Implementing collection of sets with trie: a stepping stone for performances ?

Simon Bachelard, Olivier Raynaud, Yoan Renaud

Implementing collection of sets with trie : a stepping stone for performances?

Simon Bachelard and Olivier Raynaud[1] and Yoan Renaud [2]

Research Report LIMOS/RR-06-06

7 juin 2006

[1]raynaud@isima.fr
[2]renaud@isima.fr

Abstract

Main operations of the *Set Collection* **A**bstract **D**ata **T**ype are *insertion*, *research* and *deletion*. A well known option to implement these operations is to use hashtable. Although hashtable does not admit good time complexities in the worst case, the practical time complexities are efficient. Another option is to use the data structure known as the trie. The trie is useful for two main reasons. Firstly, with such a data structure, mentionned operations admit very good theoretical time complexities. Secondly a trie can be seen as a compact representation of a collection of sets since some parts of them are merged together. Aim of this article is to evaluate performances of the trie data structure.
The Java language proposes an abstract class corresponding to the *Set Collection* A.D.T. operations .We propose in this article three different implementations of this abstract class. All of them are variations of the way to manage the sons of nodes. Theoretical complexities are then evaluated. After that, comparison with performances of hashtable are made in different contexts depending of the collection density, the set size and the ground set size. Finally we analyze our results and conclude in which cases the trie structure outperforms others structures.

Keywords: trie data structure, collection set abstract data type, performances comparison

1 Introduction

The concept notion as representation of knowledge comes from a real world modelisation. This modelisation supposes a world compounded by object sets having properties (or elements having attributs). The description of a concept summarizes the properties shared by a set of objects. The whole structure gathering and describing a set of concepts of a binary relation (objects/attributes) is known as the Galois lattice of the input relation. This Galois connection has been introduced in the sixties ([1]). Formal Concept Analysis, as a generalization of the Galois connection, focuses on the mathematization of *concept* and *conceptual hierarchy*. The Galois connection corresponding to a binary relation is then seen as a concept lattice of a context where meaning is given to concepts. Mathematical foundations of Formal Concept Analysis can be found in [2].

In a general point of view, the concept notion could be seen as a couple of sets (intention/extension). By this way the set of concepts is a collection of sets. In the same manner the context itself is a collection of sets. Finally, classical algorithms of lattices construction, of minimal basis or association rules generation manage some set collections. The natural operations to manage such a collection are *insertion*, *research* and *deletion*. By moving together a kind of objects and a set of operations we are able to define an **A**bstract **D**ata **T**ype (A.D.T.) :

Abstract Data Type : *Set Collection* ;
Object : a set collection $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ on a ground set X ;
Operation :

- $creation()$: creates and returns a new empty collection ;
- $insertion(\mathcal{C}, C)$: inserts in the collection $\mathcal{C}$ the set C ;
- $research(\mathcal{C},\ C)$: returns "true" **if and only if** $\exists i \in [1..m]$ such that $C_i = C$;
- $deletion(\mathcal{C},\ C)$: removes the set C from the collection $\mathcal{C}$;

Let us see in an example how to use this abstract data type. We consider the following generation problem which corresponds to the computation of the closure under union operator of a given collection.

Problem 1 *Closure under union operator*
Data input : *a set collection* $\mathcal{M} = \{M_1, M_2, ..., M_m\}$ *on a ground set* $X = \{1, 2, ..., n\}$ *such that* $\forall i \in [1..m],\ M_i \subseteq X$ *;*
Result : *a set collection* $\mathcal{C}_\mathcal{M} = \{\bigcup_{M \in I} \mid I \subseteq \mathcal{M}\}$.

Algorithm 1 which solves the previous problem was proposed in [5] to compute maximal rectangles of a binary relation. Author of [6] have shown

how to use it to compute the Hasse diagramm of the lattice of ideals, of the Galois lattice or of the completed Mac Neill lattice of the given relation. The fact is that by using an efficient A.D.T., the algorithm becomes very simple and the implementation trivial.

Algorithm 1: Union closure()

```
Data  : a set collection M on a ground set X
Result: a set collection C_M (C_M is closed under union operator.)
begin
    C_M = creation() ;
    for (M ∈ M) do
        for (C ∈ C_M) do
            if (not research(C_M, C ∪ M) then
                insertion(C_M, C ∪ M) ;
    return C_M ;
end
```

One question is then to evaluate the efficiency of the implementation of the operations. We think the efficiency should be supported by three criterions :

- need for memory ;
- theoritical time complexity in the worst case ;
- practical time complexity ;

The need for memory seems to be noticeable in a F.C.A. context for which the size of set collections could be huge. Moreover the practical studies in [3] has shown once again that a good theoritical complexity does not always lead to practical performances. In particular, this last point was highlighted with the implementation of the Nourine and Raynaud's algorithm ([6]) whose practical performance was disappointing.

The aim of this article is to evaluate the previous three criterions for the data structure known as the trie. The trie is useful for two main reasons. Firstly, with such a data structure, the *Set Collection* A.D.T. operations admits very good worst case theoritical time complexities. In particular the research operation can be done in a time complexity which does not depend on the collection size. Moreover, these time complexities are easy to compute. Secondly, a trie could be seen as a set collection compact representation since some parts of them are merged together. We will see that this point is measurable. Another well known option to implemente these operations is to use hashtable. Although hashtable does not admit good worst case time

complexities, the practical time complexities are efficient. For this reason the practical time complexities evaluation of our three trie implementations will be made by comparison whith hashtable performances. We choose the Java language since this language proposes an abstract class corresponding to the *Set Collection* A.D.T. operations. Tests will consist only in changing code of the called methods (the program being the same) and measuring the different operations execution time

This article is then organized as follows. In the next section we show that the *Set Collection* A.D.T. is compatible with the abstract class *Map* of Java language. Then we describe the trie properties and propose three different implementations. In the third section we describe the protocol of comparison test between the different implementations. We then give global results in different tabs depending of the collection density, the sets size and the ground set size. Finally, we conclude our report by listing cases in which the trie structure outperforms others structures.

2 Abstract Data Type : Map

To simulate the *Set Collection* A.D.T. we can use a Map A.D.T. similar to the Map interface of Java language. This Map A.D.T. maps keys to values. In case of set collection, the keys are the collection sets. The Map A.D.T. supplies the following operators :

- `new()` operator : creates a *Map* object and returns an empty map.
- `get(e)` operator : returns the value associated to the key `e` if this key maps a value, otherwise returns `Nil`.
- `put(e,value)` operator : inserts set `e` in the map and associates `value` to it.
- `remove(e)` operator : removes set `e` in the map.

There is a natural equivalence between the previous operators and operators of the *Set Collection* A.D.T. Main difference is that a value is associated to each set. Even if this value seems useless in general, one has to notice that usually some additional informations are associated to a set : an extention to an intention, a support to a set, a closure to a set... Java language defines an abstract class *Map* (or Interface) and proposes two main implementations. The first one uses a binary tree (TreeMap) and the other one uses hashtables (HashMap), the key being of any type (the only constraint being existence of a comparison operator). In the following of this article, we propose three new abstract class *Map* implementations. Each of them take advantage of the fact the key is a set. For this reason these implementations are variations of the tree structure known as the trie. Before describing the different imple-

mentations, let us see some formal results about trie (i.d. lexicographic tree).

2.1 Lexicographic tree definition

In litterature we often find the structure of trie (i.d. lexicographic tree) to store a set collection $\mathcal{M}$ on a ground set X. One constraint of the trie structure is to define an order on elements of X. This structure allows a precise evaluation of *insertion*, *deletion* and *research* operations time complexities. We used these evaluations to proove some complexity results in [4] and [6] or [7].

Let us give a formal definition of a lexicographic tree corresponding to a set collection.

Definition 1 *Let $\mathcal{M}$ be a set collection defined over X, with a total order on X denoted by $<_X$. A unique lexicographic tree is associated to $\mathcal{M}$ such that :*

- *Each edge of the tree is labeled with an element of X ;*
- *To each marked node of the tree corresponds a set of $\mathcal{M}$;*
- *To each set m of $\mathcal{M}$ correspond a unique path in the tree (starting from root and ending with a marked node) such that the union of labels in this path corresponds exactly to the set m.*
- *For any path from the root to any node, the order of the successive labels respects the order defined by $<_X$;*
- *The order of edges leaving a node respects the order defined by $<_X$.*

Figure 1 gives an example of collection and its associated lexicographic tree.

A boolean can be associated to a node of the tree in order to indicate if the node is marked (i.e. corresponds to a set of the collection and thus to a key) or not. Any field type can be associated to the node for storing the value associated to the key (on marked nodes only).

2.2 Lexicographic tree implementation

To test the lexicographic tree performances, we propose three different implementations of the abstract Java class *Map*. Each of them use the trie structure. The variations concern the choice made to store the children set of each node. Let us now describe the three structures and give the representations corresponding to the collection $\{abc, ad, bcde\}$ with $X = \{a, b, c, d, e\}$.

- **ADT_MappingList** (class ADT_L, cf. figure 2) : In that case the children set of a given node is implemented with lists. Each node of the

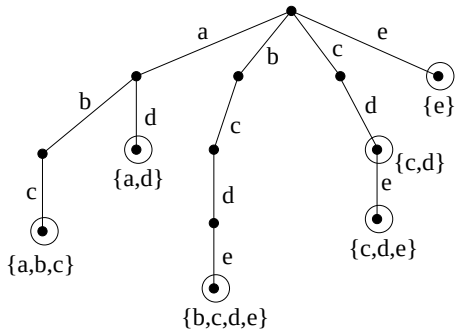

FIG. 1 – Lexicographic tree corresponding to the collection $\{abc, ad, bcde, cd, cde, e\}$, with $X = \{a, b, c, d, e\}$ and $<_X$ being the alphabetical order. Circled nodes are the marked nodes corresponding to the collection sets.

tree contains an element of the ground set X and a list of references to the children nodes of this node. The references are sorted in alphabetical order of elements corresponding to the children nodes refered in list. When a node has an empty list, there is a mapping value stored in this node corresponding to the mapping object of the set defined by the path for this node. Notice that some nodes could have both a mapping value and a list of children.

Proposition 1 *If the children set of a given node is implemented with* `Lists` *then :*

- *The* `put(m,value)` *operator as an* $\mathcal{O}(|X|)$ *time complexity ;*
- *The* `get(m)` *operator as an* $\mathcal{O}(|X|)$ *time complexity ;*
- *The* `remove(e)` *operator as an* $\mathcal{O}(|X|)$ *time complexity ;*

Access complexity is due to the fact that an element of X appears only once in a set. Cost of a node creation is done in constant time.

- **ADT_MappingTable** (class ADT_T, cf. figure 3) : In that case the children set of a given node is implemented with an array. Each node of the tree contains an element of X and an array which is indexed by the labels. The entries contain either NIL, if the node has no child corresponding to the label, or reference to the child. For each node, the array size is equal to the X size minus the position of label stored in the node. When a node has an array which all entrie contains NIL, there is a mapping value stored in this node corresponding to the mapping object of the set defined by the path. It is more restricting than the others because of the array fixed size but we will see that this structure gives a better efficiency for the different operators.

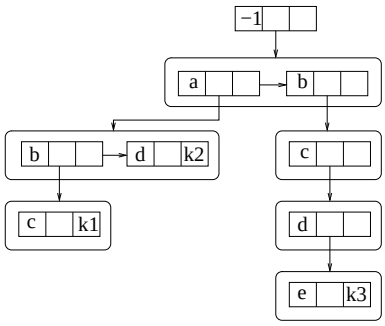

FIG. 2 – Exemple of a lexicographic tree whith the ADT_MappingList. The k1, k2, k3 values are the mapping objects corresponding to the set defined by the path in the tree.

Proposition 2 *If the children set of a given node is implemented with* `Arrays` *then :*
– *The* `put(m,value)` *operator as an* $\mathcal{O}(|m| \times \theta)$ *time complexity ;*
– *The* `get(m)` *operator as an* $\mathcal{O}(|m|)$ *time complexity ;*
– *The* `remove(e)` *operator as an* $\mathcal{O}(|m|)$ *time complexity ;*
Acces complexity is due to the fact that we have direct access to a child labeled by a given element of X. In that case θ corresponds to the allocation time plus the initialization time of a tab in the given language.

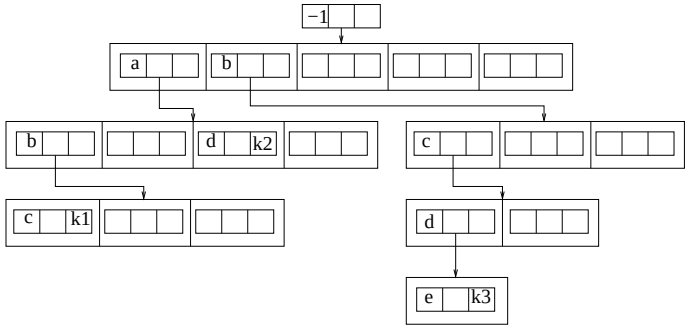

FIG. 3 – Exemple of a lexicographic tree whith the ADT_MappingTable. The k1, k2, k3 values are the mapping objects corresponding to the set defined by the path in the tree.

– **ADT_MappingMap** (class ADT_M, cf. figure 4) : In this last case the children set of a given node is implemented using the Java class

HashTable. Each node of the tree contains an element of X and an HashMap. HashMap entries contain either NIL, if the node has no child corresponding to the key value of the map, or the reference to a child. The HashMap size is defined by Java and the colisions are managed to have a good efficiency. This class gives no guarantees concerning the map order. In particular, it does not guarantee that the order will remain constant over time. When a HashMap contained on a node is empty, there is a mapping value stored in this node corresponding to the mapping object of the set defined by the path for this node.

Proposition 3 *If the children set of a given node is implemented with* `Map`*s then :*

– *The* `put(m,value)` *operator as an* $\mathcal{O}(|m| \times \theta')$ *time complexity ;*
– *The* `get(m)` *operator as an* $\mathcal{O}(|m|)$ *time complexity ;*
– *The* `remove(e)` *operator as an* $\mathcal{O}(|m|)$ *time complexity ;*

θ' corresponds to the time needed for the *Map* allocation (or reallocation) in a new node. One has to notice that the last proposition results do not correspond to worst case time complexity but to practical time complexity.

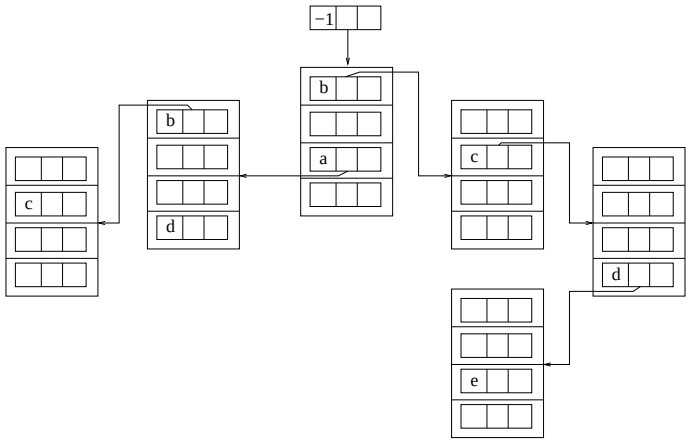

FIG. 4 – Exemple of a lexicographic tree using the ADT_MappingMap to store node children. The k1, k2, k3 values are the mapping objects corresponding to the set defined by the path in the tree

For this study, we have limited the implementation of the three classes (ADT_MappingList/Table/Map) to main operators *new()*, *get()*, *put()* and *remove()*. A next work will allow to complete these implementations. Use of our classes can be done in the same manner as any other mapping library in Java. Netherveless, the *new()* operator of the ADT_MappingTable admits a different specification. Indeed, each node creation needs to define a tab

which size has to be given as parameter. This size depends of the ground set size. For more information, source code and documentation can be found at www.isima.fr/raynaud/.

3 Experimental evaluation

3.1 Protocol

Evaluation of our three Map A.D.T. implementations and the two Java Map A.D.T. (*TreeMap* and *HashMap*) are carried out on a 2.4GHz Pentium III PC with 2 Go RAM running Red Hat Linux distribution. All Map A.D.T are evaluated on integer set collections (example files) :

- **Full** collection (*Full(n)*) : the collection of all sets on ground set X={1,2,...,n} without the empty set. The size of this collection is equal to 2^n.
- **Short** collection (*Short(n,p)*) : A collection of 10^p sets on ground set X={1,2,..,n} where the size of all sets is short. This size is determined by a probabilistic law. The file is randomly generated.
- **Large** collection (*Large(n,p)*) : A collection of 10^p sets on ground set X={1,2,..,n} where the size of all sets is long. This size is determined by a probabilistic law. The file is randomly generated.

In each example file, sets are generated in alphabetical order and there is no test for set unicity in the collection. The only restriction is the size of the sets. In the following we evaluate time (in millisecond) and memory (in megabytes) performances for the *get()*, the *put()* and the *remove()* operators. The *put()* operation consists in reading an example file and inserts in a trie each set of the corresponding collection. Note that the CPU time evaluated for this operation takes into account the loading time of the example file. The *get* and the *remove* operations are carried out on 10 000 sets randomly chosen without obligation to belong to the tested collection.

3.2 "Need for memory" comparison

We made numerous "need for memory" evaluation to compare the different A.D.T. *Map* implementations. We decide to put in light the most significant of them.

1. In figure 1 we compare the need for memory of each implementation in a **Full** context. The ground set size is between 15 and 20 and then the size collection is between 2^{15} and 2^{20}.

2. In figure 2 we compare the need for memory of each implementation for **Short** collections. In that case, n belongs to $[3, 6]$ interval and X size is fixed to be equal to 99.
3. Figure 3 and 4 compare the need for memory with X size fixed to be equal to 499 for **Short** collection and **Large** collection.

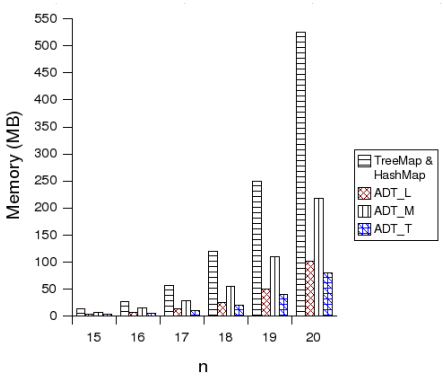

Fig 1 : Need for memory in case of **Full** collection with $|X| = n$.

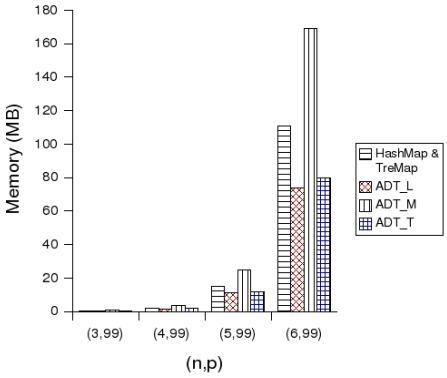

Fig 2 : Need for memory in case of **Short** collection with $|X| = 99$.

All test results concerning Java HashMap and TreeMap are merged since they are similar. One has to notice that all measures are taken when the whole collection is inserted into the structure. A first observation shows that the trie efficiency is proven (with a list or array implementation) in a **Full** context. A second observation shows that a trie implementation (with list or array) is very competitive in a **Short** context. But greater is the X size, less the array implementation is efficient. A third observation concerns the case of **Large** context and of a X size value equal to 499. In that case, for n values smaller than 5 the Java HashTable Map is the more efficient. With n greater than 5 the Java virtual machine has not enough memory to execute the test with our three new implementations.

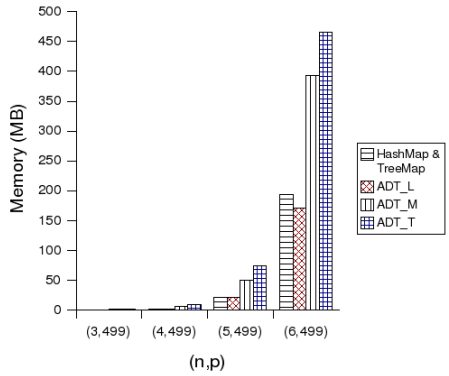

Figure 3 : Need for memory in case of **Short** collection with $|X| = 499$.

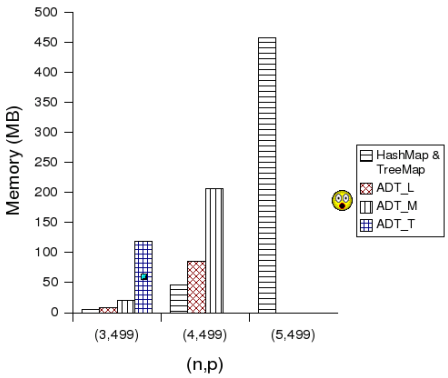

Figure 4 : Need for memory in case of **Large** collection with $|X| = 499$.

3.3 CPU time executions comparison

We made different tests on CPU time execution for *put()*, *get()* and *remove()* operation to compare the different A.D.T *Map* implementations. In Tables 1 and 2, we compare CPU times execution for HashMap and TreeMap and Tables 3 and 4 show CPU times execution for implementations ADT_L, ADT_M and ADT_T. In Tables 1 and 3, time measures are carried out on **Full** collections when the collection size is between 2^{12} and 2^{20}. For Tables 2 and 4, time measures are carried out on **Short** collections and **Large** collections. Other tests were made for this study but we only show the most significant one in these Tables.

	CPU times (milliseconds)					
	HashMap			TreeMap		
Full(n)	put	get	remove	put	get	remove
12	78	101	90	274	90	28
13	134	149	155	639	51	30
14	265	138	157	792	42	33
15	484	94	96	981	45	38
16	1197	67	42	1275	87	71
17	2366	67	39	2251	91	74
18	5251	66	41	5692	41	65
19	12139	66	38	14928	47	52
20	35770	100	195	34078	65	65

TAB. 1 – Experimental evaluation on **Full** examples using HashMap and TreeMap

	CPU times (milliseconds)					
	HashMap			TreeMap		
Short(n,p)	put	get	remove	put	get	remove
(3,99)	88	196	141	43	178	25
(4,99)	215	69	40	281	154	86
(5,99)	924	127	127	1835	77	80
(6,99)	8045	137	138	14312	96	106
(3,499)	44	202	161	72	246	182
(4,499)	349	94	56	279	78	71
(5,499)	1131	73	49	1783	97	107
(6,499)	14836	96	53	21301	112	124
Large(n,p)	put	get	remove	put	get	remove
(3,99)	57	288	193	87	127	36
(4,99)	698	69	42	379	156	48
(5,99)	3876	178	192	6054	42	30
(3,499)	324	81	47	236	43	28
(4,499)	1864	123	128	2293	119	26
(5,499)	26317	131	132	28972	101	28

TAB. 2 – Experimental evaluation on **Short** and **Large** examples using HashMap and TreeMap

	CPU times (milliseconds)								
	ADT_*L*			ADT_*M*			ADT_*T*		
Full(n)	put	get	remove	put	get	remove	put	get	remove
12	103	71	56	142	27	30	33	9	16
13	338	45	47	281	10	25	69	9	24
14	384	35	109	351	11	21	116	9	19
15	474	34	38	568	11	21	255	11	29
16	755	28	38	868	11	23	535	11	29
17	1771	38	47	1480	13	27	1132	12	40
18	5251	66	41	5692	41	65	3191	28	40
19	7240	29	49	7407	13	30	6083	12	19
20	15212	40	41	16310	14	32	12965	11	20

TAB. 3 – Experimental evaluation on **Full** examples using ADT_MappingList, ADT_MappingMap and ADT_MappingTable

	CPU times (milliseconds)								
	ADT_*L*			ADT_*M*			ADT_*T*		
Short(n,p)	put	get	remove	put	get	remove	put	get	remove
(3,99)	37	117	92	24	105	124	14	50	25
(4,99)	147	66	68	231	63	44	97	20	35
(5,99)	961	185	197	1196	73	35	543	30	35
(6,99)	10533	164	184	8493	48	40	5823	18	36
(3,499)	62	290	99	40	32	33	15	25	28
(4,499)	178	89	105	236	61	68	297	61	71
(5,499)	1547	132	135	1557	52	39	1183	17	28
(6,499)	27878	271	237	15873	65	52	14099	40	29
Large(n,p)	put	get	remove	put	get	remove	put	get	remove
(3,99)	91	87	99	82	72	28	92	80	38
(4,99)	366	35	39	707	35	17	437	11	20
(5,99)	5978	72	71	6211	19	24	7467	12	22
(3,499)	267	79	60	487	57	32	1288	36	24
(4,499)	2659	223	144	3967	28	24	none	none	none
(5,499)	none	none	none	none	none	none	none	none	none

TAB. 4 – Experimental evaluation on **Short** and **Large** examples using ADT_MappingList, ADT_MappingMap and ADT_MappingTable

A first look on tables 1 and 2 results shows that, in general, HashMap has better performances than TreeMap for the operation *put* without taking into account the chosen collections. On **Full** collections (Table 1), on one hand there is a significant variation of performances on collections with $|X| \leq 15$. On the other hand, efficiency variation is not significant for *get* and *remove* operations. Then, if we consider the Table 3 and 4 results, we notice that ADT_T has appreciably better times than the two others. Nevertheless, the performance of ADT_T decreases when the collection sizes or the ground set size grows up. Considering example of **Large**(5,99) collection in Table 4, we see that *put()* operator time execution for ADT_T is longer than those of ADT_L and ADT_M. As the same than before, efficiency variation is not significant for *get()* and *remove()* operations. Finally, if we consider all results, we observe that CPU time execution for the *put()* operation depends on the treated collection size. But it doesn't seem to be the same for *get* and *remove* operations. This result confirms theoretical evaluation complexity which shows that these operations complexities do not depends of the collection size. But, we do not see significant dependancies of these operations with the ground set size contrary to theoretical evaluations. Times

being small and, as a fact, not really significant, we cannot make definitive conclusion on this point.

4 Conclusion

In this article we have proposed three trie structure implementations. We have evaluated their performances for the Map A.D.T. operations. Theoritical time complexities results make the trie structure the most efficient. Our practical memory measure shows that the ADT_MappingTable is very efficient in a dense context. But it becomes unusable when the ground set size grows to 500, the collections sets being large too. Let us note that in that case the ADT_MappingList stays competitive. CPU time measures show that in general the ADT_MappingTable is the most efficient for the *put()* operation. Measures for the *get()* and *remove()* operations seems to be not significant.

Références

[1] M. Barbut and B. Monjardet. *Ordre et classification*. Hachette, 1970.

[2] B. Ganter and R. Wille. Formal concept analysis. In *Mathematical foundation*. Berlin-Heidelberg-NewYork :Springer, 1999.

[3] S. Kuznetsov and S. Obedkov. Comparing performance of algorithms for generating concept lattices. In *1th International Workshop on Concept Lattices-based KDD*, 2003.

[4] R. Medina, C. Noyer, and O. Raynaud. Efficient algorithms for clone items detection. In *CLA'05*, pages 70–81, 2005.

[5] E. M. Norris. An algorithm for computing the maximal rectangles of a binary relation. *Journal of ACM*, 21 :356–366, 1974.

[6] L. Nourine and O. Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, volume 71 :199–204, 1999.

[7] L. Nourine and O. Raynaud. A fast incremental algorithm for building lattices. *Journal of Experimental and Theoritical Artificial Intelligence*, 14 :217–227, 2002.