Using Tries for Subset and Superset Queries

Stas Bevc, Iztok Savnik
Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Slovenia
stas.bevc@upr.si, iztok.savnik@famnit.upr.si

Abstract. *This paper presents a data structure for storing and querying sets. The data structure is based on tries, but because the order of elements in a set is not important (unlike letters in a string) we are able to implement additional subset and superset query operations besides the trie's insert and search operation. The operations and their algorithms are described. We empirically analyze the operations in a series of experiments and present the results.*

Keywords. Subset query, superset query, trie, suffix tree, data structure.

1. Introduction

Let us begin with a formal definition of the problem: let U be a set of ordered symbols, S a set of subsets from U, and let X be a subset of U. We are interested in the following queries:

1. is X a subset of any element from S?
2. is X a superset of any element from S?
3. enumerate all Y in S such that X is a subset of Y
4. enumerate all Y in S such that X is a superset of Y

The first two queries are a special case of the third and fourth query.

Let us show an example of the first query. Let U = {1, 2, 3, ..., 100} and S = {{2, 4, 5, 7, 15}, {9, 12, 25}, {3, 5, 6, 10, 11, 24}, {9, 12, 25, 38}}. If X = {3, 6, 11}, then X is a subset of {3, 5, 6, 10, 11, 24}.

The following is an example of the second query. Let U = {1, 2, 3, ..., 100} and S = {{2, 4, 5, 7, 15}, {9, 12, 25}, {3, 5, 6, 10, 11, 24}, {9, 12, 25, 38}}. If X = {7, 9, 10, 12, 15, 25, 32}, then X is a superset of {9, 12, 25}.

A real-world scenario could be querying for objects having certain properties. In this case U would be a set of properties labeled with symbols (e.g. a=wooden, b=red, ...), S would be a set of objects having properties from U, and X would be a subset of properties. To list objects from S having all properties from X, we would run the third query. To list objects from S having properties only from X, we would run the fourth query.

This paper presents a data structure named *SetTrie* and its algorithms that solve the described problem efficiently.

SetTrie is a tree data structure similar to the well known trie tree. Like tries, its nodes contain symbols and have pointers to child nodes. The path from the root node to a marked node represents a set (nodes containing the last symbol of a set are marked). Unlike in tries where the order of letters in a string is important, in SetTrie we sort the symbols of a set before insertion. As a consequence all nodes below any chosen node contain symbols that are greater than the symbol in the chosen node, and all nodes above it contain symbols lesser than the symbol in the chosen node. This property allows us to implement subset and superset query operations.

The operations are tested in three experiments where we analyze how the change of one parameter affects the performance of the data structure.

1.1 Related work

Our problem is somewhat similar to the problem of searching substrings in strings (for which Suffix trees can be used), but in our case the order in which the symbols appear in do not matter.

Baeza-Yates and Gonnet present an algorithm in [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.

In [2] Charikar et. al. present two algorithms to deal with a subset query problem. They extend their results to a more general problem of orthogonal range searching, and other problems.

Rivest [3] examines the problem of partial matching with the use of hash functions and Trie trees. He presents an algorithm for partial match queries using Tries.

The initial implementation of SetTrie was in the context of a datamining tool "fdep" where it serves for storing and retrieving hypotheses in the process of searching the theory of a relation [4].

2. SetTrie data structure

SetTrie is a tree data structure composed of nodes. Each node contains a symbol and an array of child nodes. A node with a symbol *k* can have children with symbols that are equal or greater than *k*. The latter is the key property of the data structure. SetTrie is not limited to sets, it works with multisets as well. For this reason we will call (multi)sets in the tree *words*: unlike a set, a *word* may contain more than one occurrence of a symbol. Nodes have a boolean flag to mark the last symbol of a word.

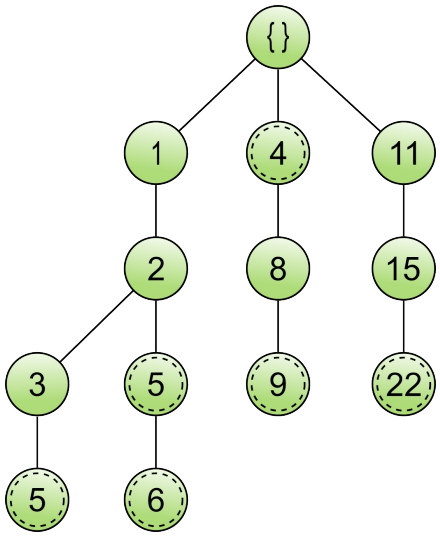

Figure 1. Example of a SetTrie tree

In Fig.1 we show an example of SetTrie containing words {1, 2, 3, 5}, {1, 2, 5}, {1, 2, 5, 6}, {4}, {4, 8, 9} and {11, 15, 22}. Nodes with a dashed inner circle represent flagged nodes.

2.1 Implementation

Our data structure is implemented in Java programming language. We will not describe every detail of the implementation, but only what is necessary to understand the operations described in the next section.

It is composed of two classes: the Tree class and the Node class. The Tree holds Nodes and implements the following operations: insert, search, existsSuperSet, getAllSuperSet, existsSubSet and getAllSubSet. An empty tree is composed of the root node only. Root node contains an empty set ({}).

The Node class has four attributes: the symbol of the node, the array of child nodes, a boolean flag to mark the last symbol of a word, and an integer for counting the number of children of a node. Symbols of the words must be integers (or converted to integers otherwise).

2.2 Operations

This section describes six operations and its algorithms we have implemented for our data structure. All operations are recursive. The insert and search operations work as in ordinary Tries.

Insert operation inserts a given word into the tree. Alg. 1 shows the operation algorithm in pseudocode.

```
1. insert(node, set, positionInSet)
2.  if positionInSet < set.length then
3.   if there is no node in child_array at element
     then
4.    create new_node with element and put it in
      child_array at element
5.    increase num_child in node
6.  if element is last in set then
7.    set the new_node's last_flag to true
8.  else
9.    insert(child_array at element, set,
      positionInSet+1)
```

Algorithm 1. The insert operation

The initial value of *node* is root, and *positionInSet* is 0. *set* represents the word being inserted and *element* is a symbol of the word. *positionInSet* marks which symbol from the word is currently selected. In line 4 a symbol is stored in the array at position *element* (ex. symbol 5 is stored at position 5). *num_child* is the counter for child nodes of a node. It is increased every time a node is created.

Search operation searches for a given word in the tree. It returns *true* if it finds all symbols of the word (i.e. the word is in the tree), and *false* as soon one symbol is not found (i.e. the word is not in the tree). The algorithm is shown in Alg. 2.

```
1. search(node, set, positionInSet)
2.  if positionInSet < set.length then
3.   if there is no node in child_array at element
     then
4.    set is not in the tree, return false
5.   else
6.    if element is last in set then
7.     if node's last_flag  true then
8.      set is in the tree, return true
9.     else
10.     set is not in the tree, return false
11.   else
12.    search(child_array at element, set,
       positionInSet+1)
```

Algorithm 2. The search operation

The initial value of *node* is root, and *positionInSet* is 0. *set* is the word being searched for. In line 6 the if-statement is true if the selected symbol is the last symbol of the given word. In line 7 the if-statement is true, if the node contains a last symbol of a word.

ExistsSuperSet operation checks if there exists a superset in the tree of the given word. It returns *true* if a superset is found, otherwise it returns *false*. Alg. 3 represents the algorithm of the operation.

```
1. existsSuperSet(node, set, positionInSet, offset)
2.  if not found then
3.   if found node with the last element then
4.    found = true
5.   else
6.    for (offset; node with possible child exists
      and not found; offset++)
7.     if found node with not last element then
8.      existsSuperSet(node.child, set,
        positionInSet+1, offset)
9.     else if element not found but possible child
       with element exists then
10.     existsSuperSet(node.child, set,
        positionInSet, offset)
11. return found
```

Algorithm 3. The existsSuperSet operation

The initial value of *node* is root, *found* is false, and *positionInSet* is 0. *set* is the given word, and *offset* is the offset in the array of child nodes. In line 3 the if-statement is true if the last symbol of the given word is found in a node. In line 6, the for-loop continues looping if a node with possible child exists. Here the operation takes advantage of the SetTrie properties. The array of child nodes is ordered, and child nodes cannot contain smaller symbols than the parent node. Also the child node counter is considered here. If all these criteria are met, the for-loop continues looping, otherwise *false* is returned.

GetAllSuperSet operation returns all existing supersets in the tree of the given word. It works similarly to the existsSuperSet operation, but it does not stop when finding the first superset. Instead it lists all the words contained in the nodes below such a node. Alg. 4 represents the algorithm of the operation.

```
1. getAllSuperSet(node, set, positionInSet,
   offset)
2.  if found node with last element then
3.   print all sets below this node
4.  else
5.   for (offset; node with possible child exists;
     offset++)
6.    if found node with not last element then
7.     getAllSuperSet(node.child, set,
       positionInSet+1, offset)
8.    else if element not found but node with
      possible child exists then
9.     getAllSuperSet(node.child, set,
       positionInSet, offset)
```

Algorithm 4. The getAllSuperSet operation

The initial value of *node* is root, *positionInSet* is 0, and *set* is the given word. There is a key difference between this algorithm and the algorithm of the existsSuperSet operation: the for-loop in line 5 does not have a *found* check, so the *print* command in line 3, can be reached at different parts of the tree.

ExistsSubSet operation checks if there exists a subset in the tree of the given word. It returns *true* if a subset is found, otherwise it returns *false*. Alg. 5 represents the algorithm of the operation.

```
1. existsSubSet(node, set, positionInSet)
2.  if not found then
3.   if node with element found then
4.    if node's last_flag true then
5.     found = true
6.    else if element not last then
7.     existsSubSet(node.child, set,
       positionInSet+1)
```

```
8.  increase positionInSet until next element
    different from current element
9.   existssubset(node, set, positionInSet)
10. return found
```

Algorithm 5. The existsSubSet operation

The initial value of *node* is root, *found* is false, *positionInSet* is 0, and *set* is the given word. In line 4, the if-statement is true if a word in the tree ends in that node. In line 8 the algorithm takes the next symbol from the word if it is the same as the currently selected symbol.

GetAllSubSet operation returns all existing subsets in the tree of the given word. It works similarly to the existsSubSet operation, but it does not stop when finding the first subset. It stops when there are no more symbols to take from the given word. The algorithm is shown in Alg. 6.

```
1. getAllSubSet(node, set, positionInSet)
2.  if found node with element then
3.   if node's last_flag true then
4.    print subset
5.   if element not last then
6.    getAllSubSet(node.child, set,
       positionInSet+1)
7.  increase positionInSet until next element
    different from current element
8.   getAllSubSet(node, set, positionInSet)
```

Algorithm 6. The getAllSubSet operation

The initial value of *node* is root, *positionInSet* is 0, and *set* is the given word. Line 4 prints the found subset of the given word.

3. Experiments

With our experiments we observe how the average number of visited nodes for existsSuperSet, getAllSuperSet, existsSubSet and getAllSubSet operations change, while changing one of the following three parameters: the number of words in the tree, the alphabet size (how many different symbols there are), and the maximum word length. We mark these parameters as *numTreeWord*, *alphabetSize* and *maxSizeWord*. The minimum word length was set to 2. The operation names are abbreviated as esr (existsSuperSet), gsr (getAllSuperSet), esb (existsSubSet) and gsb (getAllSubSet). For each of the three experiments the words in the tree and in the test set were randomly generated. Test set had 5000 words and were generated with the same parameters as the words in the tree. The words from the test set were used as inputs for the operations. The averages presented in this section were obtained by summing up the number of visited nodes for each word from the test set, and dividing it by the number of the words in the test set.

In experiment1 we created four trees with *alphabetSize* 30 and *numTreeWord* 50,000. *maxSizeWord* was 20, 40, 60 and 80 for tree1, tree2, tree3 and tree4 respectively. The number of nodes in the trees were: 332,182, 753,074, 1,180,922 and 1,604,698. We calculated the average number of visited nodes for each word length separately, but in table 1 (due to space constraints) we present the sum of these averages.

When we increased *maxSizeWord*, the average number of visited increased for esr, gsr and gsb operations. We repeated the experiment multiple times and the esb operation was generally between 2 and 10, but sometimes also visiting up to 1000 nodes for any tree. This seems to be dependent on what words were randomly generated. Because the words were randomly generated between 2 and *maxSizeWord*, gsr and gsb found more results when the tree contained words of a greater size.

Table 1. Experiment1 results

	esr	gsr	esb	gsb
tree1	2830	21755	2	427
tree2	5318	57958	45	2833
tree3	7130	106177	2	8530
tree4	9497	175105	8	17903

Table 2. Percentage of visited nodes

	esr	gsr	esb	gsb
tree1	0.85%	6.55%	0.001%	0.13%
tree2	0.71%	7.7%	0.01%	0.38%
tree3	0.6%	8.99%	0.0002%	0.72%
tree4	0.59%	10.91%	0.001%	1.12%

Table 2 shows the percentage of visited nodes in experiment1. For the esr operation the percentage of visited nodes decreases when the

word length increases, and for gsr and gsb it increases.

Table 3. Number of nodes in trees

	numTreeWord	nodes
tree1	10000	115780
tree2	20000	225820
tree3	30000	331626
tree4	40000	437966
tree5	50000	541601
tree6	60000	644585
tree7	70000	746801
tree8	80000	846388
tree9	90000	946493
tree10	100000	1047192

In experiment2 we created ten trees with *alphabetSize* 30 and *maxSizeWord* 30. We increased *numTreeWord* from 10,000 to 100,000. Table 3 shows the number of nodes in the trees.

The average number of visited nodes for experiment2 is shown in table 4.

Table 4. Experiment2 results

	esr	gsr	esb	gsb
tree1	1559	8907	12	477
tree2	2377	16583	8	720
tree3	2945	23765	3	923
tree4	3569	30940	9	1160
tree5	3915	37770	2	1283
tree6	4399	44552	2	1460
tree7	4788	50179	2	1615
tree8	5024	56546	2	1775
tree9	5475	64526	2	1852
tree10	5594	68661	2	1970

The esr, gsr and gsb operations visited more nodes in trees with more words, but the increase was minimal for esr and gsb operations. The gsr operation is much more affected by the increase in the number of words. On the other hand, the number of visited nodes for the esb operation decreased when increasing the number of words. This is to be expected, since it is "easier" to find a subset of a given word in a tree with more words.

Table 5. Percentage of visited nodes

	esr	gsr	esb	gsb
tree1	1.35%	7.69%	0.01%	0.41%
tree2	1.05%	7.34%	0.004%	0.32%
tree3	0.89%	7.17%	0.001%	0.28%
tree4	0.81%	7.06%	0.002%	0.26%
tree5	0.72%	6.97%	0.0004%	0.24%
tree6	0.68%	6.91%	0.0003%	0.23%
tree7	0.64%	6.72%	0.0003%	0.22%
tree8	0.59%	6.68%	0.0002%	0.21%
tree9	0.58%	6.82%	0.0002%	0.2%
tree10	0.53%	6.56%	0.0002%	0.19%

We also ran this experiment with *wordSize* 50. The number of visited nodes was higher for all four operations, as we have seen in experiment1 already, but the behavior of the operations was the same.

In table 5 we show the percentage of visited nodes in experiment2. In all four operations, the percentage of visited nodes slightly decreased when increasing the number of words in the tree.

In experiment3 we created five trees with *maxSizeWord* 50 and *numTreeWord* 50,000. *alphabetSize* was 20, 40, 60, 80 and 100 for tree1, tree2, tree3, tree4 and tree5 respectively. The number of nodes in the trees were: 869,373, 1,011,369, 1,069,615, 1,102,827 and 1,118,492. Table 6 shows the average number of visited nodes.

Table 6. Experiment3 results

	esr	gsr	esb	gsb
tree1	4244	108017	2	11605
tree2	8319	66373	35	2874
tree3	11241	52495	57	1289
tree4	12140	47930	46	789
tree5	13730	41533	39	547

When increasing *alphabetSize*, the tree becomes sparser – the array of child nodes in the nodes is larger, but the number of nodes in all five trees is roughly the same. For gsr and more noticeably gsb operation, the average number of

visited nodes decreased when *alphabetSize* increased. The esr operation on the other hand visited more nodes in trees with larger *alphabetSize*. The behavior of the esb operation does not seem to be in relation with the value of *alphabetSize*.

Table 7. Percentage of visited nodes

	esr	gsr	esb	gsb
tree1	0.49%	12.42%	0.0002%	1.33%
tree2	0.82%	6.56%	0.003%	0.28%
tree3	1.05%	4.91%	0.01%	0.12%
tree4	1.1%	4.35%	0.004%	0.07%
tree5	1.23%	3.71%	0.003%	0.05%

Table 7 shows the percentage of visited nodes in the tree for experiment3. Operations gsr and gsb visited a smaller percentage of the nodes in trees with a larger *alphabetSize*, and for the esr operation the percentage increased.

4. Conclusions

We presented a data structure and its operations created for solving the problem described in section 1.1. The data structure was tested in three experiments where we analyzed how the word size, the number of words in the tree, and the alphabet size affect the number of visited nodes of the operations. In all three experiments existsSubSet was least affected by the changes in the trees. The changes were most noticeable when changing the word size and the least when increasing the number of words in the tree. The most affected operations were getAllSuperSet and getAllSubSet.

5. References

[1] Baeza-Yates R, Gonnet G, Fast text searching for regular expressions or automation searching on tries. Journal of ACM 1996; 43(6): 915-936.

[2] Charikar M, Indyk P, Panigrahy R. New Algorithms for Subset Query, Partial Match, Orthogonal Range Searching and Related Problems. Lecture Notes In Computer Science 2002; Vol 2380, p. 451-462.

[3] Rivest R, Partial-Match Retrieval Algorithms. SIAM Journal on Computing 1976; 5(1)

[4] 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, p. 174-185.