = p$. Note that the expected number of sets stored in W equals to $p \cdot 2^\sigma$. For simplicity, we assume that $\Sigma = \{1, 2, \ldots, \sigma\}$. For any subset $Y \subseteq \Sigma$ we denote $S[Y]$ the corresponding subgraph of S, induced by all i-labeled vertices, where $i \in Y$.

By Observation 1, there are 2^{i-1} i-labeled vertices in a complete set-trie $\mathcal{T}_\sigma$, but not all will be present in S. For any particular i-labeled node v from $\mathcal{T}_\sigma$, let $p_{i,\sigma}$ be the probability that $v \in S$, where S is generated by our model, on parameters σ and p. Observe that v is a root of a copy of $\mathcal{T}_{\sigma-i}$, hence $v \in S$ if and only if any word corresponding to members of its subtree is also member of W. It is hence clear that

$$p_{i,\sigma} \quad = \quad 1 - \prod_{v \in \mathcal{T}_{\sigma-i}} (1-p) = 1 - q^{2^{\sigma-i}}.$$

Let $X = \alpha_1\alpha_2\cdots\alpha_k$ be a user specified input set consisted of k members of Σ. Our goal is to estimate an average time complexity for method EXISTSSUPERSET(S, X), GETALLSUPERSETS(S, X), EXISTSSUBSET(S, X) and GETALLSUBSETS(S, X). In order to get a proper insight of the model used, observe the following estimation of the expected number of nodes in S.

3.1.1. The size of $|S|$.

As we observed, it is easy to estimate the cardinality of W. In the sequel, we try to upper-bound the value of $|S|$. The result, as well as the calculation, will help us later in complexity analysis of mentioned methods. We start by calculating the conditional probability that, if $x_i \in S$, then x_i contains a neighbor labeled $i+1$, which is denoted by x_{i+1}.

Lemma 2. *Let $i < j$ be positive integers and let x_i be an arbitrary but fixed i-labeled vertex from S. The probability that it contains a j-labeled neighbor x_j under the assumption that $x_i \in S$ can be upper-bounded to*

$$P[x_j x_i \in E(S) | x_i \in S] \leq \left(1 + q^{2^{\sigma-j}}\right)^{-1},$$

with equality if and only if $j = i+1$.

Proof. By definition of conditional probability, we have

$$\begin{aligned} P[x_j \in S | x_i \in S] \quad &= \quad \frac{P[x_j \in S \wedge x_i \in S]}{P[x_i \in S]} = \frac{P[x_j \in S]}{1 - q^{2^{\sigma-i}}} = \frac{1 - q^{2^{\sigma-i}} + q^{2^{\sigma-i}} - q^{2^{\sigma-j}}}{1 - q^{2^{\sigma-i}}} \\ &= \quad 1 - q^{2^{\sigma-j}} \frac{1 - q^{2^{\sigma-j}(2^{j-i}-1)}}{1 - q^{2^{\sigma-i}}} \leq 1 - q^{2^{\sigma-j}} P[x_j \in S | x_i \in S]. \end{aligned}$$

By isolating the term $P[x_j \in S | x_i \in S]$ from expression above, the claim follows. □

For each character $i \in \Sigma$ we now measure the size of a subtree of S, induced on all instances of vertices, with labels smaller or equal to i. In particular, let f_i be a probability generating function for the corresponding distribution of the subtree size, i.e.

$$f_i(x) = \sum_{j=0}^{2^i} P\left[|S\left[1,\ldots,i\right]| = j\right] \cdot x^j.$$

Let us state the standard definition of majorizing sequence.

Definition 3. *Let $A = a_1, a_2, \ldots, a_d$ and $B = b_1, b_2, \ldots, b_d$ be two number sequences, such that $\sum_{i=1}^{d} a_i = \sum_{i=1}^{d} b_i$. We say that A* majorizes *B, if $\sum_{i=k}^{d} a_i \geq \sum_{i=k}^{d} b_i$, for each $k \in [1, d]$.*

We now define a generating function $g_i(x) = \sum_{j=0}^{2^i} a_{i,j} x^j$, with $g_0(x) = x$ which is recursively defined as $g_{i+1}\left(x\right) = g_i\left(p_i(x)\right)$, where $p_i(x) = \frac{x^2 + x\, q^{2^{\sigma-i}}}{1+q^{2^{\sigma-i}}}$. Function $g_i(x)$ is conveniently defined, as its coefficients majorizes these from $f_i(x)$, as observed in the following claim.

Lemma 4. *For each $i \in [1, \sigma]$ and $k \in \left[0, 2^i\right]$, we have*

$$\sum_{j=k}^{2^i} a_{i,j} \geq P\left[|S\left[1,\ldots,i\right]| \geq k\right],$$

i.e. the coefficients form g_i majorize these from f_i.

Proof. We prove the claim by induction on i. For $i = 1$, we have $g_1(x) = \frac{x^2 + x \cdot q^{2^\sigma}}{1+q^{2^\sigma}}$, while $f_1(x) = px^2 + qx$. Now suppose that the coefficients form g_i majorize these from f_i, and observe that a probability generating function $g_{i+1} = g_i\left(p_i(x)\right)$ represents some distribution obtained by a modified Galton-Watson process, with dynamic growth function $p_i(x) = \frac{x^2 + x\, q^{2^{\sigma-i}}}{1+q^{2^{\sigma-i}}}$. From Claim 2 it is clear, that the mentioned growth function actually upperbounds the actual growth of coefficients of f_{i+1}, hence the conclusion. □

Proposition 5. *The expected size of S is at most $\prod_{i=0}^{\sigma-1} \left\lceil \frac{2+q^{2^{\sigma-i-1}}}{1+q^{2^{\sigma-i-1}}} \right\rceil$.*

Proof. From Lemma 4 it is clear that for any i, the distribution given by probability generating function f_i majorizes the one given by g_i. Note that for any probability generating function $\mathcal{F}(x)$, it is well known that the expectation of the corresponding distribution equals to $\frac{\partial}{\partial x}\mathcal{F}'\left(x\right)|_{x=1}$[1]. As pointed out in the Lemma 4, note that functions $\{g_i\}_{i\geq 0}$ are generated in a way similar to the Galton-Watson process, apart from the fact that in our case the reproduction function is not constant throughout the process, but changes at every step, and is given by $p_i = \frac{x^2 + x\, q^{2^{\sigma-i}}}{1+q^{2^{\sigma-i}}}$ for i-th generation. Furthermore, g_i may be expressed as

$$g_i\left(x\right) = g_{i-1}\left(p_{i-1}(x)\right) = p_0 \circ p_1 \circ \cdots \circ p_{i-1}\left(x\right),$$

while by the composition rule, its derivation is

$$g_i'\left(x\right) = \prod_{j=0}^{i-1} p_j' \circ \cdots \circ p_{i-1}\left(x\right).$$

[1]For detailed survey on probability generating function we refer the reader to the book by Johnson et. al. [9].

The proof of the claim is concluded by

$$\begin{aligned}\mathbb{E}\left(|S|\right) = f'_{\sigma}\left(x\right)|_{x=1} &< g'_{\sigma}(x)|_{x=1} = \prod_{j=0}^{i-1} p'_j \circ \cdots \circ p_{i-1}\left(x\right)|_{x=1} \\ &= \prod_{i=0}^{\sigma-1} p'_i(x)|_{x=1} = \prod_{i=0}^{\sigma-1}\left[\frac{2+q^{2^{\sigma-i-1}}}{1+q^{2^{\sigma-i-1}}}\right],\end{aligned}$$

where the second line follows by the fact that $p_i(1) = 1$ and $p'_i(x)|_{x=1} = \frac{2+q^{2^{\sigma-i-1}}}{1+q^{2^{\sigma-i-1}}}$, for any admissible i. □

3.2. Analysis of subsets

In this section we analyze the method regarding the subset queries. For GETALLSUBSETS(S, X), it is clear that the length of output is exponential in k, hence we expect the obtained upper-bound to also be exponential.

In the case of EXISTSSUBSET(S, X), the situation turns out to be more complex. Our upper-bound consists of two functions; one being useful for big densities of our set-trie, and the second one for sparse cases.

3.2.1. Analysis of GETALLSUBSETS(S, X).

When traversing the set-trie S, the algorithm visits only nodes which labels appear in X. These nodes represent a subtree of depth at most k in S. We denote this tree with $S\left[X\right]$, and let w_X be its size. In light of Proposition 5, it is easy to give an upper estimate of $\mathbb{E}\left(w_X\right)$ in the same way as we did with S, hence

$$\mathbb{E}\left(w_X\right) \leq \prod_{i\in X}\left[\frac{2+q^{2^{\sigma-i-1}}}{1+q^{2^{\sigma-i-1}}}\right].$$

As expected, since we do not assume that the input word X is chosen uniformly at random, the result shows that the expected running time in our model is highly input-sensitive. Observe that knowing the relative frequency of characters in given input would allow us to choose the initial total ordering of the alphabet Σ in a correct way. In particular, the least frequent characters should be are ranked higher than others. Indeed, observe that in real-world applications, X may be efficiently modeled by various randomized generations, such that given a character $c \in X$ one may assume any of the following:

- c is chosen uniformly at random from $[1, \sigma]$,
- $c \in [1, \log\sigma]$, with high probability,
- $c \in [\sigma - \log\sigma, \sigma]$, with high probability.

The trivial upper bound of w_X is 2^k, which is attained at $p = 1$. Also, the reader should observe that the obtained bound is exponential in k, i.e. for any input word X there exist a constant $\alpha \in [1.5, 2]$, such that $\mathbb{E}\left(w_X\right) = \alpha^k$. Finally, the reader should realize that for most of randomized models of X (including these mentioned above), it is easy to see that $\alpha \to 2$ when $\sigma \to \infty$.

3.2.2. Analysis of EXISTSSUBSET(S, X).

In contrast to GETALLSUBSETS(S, X), when traversing the tree $S\left[X\right]$, the method EXISTSSUBSET(S, X) stops at the first found flagged node. At any of vertices in $S\left[X\right]$, the algorithm stops with probability p or continues otherwise. First note that in the case when the result of EXISTSSUBSET(S, X) is `False`, then the complexity of EXISTSSUBSET(S, X) in fact equals the complexity of GETALLSUBSET(S, X), analyzed in Section 3.2.1, and hence takes precisely w_X time steps. However, if this is not the case, we point out that the number of steps is distributed according to a geometric probability distribution on parameter p.

To be more precise, the probability of stopping at i-th visited node is $p \cdot (1-p)^{i-1}$, for $i \leq w_X$. After searching through all possible sub-sets (i.e. after w_X steps), algorithm stops – probability of this scenario is

clearly $(1-p)^{w_X}$. In particular, let $\mathcal{X}$ be a random variable that measures the number of steps made when running EXISTSSUBSET(S, X). The value of $P[\mathcal{X} = t]$ is defined as follows:

$$P[\mathcal{X} = t] = \begin{cases} p \cdot (1-p)^{t-1} & \text{if } 1 \leq t < w_X; \\ p \cdot (1-p)^{w_X - 1} + (1-p)^{w_X} & \text{if } t = w_X. \end{cases}$$

It is easy to check that this is a proper probability space that sums to 1, and that the probability space is very similar to a geometric distribution with parameter p (where the end of right tail is cut off at w_X). While w_X is still the upper-bound of $\mathcal{T}$ (EXISTSSUBSET(S, X)), calculating an expectation of $\mathcal{X}$ gives us another upper-bound.

Theorem 6. The expected number of steps of EXISTSSUBSET(S, X) is equal to:

$$\frac{1}{p} \cdot \left(1 - (1-p)^{w_X}\right).$$

In particular, the expected number of steps is at most $\frac{1}{p}$.

Proof. We calculate the expectation of the random variable $\mathcal{X}$,

$$\mathbb{E}(\mathcal{X}) = \sum_i i \cdot P[\mathcal{X} = i] = w_X (1-p)^{w_X} + p \cdot \sum_{i=0}^{w_X} i \cdot (1-p)^{i-1}. \tag{1}$$

Setting $q := 1 - p$, we get a closed form by integrating

$$\int \left(\sum_{i=0}^{w_X} i \cdot q^{i-1} \right) dq = \frac{1 - q^{w_X+1}}{1-q} + C.$$

Plugging this back into (1), we get

$$\begin{aligned} \mathbb{E}(\mathcal{X}) &= w_X q^{w_X} + (1-q) \cdot \frac{d}{dq} \left(\frac{1 - q^{w_X+1}}{1-q} + C \right) \\ &= w_X q^{w_X} + \frac{w_X q^{w_X+1} - (w_X+1) q^{w_X} + 1}{1-q} \\ &= w_X q^{w_X} + \frac{1 - q^{w_X} (w_X - w_X q + 1)}{p} = w_X q^{w_X} \left(1 - \frac{1-q}{p}\right) \\ &= \frac{1}{p} \left(1 - (1-p)^{w_X}\right) \end{aligned} \tag{2}$$

□

It is easy to see that the bound above is quite efficient in some cases - here we point out the two efficient situations. In the Example 7 we have a large alphabet comparing to the length of an input.

Example 7. *Assume that our input is very short, i.e. $k = o(\log \sigma)$. Then we have*

$$\mathbb{E}(\mathcal{X}) \leq w_X \leq 2^k \leq o(\sigma).$$

In the Example (ii) we have a dense set-trie comparing to the length of an alphabet.

Example 8. *Suppose that our set-trie is quite dense, i.e. if $p > \epsilon \cdot \left(\frac{1}{\sigma}\right)$, for arbitrary small constant $\epsilon > 0$. Then clearly $\mathbb{E}(\mathcal{X}) \leq \frac{1}{p} < O(\sigma)$. We believe that many of the real-world datasets are* dense *with respect to the criteria above. For example, an actual set of all words in English dictionary consists of approximately a million words, where a standard 26-letter English alphabet is used. Setting the appropriate values with*

$\sigma = 26$, $p = 0.0148$ *and* $\epsilon = 0.385$*, we expect the number of sets visited by* EXISTSSUBSET(S, X) *to be bounded around* 68 *or less, or even less when*$|X|$ *is small.*[2]

While the analysis above mostly focuses on the expectation of the running time, observe that the distribution of $\mathcal{X}$ is well-concentrated around its expectation - for instance, even Markov inequality implies w.h.p. that $\mathcal{X}$ does not exceed its expectation by a factor of $\log \sigma$, i.e.

$$P\left[\mathcal{X} > \mathbb{E}\left(\mathcal{X}\right) \log \sigma\right] \leq \frac{1}{\log \sigma} \underset{w.h.p.}{\longrightarrow} 0.$$

Since in our model all sets are chosen with uniform probability p - resembling a geometric distribution, the type of the traversal algorithm does not play any role in this theoretical model. However, in real-world situations we suspect this is not the case – it may be useful to consider various tree-traversal strategies depending on a letter-frequency which is usually far from uniform.

3.3. Analysis of supersets

Our goal is to estimate an average time complexity for method GETALLSUBSETS(S, X), and EXISTSSUPERSET(S, X). Similarly as with subset-queries, since the length of output for GETALLSUBSETS(S, X) is exponential in α_k, we expect the obtained upper-bound to be exponential. For the function EXISTSUBSET(S, X) we observe that, as long as p and α_k are not very small and fairly large, respectively, the expected number of visited nodes will not be large.

3.3.1. Analysis of GETALLSUPERSETS(S, X).

When traversing the set-trie S, algorithm visits the nodes in a depth-search manner, and visits only the branches that do not miss any member of X. Let the random variable $\mathcal{X}$ represent the upper bound of number of steps made when running the method existsSuperset(S, X). Since algorithm can only stop at α_k-vertex, the full-tree of size 2^{α_k} is a trivial upper-bound of number of GETALLSUPERSETS(S, X). Using the result from Proposition 5, we can obtain the upper-bound of the expected cardinality of the set-trie on alphabet of size α_k, which can be reduced to

$$\mathbb{E}\left(|\mathcal{X}|\right) < \prod_{i=0}^{\alpha_k - 1} \left[\frac{2 + q^{2^{\sigma - i}}}{1 + q^{2^{\sigma - i}}}\right].$$

The expression is clearly exponential in value of α_k, i.e. the *rank* of the maximal element in $\mathcal{X}$. Note that this bound is far from tight, as we do not consider the aspect of "branch pruning" in GETALLSUPERSETS and assume that the algorithm visits all the vertices in $S\left[0, \ldots, \alpha_k - 1\right]$. Nontheless, in the Experiment 3 we will observe that input sets with low-ranked members perform faster then these with high-ranked ones.

3.3.2. Analysis of EXISTSSUPERSET(S, X).

Using Bernoulli distribution, we now estimate the number of needed steps for EXISTSSUPERSET(S, X). Although many of them will not be present, there is at most $2^{\alpha_k - k}$ possible positions of α_k-node in S, and at the encounter of any α_k-node, algorithm stops immediately. If algorithm failed to reach some potential α_k-node in S, it need to visit at most α_k additional nodes to come to the next one (or determine that it does not exist).

At any instance of α_k-vertex, the algorithm stops with probability $p' = \left(1 - (1-p)^{2^{\sigma - \alpha_k}}\right)$ or continues otherwise. If the algorithm does not stop after trying to visit all possible instances of α_k-vertices, it returns `False` with probability $(1 - p')^{2^{\alpha_k - k}} = (1-p)^{2^{\sigma - k}}$ after making at most $2^{\alpha_k - k} \cdot k$ steps. Let $\mathcal{X}$ be a random variable with the following distribution:

$$P\left[\mathcal{X} = ki\right] = \begin{cases} p' \cdot (1 - p')^{i-1} & \text{if } 1 \leq i < 2^{\alpha_k - k}; \\ (1 - p')^{2^{\alpha_k - k} - 1} & \text{if } i = 2^{\alpha_k - k}. \end{cases}$$

[2] In this dataset, the result could be further improved by the fact that not all words appear in English dictionary with the same probability; hence a depth-first search approach may improve the results.

α_k	12	6	17	24	5	14	24
p	10^{-4}	10^{-6}	0.03	0.08	0.0015	0.0015	0.0015
p'	0.559	0.408	0.999	0.154	1.000	0.954	0.003
$\frac{\alpha_k}{p'}$	21	15	17	156	5	15	8006

Table 1: The correlation between p, p' and $\frac{\alpha_k}{p'}$, with $\Sigma = \{0, \ldots, 24\}$ and various values of p and α_k.

It is easy to check that this is a proper probability space that sums to 1. Indeed,

$$p' \cdot \sum_{i=0}^{2^{\alpha_k - k}-1} (1-p')^i + (1-p)^{2^{\alpha_k - k}} \quad = \quad p' \cdot \frac{1-(1-p')^{2^{\alpha_k-k}}}{1-(1-p')} + (1-p')^{2^{\alpha_k-k}} = 1.$$

Theorem 9. Let $\mathcal{A}$ be the expected number of visited nodes in the algorithm EXISTSSUPERSET(S, X), where S is a set-trie that corresponds to $W \subseteq 2^\Sigma$, and $X \in \binom{\Sigma}{k}$. Then $\mathcal{A} \leq \mathcal{X}$. In particular,

$$\mathcal{A} < \alpha_k 2^{\alpha_k - k}.$$

Proof. Let $w' = 2^{\alpha_k - k}$, let $p', \mathcal{X}$ be as defined above, and let $q' = 1 - p'$. Again observe, that

$$\int \sum_{i=0}^{w_X} i \cdot q'^{i-1} dq' = \frac{1 - q'^{w_X+1}}{1-q'} + C.$$

The calculations of the expectation of random variable $\mathcal{X}'$ below are similar than in previous section:

$$\begin{aligned}
\mathbb{E}(\mathcal{X}') &= \sum_i i \cdot P[\mathcal{X}' = i] \\
&= \alpha_k \cdot w' (1-p')^{w'} + p'\alpha_k \cdot \sum_{i=0}^{w'} i \cdot (1-p')^{i-1} \\
&= \alpha_k \cdot w' (1-p')^{w'} + p'\alpha_k \cdot \frac{\partial}{\partial(1-p')} \left(\frac{1-(1-p')^{w'+1}}{1-(1-p')} + C \right) \\
&= \frac{\alpha_k}{p'} \left(1 - (1-p')^{w'}\right).
\end{aligned}$$

□

Again, notice that $\left(1-(1-p')^{w'}\right)$ clearly lies in an $[0, 1]$. The obtained bound is useful whenever p' is not very small, but the reader should notice that, while both p and p' are members of $[0, 1]$, it may often be the case that $p' \gg p$. For more intuition on the correlation between p, p' and $\frac{\alpha_k}{p'}$, and for easier correlation with Experiment 3, (see Figure 4), see some sample values on Table 1. One may observe that, as long as p and α_k are not very small and fairly large, respectively, the expected number of visited nodes will not be large. If however that would be the case, the results may not be as promising. The behavior of the time efficiency of the algorithm with various values of p can be observed in Experiment 1, while the relation with the value α_k is discussed in Experiment 3.

4. Experiments

This section presents the empirical study of the data structure set-trie. The data structure set-trie is implemented in the programming language C in form of a library [16]. An initial implementation of the set-trie was

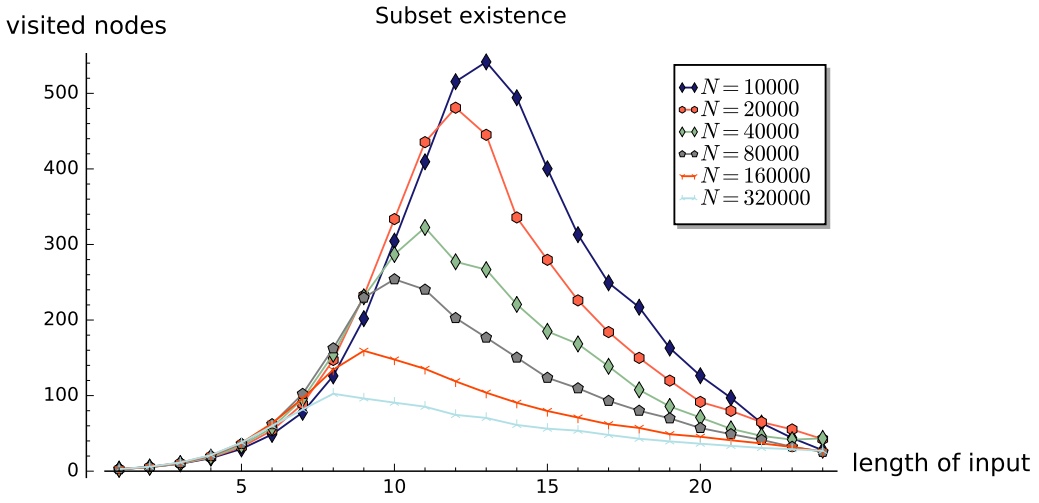

Figure 3: Experiment 1: Performance of EXISTSSUBSET(S,X).

in the frame of the system for the induction of functional dependencies from relations `fdep` [17], also implemented in C, and, later, as a part of the program for the induction multi-valued dependencies implemented in Sictus Prolog [18].

The experiments were performed on randomly generated sets of sets. In the experiments we are analyzing the influence of the selected parameters to the performance of the data structure set-trie. The performance of set-trie is always measured by the number of nodes visited by the particular operation that we observe.

Let us present the basic terminology to be used for the description of the experiments. The alphabet used for the experiments is $\Sigma = \{1, 2, \ldots, \sigma\}$, where σ is the size of the alphabet. Each operation is tested by using the sets X from a test set of sets T of size M. The number of sets in a set-trie S is denoted by N. By the term *density* of the set-trie, we refer to the value of $N/2^\sigma$.

Experiment 1 presents the influence of changing the size of a set of sets W and the corresponding set-trie S to the performance of set containment operations. The influence of the size of the alphabet Σ on the performance of the set containment operations is studied in Experiment 2. Finally, Experiment 3 presents the influence of the shape of the sets $X \in T$, i.e., the selection of the elements from Σ in the test set of sets T, to the performance of set-trie. Let us now give the description of the generating procedure for the construction of the test set-tries in the following section, and, after this, present the experiments in the subsequent sections.

4.1. The description of generating procedure and related notions

We construct the set-tries S from the collection of N subsets of Σ, where each subset is selected from $\mathcal{P}(\Sigma)$ uniformly at random, each by a given probability p. In other words, we traverse through all subsets of Σ and add each set to S with probability p.

The test set T of size M is generated in the similar way to the generation of the set-trie S. However, we construct T in a way that all lengths of the sets from T have approximately the same number of instances. Note that the parameters p and M have direct influence on the cardinality of a given set-trie or test set, respectively.

4.2. Experiment 1.

In the first experiment we observe the influence of the size N of the set-trie S to the number of nodes visited in S by the operations EXISTSSUBSET, EXISTSSUPERSET, GETALLSUBSETS, and GETALLSUPERSETS.

The size of the alphabet Σ is fixed to 25 so there can be at most $2^{25} = 33554432$ sets constructed from Σ. The number of sets N in the set-tries S are: 10000, 20000, 40000, 80000, 160000 and 320000. Each operation is tested with the sets X from the test set T of the fixed size 50000.

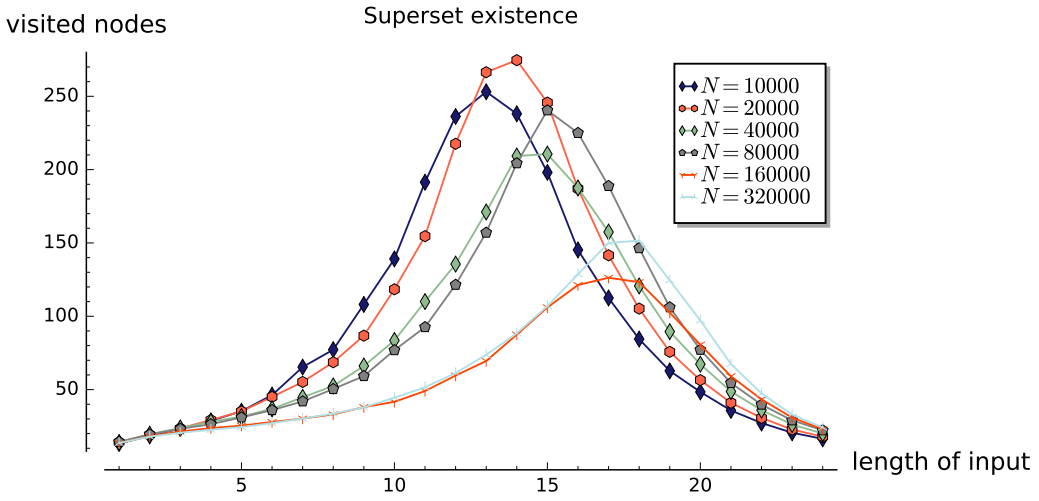

Figure 4: Experiment 1: Performance of EXISTSSUPERSET(S,X).

The results of Experiment 1 for the operation EXISTSSUBSET(S, X) are presented in Figure 3. When the size of X is small, algorithm visits few nodes to decide if a subset of X exists in S. The reason for this is in the small search space—only the elements of X need to be checked. Similarly, when the size of X is close to 25, only from 30 to 120 nodes are visited. In this case, most of the elements of Σ are included in X, and, therefore, some subset of X in S can be found quickly.

The maximums of functions for the operation EXISTSSUBSET are around the sets of size 9-14. The subset of X in S may not be covered by the first path generated by the algorithm—the depth-first algorithm of EXISTSSUBSET may backtrack few times until the subset is found. The maximums for the set-tries S with the smaller number of sets, including from 10000 to 20000 sets, are around the sets of the size 13-14. The maximum is around the sets of size 9-10 for the larger set-tries storing from 160000 to 320000 sets. The shift of the peek to the left of the center is the consequence of the increased size of S, as it will be presented in more detail in the sequel.

The algorithm of EXISTSSUPERSET is, in a way, opposite to the algorithm of EXISTSSUBSET. While the algorithm of EXISTSSUBSET searches among the elements of X, the algorithm of EXISTSSUPERSET must include all the elements of X and searches among the elements that are not in X. The set X in algorithm of EXISTSSUPERSET therefore acts as a constraint—the path in S that includes the superset of X must include all the elements from X.

The results of the operation EXISTSSUPERSET(S,X) are presented in Figure 4. The number of visited nodes is significantly smaller for the operation EXISTSSUPERSET in comparison to the number of visited nodes of the operation EXISTSSUBSET. However, the number of visited nodes is changing with regards to the number of sets in S. After some initial threshold in the number of sets in S, the performance of EXISTSSUBSET becomes better than the performance of EXISTSSUPERSET. This can be observed in Figures 3 and 4 where the performance of EXISTSSUPERSET and EXISTSSUBSETS flips while changing $|S|$ from 80000 to 320000. The flip is obvious from the results of Experiment 2.

The peeks of the functions from Figure 4 are, in comparison to the results of EXISTSSUBSET in Figure 3, moved slightly to the right of the center point $\sigma/2$—the peeks move further to the right with the increasing size of S. The search for supersets is harder when the size of X is slightly larger than $\sigma/2$. The operation EXISTSSUPERSET can find supersets more easily when the size of S increases towards σ.

The figures of EXISTSSUBSET and EXISTSSUPERSET can be seen as mirrored across the line defined by $\sigma/2$. The explanation of this is the duality of the set operations subset and superset, as they define dual partial orderings of subsets of the alphabet Σ. This phenomena is further studied in Experiment 3 where the duality of

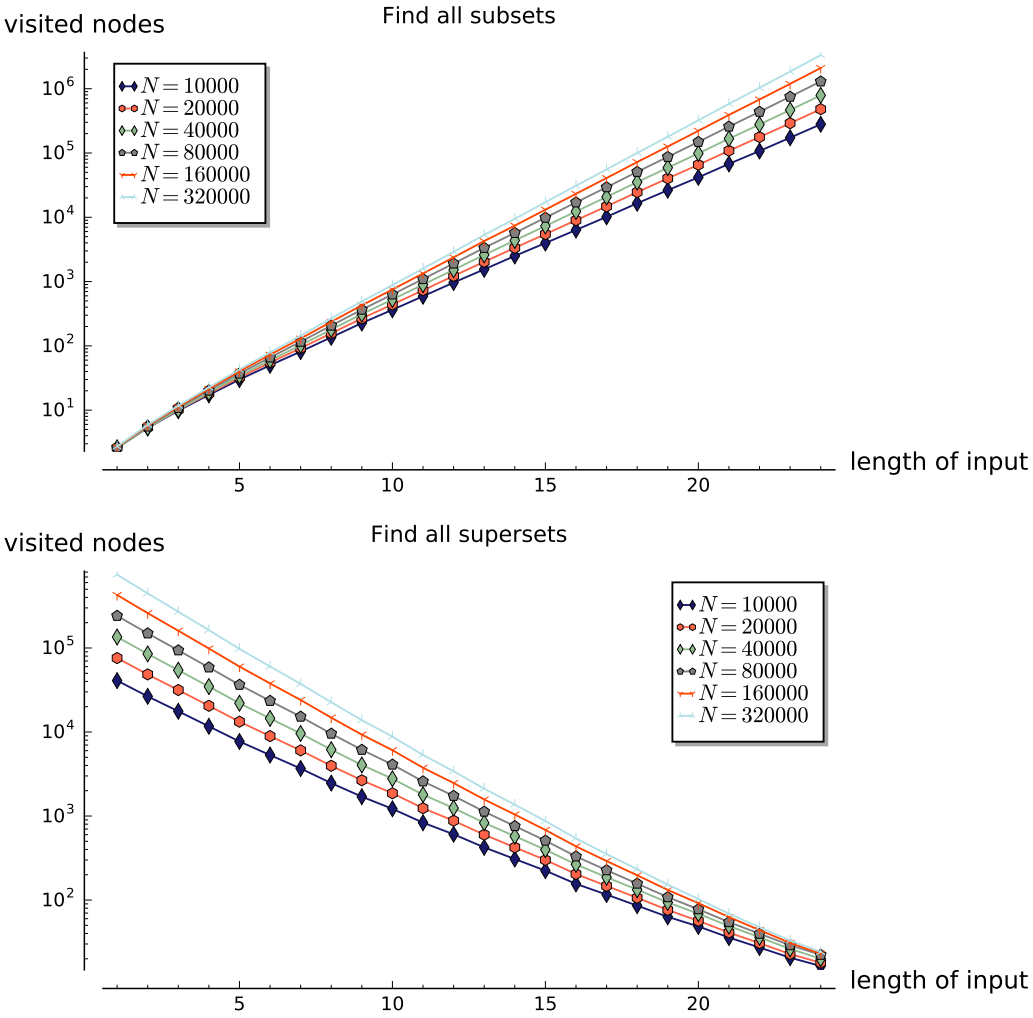

Figure 5: Experiment 1: Performance of GETALLSUBSETS(S,X) and GETALLSUPERSETS(S,X).

subset/superset operations is very obvious.

Figure 5 shows the performance of operations GETALLSUBSETS and GETALLSUPERSETS. The functions in Figure 5 are represented in logarithmic scale. All functions are either growing or decaying linearly. This means that the number of visited nodes is either growing exponentially in the case of operation GETALLSUBSETS, or, decaying exponentially in the case of operation GETALLSUPERSETS. The behavior of the functions can be easily explained by seeing that the number of subsets in S grows exponentially with the size of X, and, the number of supersets in S decays exponentially with the growing size of X.

4.3. Experiment 2.

In the second experiment we study the influence of the size σ of the alphabet Σ to the performance of set-tries. The sizes σ used in experiments are 14, 17, 20, 23 and 26. For each particular case, the set of sets W is constructed including 1.5% of all the subsets of Σ, for instance, in the case of $\sigma = 24$ we have approximately 250000 sets. The constructed sets of sets W are the inputs for the construction of the set-tries S. Furthermore, the test sets of sets T are constructed for each Σ to include 50000 sets.

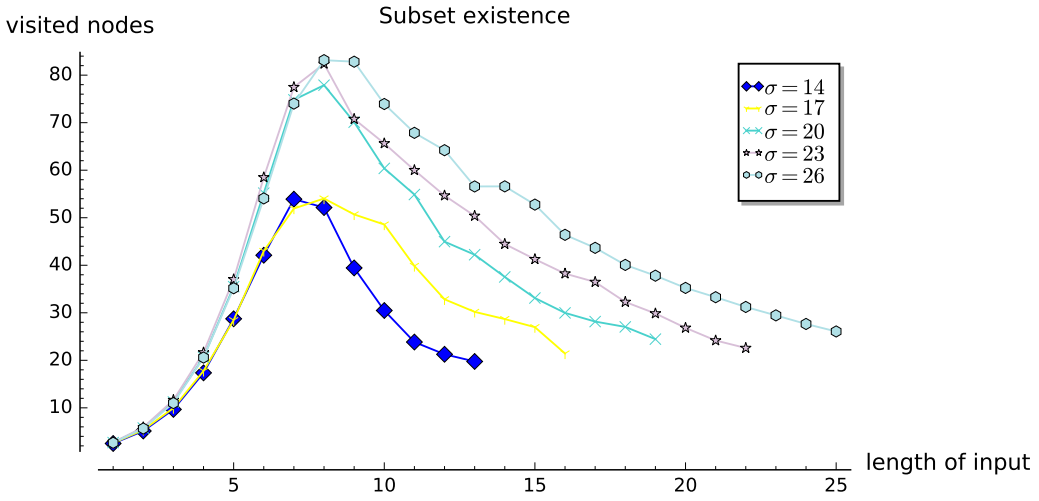

Figure 6: Experiment 2: Performance of EXISTSSUBSET(S,X).

First of all, let us note that because of the higher number of sets from S, the operation EXISTSSUBSET visits less number of tree nodes than the operation EXISTSSUPERSET. As we have described in the presentation of Experiment 1, the flip point, i.e., the point where the performances of subset/superset operations reverse, is approximately when there are $0.01 \cdot 2^{\sigma}$ (1% of all possible subsets of Σ) sets in S. Since in Experiment 2 we have 1.5% of all possible sets in all cases, the results confirm the findings of the Experiment 1.

The results of Experiment 2 for the operation EXISTSSUBSET(S, X) are presented in Figure 6. The maximal number of visited nodes for the alphabets of sizes 14, 17, 20, 23 and 26 are from 55 up to 85. The maximums of functions are around the elements with the indexes from 7 to 9, moving slightly to the right with the increasing size of the alphabet Σ. In the same way as in Figure 3 of Experiment 1, the peeks are actually moving slowly to the left of the central element with the increasing size of the alphabet Σ. However, the increase of the alphabet size causes that the maximum stays around the indexes from 7 to 9.

The results of the operation EXISTSSUPERSET(S,X) are presented in Figure 7. The operation EXISTSSUPERSET visits more tree nodes than the operation EXISTSSUBSET which is the consequence of the increase in the number of sets in S. The maximums of functions are always on the right-hand side of the middle elements of the alphabets Σ, and, they are moving further to the right with the increasing size of the alphabet Σ. The results are comparable to the results of Experiment 1 presented in Figure 4.

Finally, Figure 8 shows the performance of operations GETALLSUBSETS and GETALLSUPERSETS. Again, as in the case of Figure 5, the number of visited nodes of the operations GETALLSUBSETS and GETALLSUPERSETS either increase or decrease linearly in the logarithmic scale, which means that the number of visited nodes either increases or decreases exponentially with the size of set X. This reflects the number of sets that are the results of the operations GETALLSUBSETS and GETALLSUPERSETS.

4.4. Experiment 3.

This experiment is designed to provide more details about the behavior of operations when the input sets X include the elements from different ranges of the alphabet Σ. We show through the experiment that the operation SUBSET works faster if X has predominately the elements with higher indexes, and, works slower in the case X includes the elements with lower indexes. The operation SUPERSET behaves in the opposite way: operation visits only the small number of nodes if X includes predominately the elements with lower indexes, and, it visits larger number of nodes if X includes the elements with the higher indexes. The experiment clearly shows the duality of operations SUBSET and SUPERSET.

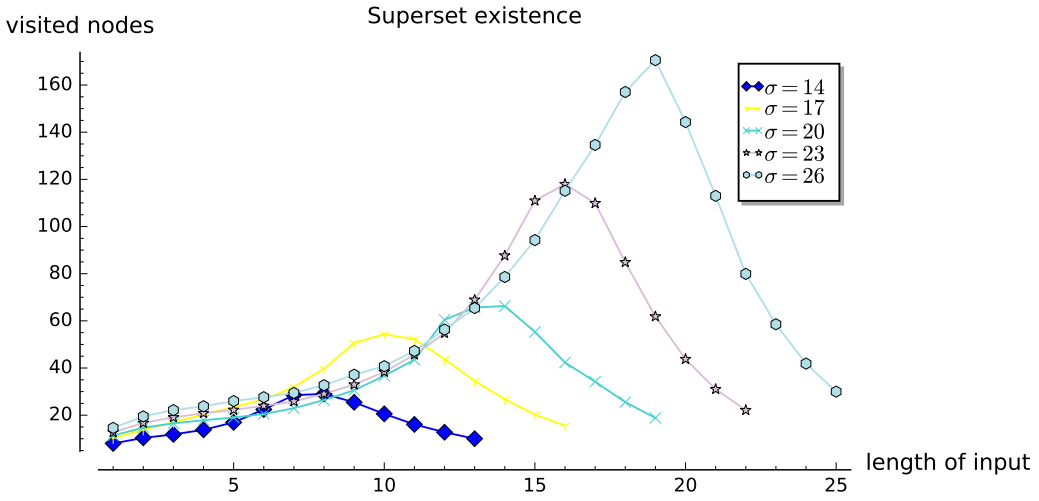

Figure 7: Experiment 2: Performance of EXISTSSUPERSET(S,X).

Let us now present the experiment in more detail. The settings of the experiment are as follows. The size of the alphabet Σ is 25. We use 4 test sets T as the input of set containment operations. The test sets are named FIRST, LAST, MIDDLE, and SPREAD. Each of the test sets includes exactly 25 sets.

Let k in this paragraph stand for an index from $[0, 24]$. The set FIRST includes the sets that for all k contain the indexes from the interval $[0, k]$. The set LAST includes 25 sets that contain the indexes from the interval $[k, 24]$. The set MIDDLE includes the sets that contain the elements around the center index 12. Each of them contains the elements from the range $[(12 - \lceil k/2 \rceil), (12 + \lfloor k/2 \rfloor)]$. Finally, the set SPREAD includes 25 sets, one for each of the lengths k. These sets include the indexes that are spread as much as possible in the given range $[0, 24]$. The examples of the sets of the size 7 for each of the above presented set of sets are as follows.

FIRST :{0,1,2,3,4,5,6}
LAST :{18,19,20,21,22,23,24}
MIDDLE :{9,10,11,12,13,14,15}
SPREAD :{0,4,8,12,16,20,24}

The sets FIRST, LAST, MIDDLE, and SPREAD are tested on 20 different set-tries. The set-tries were generated by using the uniform distribution of the set elements as presented in Section 4.1. Each set-trie stores 1.5% of all possible subsets of $[0, 24]$, i.e., approximately 500000 sets. The results for the set containment operations applied to the parameter sets FIRST, LAST, MIDDLE, and SPREAD, are presented as the four functions drawn in Figures 9-11. Note that each function value represent an average of 20 measurements.

Figure 9 shows the performance of operation EXISTSSUBSET. The maximal number of visited nodes for the input set of sets FIRST is for the sets of sizes from 12 to 16 where the number of visited nodes is from 250 to 370. This is more than it can be expected from the results presented in Figure 3 of Experiment 1. The reason for this is in the shape of the sets from FIRST that causes depth-first search algorithm of EXISTSSUBSET to search on the left-hand side of set-trie where the largest number of sets is stored. The sets X from FIRST do not include the elements with higher indexes, so in many cases X does not cover the complete set of S with its search path. Therefore, the algorithm of EXISTSSUBSET may search middle area from the leftmost edge towards the center of the set-trie to find a subset of X. After the size of X is bigger than 16 the algorithm finds a subset much easier in the left-deep part of set-trie since X includes more elements.

The results for the input set of sets LAST can be explained in the following way. When the sizes of sets from LAST are from 1 to 15 the search of EXISTSSUBSET is performed on the sub-trees from the right-hand side of the set-trie where the number of nodes in sub-trees fall exponentially with the increasing index of first element.

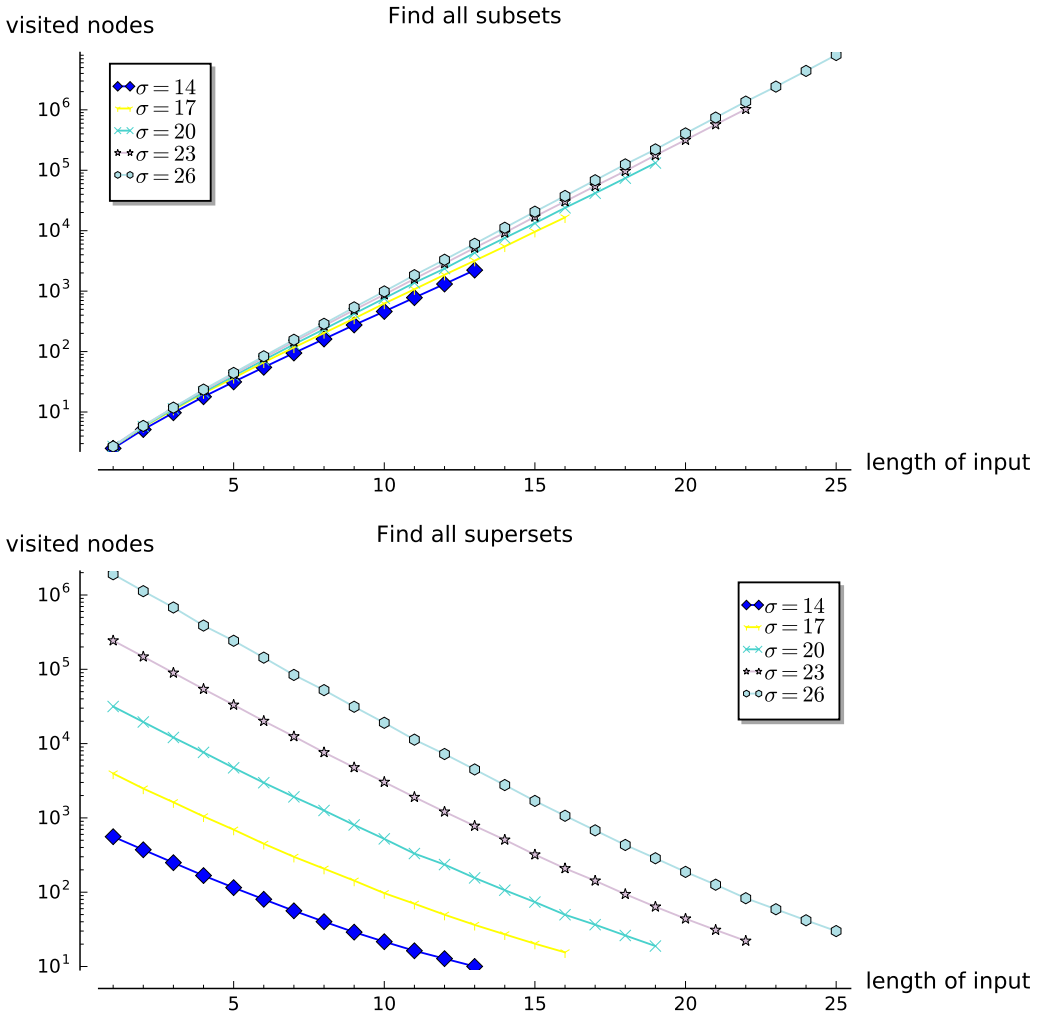

Figure 8: Experiment 2: Performance of GETALLSUBSETS(S,X) and GETALLSUPERSETS(S,X).

The algorithm of EXISTSSUBSET needs to visit smaller number of nodes in smaller sub-trees. After the size of X is larger than 15 it is much easier to find the subset of a larger set as in the case of FIRST.

The number of visited nodes by the algorithm of EXISTSSUBSET for the input set of sets MIDDLE is actually in the middle between the results obtained for FIRST and LAST. Since the test sets X include indexes centered around 12, the sub-trees searched by the algorithm are smaller than in the case of the set of sets FIRST. Therefore, operation EXISTSSUBSET visits fewer nodes when applied to MIDDLE than when applied to FIRST. Finally, since the shape of the sets from SPREAD are close to the random sets used in Experiment 1, the results for the input set of sets SPREAD are very close to the results of Experiment 1.

Let us now present the results of the operation EXISTSSUPERSET. The results are presented in Figure 10. Firstly, the results of the operation EXISTSSUPERSET when applied to FIRST and LAST are the opposite to the results of the operation EXISTSSUBSET. The algorithm of operation EXISTSSUPERSET visits more nodes when applied to LAST than when applied to FIRST. On the contrary, the operation EXISTSSUBSET visits more nodes when applied to FIRST than when applied to LAST.

Large number of nodes visited by the operation EXISTSSUPERSET applied to LAST is the consequence of the shape of the input sets X, and, the nature of depth-first search algorithm of EXISTSSUPESET. The input sets

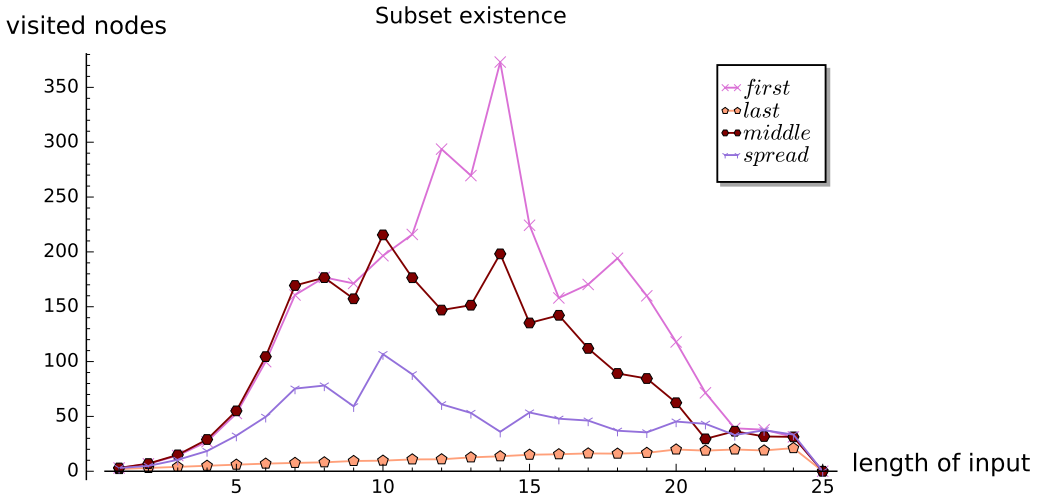

Figure 9: Experiment 3: Performance of EXISTSSUBSET(S,X).

X include in all instances the elements with the highest indexes $\{k, \ldots, 24\}$. To check the highest elements the algorithm has to climb to the branches of the tree, and, in the case that superset is not found it has to climb into the next branch, etc. Since the algorithm of EXISTSSUPERSET starts searching with the path including index 0 and then extends the path in the depth-first manner, it necessarily searches first the deepest part of the tree, progressing towards lower branches. Note that the path can include any element from $\{0, \ldots, k-1\}$. The performance of operation EXISTSSUPERSET improves very quickly when the indexes of the parameter sets X are moved down towards 0.

The results of the performance of EXISTSSUPERSET on the parameter sets MIDDLE, and SPREAD are actually as we could expect them from the results of Experiment 1. Indeed, the shape of input sets from MIDDLE, and SPREAD are quite similar to the randomly generated input sets that are used for Experiment 1.

The case of FIRST is actually a degenerated case. The algorithm searches for the path that includes all indexes in X but it may include some other indexes that are not in X. In the case that the indexes of the input set X are not far apart, the algorithm of EXISTSSUPERSET narrows the search while trying to include all elements from X. Since the sets from FIRST are of the form $\{0, 1, \ldots, k\}$ there is no need for searching, but the algorithm directly descends into some node, if the superset exists, or, can not find some element from X, if the superset does not exist.

The performance of operations GETALLSUBSETS and GETALLSUPERSETS is presented in Figure 11. The number of visited nodes grows exponentially with increasing the size of test sets X for the operation GETALLSUBSETS, and, with decreasing the size of test sets X for the operation GETALLSUPERSETS. However, the shapes of the functions are quite different to those in Figure 5 of Experiment 1—they reflect clearly the results of the operations EXISTSSUBSET and EXISTSSUPERSET shown in Figures 9 and 10. The functions obtained for the operation GETALLSUBSETS are the inverse of the functions obtained for the operation GETALLSUPERSETS. The functions of the test sets of sets FIRST and LAST represent the upper and the lower border for the operation GETALLSUBSETS, and, they represent the lower and the upper border for the operation GETALLSUPERSETS.

5. Related work

The problem of querying sets of sets appears in various areas of computer science. Firstly, the problem has been studied in the form of *substring search* by Rivest [14], Baeza-Yates [1] and Charikar [3]. Secondly, the subset queries are studied in various sub-areas of AI for storing and querying: pre-conditions of a large set of rules [6], states in planning for storing goal sets [8] and hypotheses in data mining algorithms [10]. Finally,

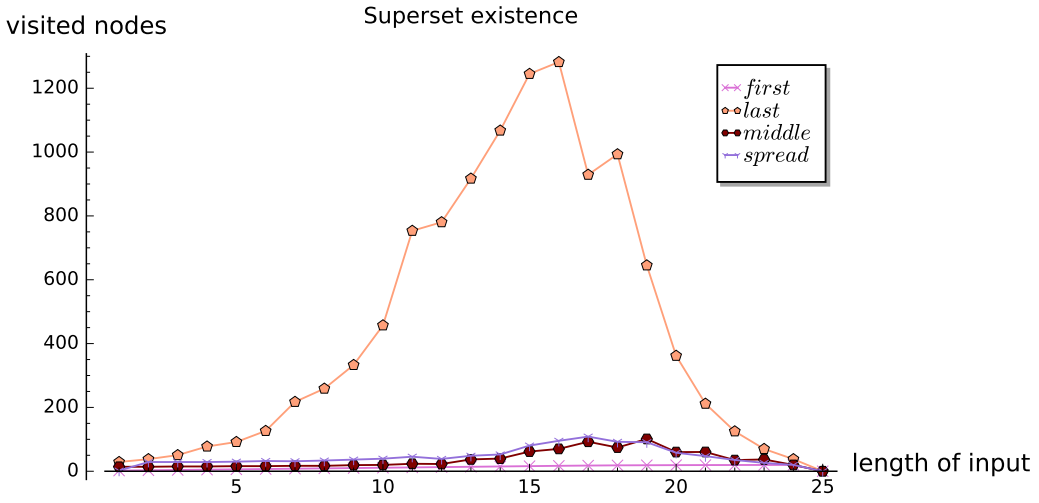

Figure 10: Experiment 3: Performance of EXISTSSUPERSET(S,X).

querying sets is an important problem in object-relational database management systems where attributes of relations can range over sets [21, 13, 7, 22, 23].

5.1. Partial-matching and containment query problem

The data structure we propose is similar to the data structure *trie* [14, 15]. Since we are not storing sequences but *sets* we can exploit the fact that the order in sets is not important. Therefore, we can take advantage of this to use the syntactical ordering of the set elements and obtain the additional functionality of tries.

Our problem is similar to searching substrings in strings for which the tries and th Suffix trees can be used. Firstly, Rivest examines [14] the problem of partial matching with the use of the hash functions and the trie trees. He presents an algorithm for the partial match queries that uses tries. However, he does not exploit the ordering of indices that can only be done in the case that the sets or multisets are stored in tries.

Baeza-Yates and Gonnet present an algorithm [1] for searching regular expressions using Patricia trees as the logical model for the index. They simulate a finite automata over a binary Particia tree of words. The result of a regular expression query is a superset or a subset of the search parameter.

Finally, Charikar et. al. [3] present two algorithms to deal with a subset query problem. The functionality of their algorithms is similar to the operation EXISTSSUPERSET. They extend their results to a more general problem of the orthogonal range searching, and some other problems. They propose a solution for "containment query problem" which is similar to our 3rd query problem introduced in Section 2.

5.2. Querying hypotheses and states in AI systems

The initial implementation of set-trie was in the context of a datamining tool FDEP [17] which is used for the induction of functional dependencies from the relations [19, 5]. It has been further used in datamining tool MDEP [18] for the induction of multivalued dependencies from the relations. In both cases, the sets are used as the basis for the representation of dependencies. Hypotheses (dependencies) are checked against the negative cover of *invalid dependencies* represented by means of the data structure set-trie. Furthermore, the positive cover including valid dependencies is minimized by using the data structure set-trie as well.

Doorenbos in [4] proposes an index structure for querying pre-conditions of rules to be matched while selecting the next rule to activate in a rule-based system Rete [6]. The index structure stores the conditions in separate nodes that are linked together to form the pre-conditions of rules. Common conditions of rules are shared among the rules: the lists of conditions with a common prefix share all nodes that form the prefix. Given

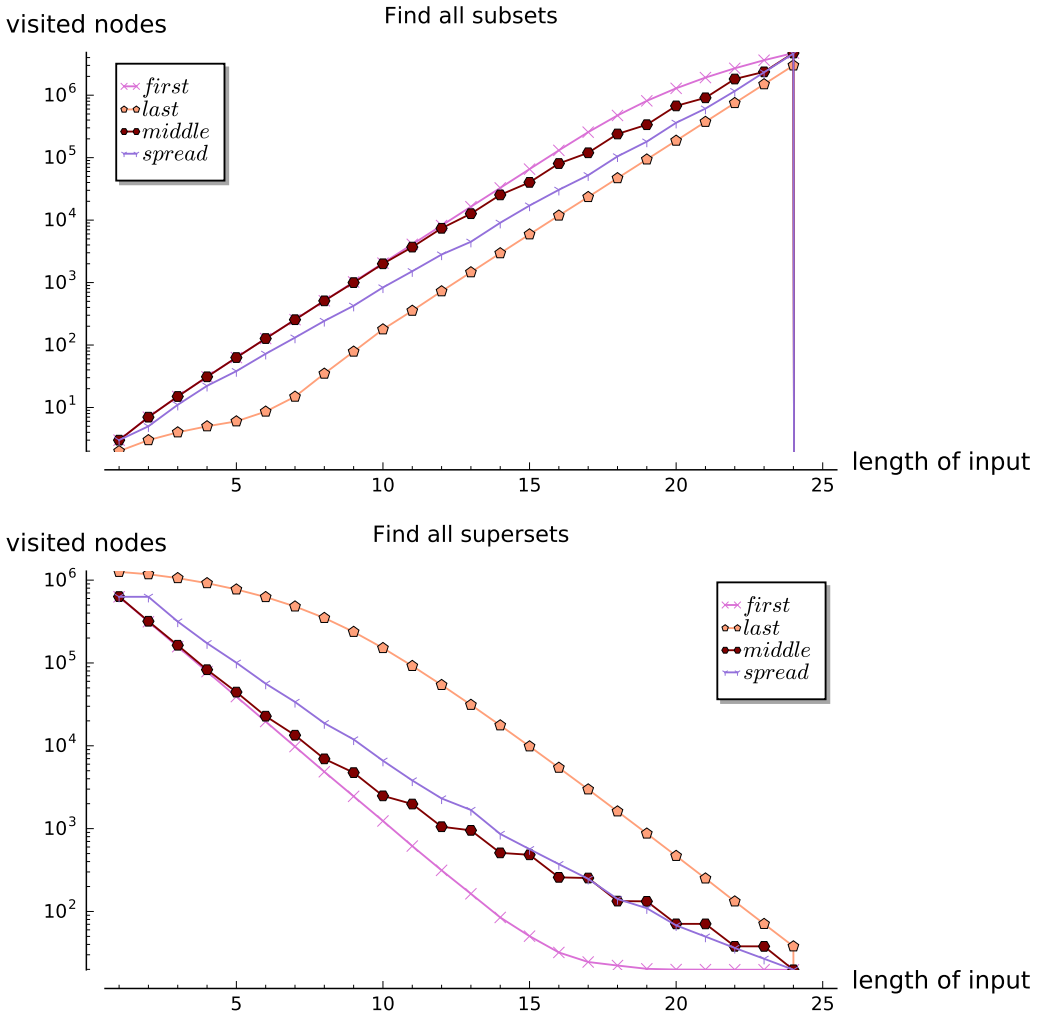

Figure 11: Experiment 3: Performance of GETALLSUBSETS(S,X) and GETALLSUPERSETS(S,X).

a set of conditions that are fulfilled, all rules that contain as pre-condition a subset of given set of conditions can be activated.

An index data structure for storing the sets of sets is proposed by Hoffman and Koehler as Unlimited Branching Tree (abbr. UBTree) [8]. The main difference to the representation of rules in the expert systems is that UBTree does not use variables. The children of a node are stored in a list attached to the node. A set is in UBTree represented by a path from root to final node; path is labeled by the elements of the set. The search procedures for the subset and superset problems are similar to those we propose, however, the main difference in procedures is that we explicitly use ordering of sets for search while Hoffman and Koehler give a general algorithm allowing other heuristic to be exploited. Our publication in 1993 [19] evidently presents the independence of the work.

5.3. Indexing set-valued attributes of object-relational databases

The sets are among the important data modeling constructs in the object-relational and object-oriented database systems. The *set-valued* attributes are used for the representation of properties that range over the sets

of atomic values or objects. Database community has shown significant interest in indexing structures that can be used as the access paths for querying set-valued attributes [21, 13, 7, 22, 23].

Set containment queries were studied in the frame of different index structures. Helmer and Moercotte investigated four index structures for querying set-valued attributes of low cardinality [7]. All four index structures are based on conventional techniques: signatures and inverted files. The index structures that they compare are: the sequential signature files, the signature trees, the extendable signature hashing, and B-tree based implementation of inverted lists. Inverted file index showed the best performance over the other data structures in most operations.

Zhang et al. [23] investigated two alternatives for the implementation of the set containment queries: a) a separate IR engine based on the inverted lists, and, b) the native tables of the relational database management system. They have shown that while RDBMS are poorly suited for the set containment queries, they can outperform the inverted list engine in some conditions. Furthermore, they have shown that with some modifications RDBMS can support containment queries much more efficiently.

Another approach to the efficient implementation of the set containment queries is the use of signature-based structures. Tousidou et al. [22] combine the advantages of two access paths: the linear hashing and the tree-structured methods. They show through the empirical analysis that S-tree that uses linear hash partitioning is an efficient data structure for the subset and superset queries.

6. Concluding remarks and future work

Experiment 1 and Experiment 2 indicate some of very intuitive properties of the data structure set-trie. Firstly, Experiment 1 shows that the number of the nodes visited by the set containment operations decreases with the increase of the size of the set-trie S—it is easier to find either a subset or a superset in the case when there are more sets in the set-trie. Secondly, Experiment 2 shows that the number of visited nodes increases with the increase of the alphabet size. However, in both cases the existent experiments do not reveal more precisely the nature of the functions expressing the influences of the increase of either the size of S or the size of Σ to the number of visited nodes.

The theoretical analysis, backed by the Experiment 3, shows that the efficiency of the presented algorithm is correlated by the characters of the input. In particular, the ordering of the elements in the alphabet Σ may play an important role in the process of algorithm design for the particular application. While the results of Experiment 3 may look very insightful, we are aware that the structure of each of four given inputs (first, last, middle, spread) is generated in a very artificial way, and that in real-world set databases such inputs may not be common. Furthermore, while in this paper we consider a natural probabilistic approach to generate our set-tries, one would need to test the performance of the mentioned structure on different set-databases, generated also by other well-known methods.

For all the above-mentioned reasons, we believe that the further study on the performance of the presented data structure is needed. In particular, it would be interesting to repeat similar tests on some real-world datasets, as well as some other probabilistic models. In addition, the sensitivity of the input should be studied further, as we believe it may lead to much better performance.

Finally, the initial experiments have been done to investigate if the data structure set-trie can be employed for searching the substrings and superstrings in texts. For this purpose the data structure set-trie is augmented with the references to the positions of words in the text. As in the case of indexes used in the area of information retrieval [12], set-trie can be decomposed into dictionary and postings. It would be interesting to study and analyze empirically the memory consumption and the efficiency of the set-trie when used for indexing huge quantities of texts.

7. Acknowledgments

The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. P1-00383).

[1] Baeza-Yates, R., Gonnet, G., Fast text searching for regular expressions or automation searching on tries. Journal of ACM, 1996, Vol. 43, No. 6, pp. 915-936.

[2] Blurn, A., Furst, M., Fast planning through planning graph analysis, Artificial Intelligence, Vol. 90, Issue 1-2, pp. 279-298, 1997.

[3] Charikar, M., Indyk, P., Panigrahy, R., New Algorithms for Subset Query, Partial Match, Orthogonal Range Searching and Related Problems. LNCS 2002, Vol. 2380, pp. 451-462.

[4] Doorenbos, R., Combining left and right unlinking for matching a large number of learned rules, AAAI-94, pp. 451-458, 1994.

[5] Flach, P.A., Savnik, I., Database dependency discovery: a machine learning approach. AI Communications, Vol. 12, No.3, IOS Press, 1999, pp. 139-160.

[6] Forgy, C., Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence, Vol. 19, pp. 17-37, 1982.

[7] Helmer, S., Moerkotte, G., A performance study of Four Index Structures for Set-Valued Attributes of Low Cardinality, The VLDB Journal - The International Journal on Very Large Data Bases, Volume 12 Issue 3, 2003 pp. 244-261.

[8] Hoffmann, J., Koehler, J., A New Method to Index and Query Sets, IJCAI, 1999.

[9] Johnson, N. L., Kemp, A. W., Kotz, S., Univariate discrete distributions Vol. 444. John Wiley & Sons, 2005.

[10] Mamoulis, N., Cheung, D.W., Lian, W., Similarity Search in Sets and Categorical Data Using the Signature Tree, ICDE, 2003.

[11] Mannila, H., Toivonen, H., Levelwise search and borders of theories in knowledge discovery, Data Mining and Knowledge Discovery Journal, Vol. 1, No. 3, 1997, pp. 241-258.

[12] Manning, C.D., Raghavan, P., Schütze, H., An Introduction to Information Retrieval, Draft, Cambridge University Press, 2009.

[13] Melnik, S., Garcia-Molina, H., Adaptive Algorithms for Set Containment Joins, ACM Transactions on Database Systems, Vol. 28, No. 2, 2003, pp. 1-38.

[14] Rivest, R., Partial-Match Retrieval Algorithms. SIAM Journal on Computing, 1976, Vol. 5, No. 1.

[15] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms, Second Edition, MIT Press, 2001.

[16] Savnik, I., Krnc, M., Škrekovski, R., Data structure set-trie, osebje.famnit.upr.si/˜savnik/set2-lib/, FAMNIT, University of Primorska, Koper, 2016.

[17] Savnik, I., Flach, P.A., Program for Inducing Functional Dependencies from Relations, www.cs.bris.ac.uk/˜flach/fdep/, University of Bristol, 1999.

[18] Savnik, I., Flach, P.A., Program for Inducing Multivalued Dependencies from Relations, www.cs.bris.ac.uk/˜flach/mdep/, University of Bristol, 2000.

[19] Savnik, I., Flach, P.A., Bottom-up Induction of Functional Dependencies from Relations. Proc. of KDD'93 Workshop: Knowledge Discovery from Databases, AAAI Press, 1993, Washington, pp. 174-185.

[20] Savnik, I., Index data structure for fast subset and superset queries, CD-ARES, IFIP LNCS, 2013.

[21] Terrovitis, M., Passas, S., Vassiliadis, P., Sellis, T., A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes, Proc. of ACM International Conference on Information and Knowledge Management, 2006.

[22] Tousidou, E., Bozanis, P., Manolopoulos, Y., Signature-based Structures for Objects with Set-valued Attributes, Information Systems 27, 2002, pp. 93-121.

[23] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G., On Supporting Containment Queries in Relational Database Management Systems, ACM SIGMOD, 2001.