https://doi.org/10.37420/j.eeer.2024.002

Adaptive Binary Search Tree for Integers Sets with Few Holes

Ruixin Wang[1*] ,Nicolas Barnier[2]

[1]Laboratory of Complex System Safety and Intelligent Decisions, CAUC-ENAC Joint Research Center of Applied Mathematics for ATM, Civil Aviation University of China, Tianjin, China,ruixin.wang@recherche.enac.fr

[2]ENAC Lab, Université de Toulouse, France,nicolas.barnier@recherche.enac.fr

[*]Corresponding author,E-mail:ruixin.wang@recherche.enac.fr

Abstract

We present the Adaptive Compact Tree (AC-Tree), a data structure similar to a self-balancing binary search tree (BST) that improves storage and operations for sets of integers with few “holes”. AC-Trees can be implemented on top of any classic BST (e.g. AVL trees [3]), holding discontiguous intervals at each node to obtain a compact representation only requiring O(k) storage and supporting efficient O(log k) operations, with k the number of intervals. These properties can improve the performances of applications, like Constraint Programming (CP) solvers, that incrementally remove values and intervals on domains initially consisting of one large interval. First experiments show that AC-Trees outperform classic self-balancing BSTs in most cases and improve CP solvers performances for problems with large domains.

Keywords:self-balancing binary search tree, adaptive data structure, compact representation

1 INTRODUCTION

Self-balancing binary search trees (BST) [3] are a well-known and widely used implementation of the dynamic ordered set data structure with O(n) storage and O(log n) query, addition and removal operations, where n is the set cardinal. However, in some applications like Constraint Programming (CP), solvers have to maintain finite domains (i.e. large sets of integers that represent the possible values of variables), which generally start out as single intervals before elements or sub-intervals are removed by constraint propagation. In such a context, standard BSTs suffer from a lack of compactness of representation of intervals, preventing efficient storage and interval arithmetic operations, and thus hindering the solver performances. As such, BSTs are seldom used to implement domains within CP systems.

For example, the FaCiLe CP library [1] uses an ordered list of intervals called a range sequence [8], which also maintains the maximum and cardinal incrementally to minimize the cost of these frequently used operations, while preserving the compactness of representation of intervals by their bounds. However, when the propagation of combinatorial constraints induces a growing number of holes in large domains, this sequential representation becomes inefficient as operations take at least linear time w.r.t. the number of intervals.

We therefore introduce the Adaptive Compact Tree (AC-Tree), a data structure similar to a self-balancing BST, and the associated algorithms to perform standard and interval set operations. AC-Trees can be built on top of any self-balancing BST, but holding dis-contiguous[1] integer interval bounds at each node instead of single elements. AC-Trees are thus able to optimize the amount of memory required to represent large sets as it needs O(k) storage only for a set with k intervals, regardless of the cardinal of the set. On the other hand, the storage cost may be twice the one of a BST for sets made of isolated singletons.

However, during the addition or removal of a single element, nodes in an AC-Tree are not created or deleted as in a classic BST whenever the element interferes with the current intervals of the set: the addition of x implies to merge intervals [a, x 1] and [x + 1, b] in a single node [a, b] if they are both present and, conversely, the removal of x contained in an interval [a, b] adds one node to the tree if a < x < b ([a, b] becomes [a, x 1] and [x + 1, b]). So maintaining the dis-contiguity property incurs a constant overhead of extra intervals bounds checks at each visited node and possibly the same amount of rotations as for the remove operation on a classic BST (and vice versa for the removal if one node must be created). These costs can be amortized during subsequent operations for which computation steps may be saved over classic BST operations by benefiting from the compactness of representation of AC-Trees. In other cases, only the adjustment of one bound of a node needs to be performed instead of the creation or deletion of one node as in a BST. Furthermore, interval set operations, which are very frequent when handling arithmetic constraints within a CP solver, can be performed much more efficiently than with a standard BST.

Moreover, to benefit from the representation of finite domains by BSTs within a CP solver, other pervasive operations like cardinal, minimal and maximal elements must also have a constant-time complexity and must therefore be maintained incrementally. This can be achieved at the extra cost of adding these data at each node of the tree, which means that the node of an AC-Tree holds eight integers and pointers attributes (i.e. twice as much as classic BSTs): the bounds of its interval, its left and right children, its height or imbalance (for an AVL tree), the extrema of the corresponding subtree and its cardinal. These extra data can be maintained in constant time while performing classic BST operations and enable to optimize the other set operations further.

To the authors' knowledge, the idea to represent range sequences as BSTs has been introduced by [7] as Gap Intervals Trees to implement the finite domains of the Naxos CP solver [6]. But the corresponding

1 Integer intervals [a, b] and [c, d] (with a < c) are dis-contiguous iff b + 1 < c, which is not necessarily the case for simply disjoint intervals.

BSTs are not balanced, which could lead to pathological sequential structures with linear worst-case time complexity, only the bounds of the holes between intervals are stored in the nodes and only two operations are provided (removal and dis-jointness of an interval). With such intervals BSTs, the Naxos solver is able to outperform ILOG Solver and ECLiPSe, two of the most widely used CP solvers, on a bioinformatics RNA motifs detection problem with huge domains. Note that AC-Trees should not be mistaken with Interval Trees described in [2], which are designed to represent possibly overlapping real intervals and query them efficiently, though they stem from the same general technique that consists in augmenting a classic data structure.

AC-Trees have been implemented on top of the Set module of the OCaml language standard library [4] which provides functional AVL trees, and thoroughly tested against the Set module itself. Our first results show that AC-Trees outperform standard BSTs on random sets for almost all operations except in a few cases where the proportion of holes is very high (the corresponding source code is available at recherche.enac.fr/~barnier/actree), as well as FaCiLe range sequences on the RNA motifs detection problem mentioned in[7].

We first describe in section 2 the AC-Tree data structure and several algorithms that implement set operations while maintaining intervals dis-contiguity. Section 3 then focuses on how the cardinal and extrema are also maintained and exploited to improve set operations. Section 4 presents our experiments with set operations on randomly generated data with our implementation of AC-Trees on top of the Set module of the OCaml standard library, as well as a CP application with the FaCiLe constraint solver [1] on the aforementioned bioinformatics problem. Eventually, we conclude and detail possible further works in section 5.

2 MAINTAINING DIS-CONTIGUOUS INTERVAL IN A BINARY SEARCH TREE

The self-balancing BST (e.g. AVL trees, Red-Black trees) is a well-known and pervasive implementation of the dynamic ordered set data structure [3]. It is able to perform simple operations (e.g. addition, deletion, membership) in O(log n) time w.r.t. its cardinal n, which is the optimal worst-case access time for comparison-based data structures, while requiring O(n) storage. However, when storing discrete elements like integers, BSTs do not exploit the fact that the intervals that may occur within the set could be more efficiently represented by their bounds, leading to a data structure with only O(k) storage requirement on which the same operations take O(log k) time only, with k the number of dis-contiguous intervals.

In applications like CP solvers, variables often take their values from an initial interval, e.g. [0, d 1], that must support single value and range deletions. The most common implementations [5] are bit vectors, which provide constant-time membership and removal (and very efficient union and intersection based on fast hardware operations), but with a storage cost of O(d), and range sequences, which basically amount to sorted dis-contiguous intervals lists. For huge initial intervals (e.g. RNA motifs detection in bioinformatics [7]), the former can be intractable because of memory overflow, while the latter only requires O(k) storage, but at the cost of operations in O(k) time (and possible stack overflows).

Surprisingly, the idea to benefit from the logarithmic time operations of BSTs and the compactness and efficiency of representation of range sequences has only emerged a few years ago in [7], and only with unbalanced BSTs, which could lead to pathological trees with linear height, depending on the order of operations on the structure. The authors only specify two operations (interval removal and dis-jointness) and their Gap Intervals Trees actually represent the complement of the set to specifically optimize the data structure for CP operations (which only remove values from the variables domains). Results seem promising as their solver outperforms bit vectors based ones (ILOG Solver and ECLiPSe), but the significance of such unbal-

Definition 1 (Range sequence (RS) of a finite integers set). The *range sequence* of S is the shortest ordered sequence of disjoint intervals$< [a_1, b_1], \ldots, [a_k, b_k] >$such that$S = \bigcup_{i=1}^{k} [a_i, b_i]$, with $a_i \leq b_i, \forall i \in [1, k]$ and $a_i \leq a_{i+1}, \forall i \in [1, k-1]$.

As a RS is the shortest sequence of intervals, it cannot contain two intervals [a, b] and [b+1, d], otherwise the sequence with interval [a, d] would be shorter. Therefore, all successive intervals and are dis-contiguous, i.e. , . A RS can be efficiently computed from an integers list in O(n log n) by sorting, then traversing the list in linear time. However, for CP applications, domains generally start as intervals, for which a RS or an AC-Tree of one node can be built in constant time.

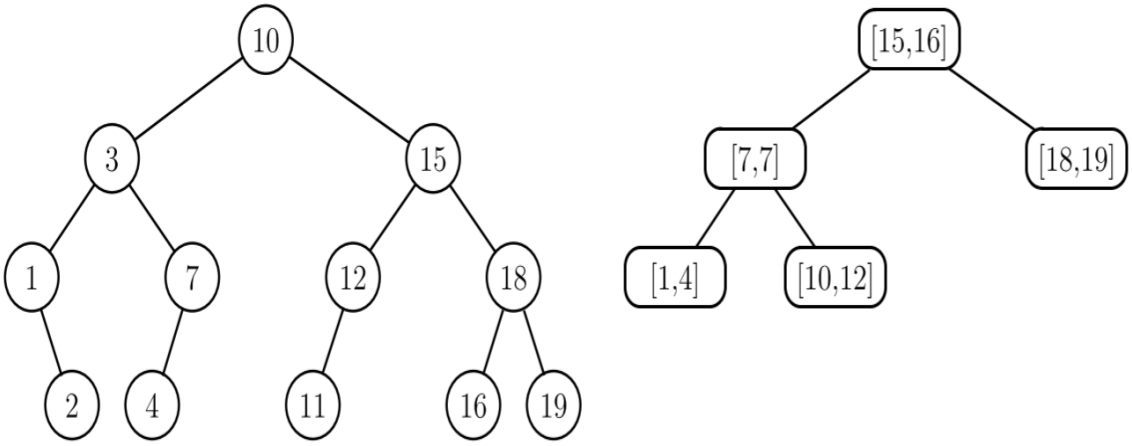

Figure 1 Representation of a set by a classic BST (left) and an AC-Tree (right).

To represent a set with an AC-Tree, the intervals of its RS can be inserted in an empty BST, using the comparison of their lower bounds as ordering: [a, b] [c, d] iff a < c. Though, the creation of a balanced BST directly from an array of ordered elements can be performed in linear time by putting its middle element at the root node and inserting recursively the left part of the array in the left subtree and the right part in the right subtree. Figure 1 depicts the representation of set {1, 2, 3, 4, 7, 10, 11, 12, 15, 16, 18, 19} of size 12 as a balanced BST with 12 nodes in the left tree, corresponding to RS <[1, 4], [7, 7], [10, 12], [15, 16], [18, 19]>, which can be represented by an AC-Tree with 5 nodes as in the right tree. More formally, we define AC-Trees as follows:

Definition 2 (Adaptive Compact Tree (AC-Tree)). An Adaptive Compact Tree (AC-Tree) is a self-balancing binary search tree that contains at each node the bounds of an integer interval as well as the cardinal and the extrema of the set of integers represented by the subtree (the node itself included), such that intervals are ordered by and the ordered sequence of all the intervals of the tree is a RS.

In the following sections, we describe how the dis-contiguity of the intervals of an AC-Tree can be maintained to ensure the compactness of representation when single elements and whole intervals are added or removed.

2.1 Addition of a Single Element

When adding an element x to a set S represented by a RS, several cases have to be considered:

（1）if such that i.e. then the RS is left unchanged;

（2）else if such that bi = x 1 and ai+1 = x + 1, then intervals [ai, bi] and [ai+1, bi+1] must be merged and replaced by [ai, bi+1];

（3）else if such that bi = x 1 and, if i < k, ai+1 > x + 1, then interval [ai, bi] must be replaced by interval [ai, x];

（4）else if such that ai = x + 1 and, if i > 1, bi 1 < x 1, then interval [ai, bi] must be replaced by interval [x, bi];

（5）otherwise singleton interval [x, x] must be inserted: at the beginning of the RS if x < a1, else at the end of the RS if x > bk, otherwise such that bi < x < ai+1 and [x, x] must be inserted between [ai, bi] and [ai+1, bi+1].

So in case 2, a node must be withdrawn from the AC-Tree representing the RS, and in case 5, a node must be added, exactly as in classic BST. All other cases leave the structure of the tree unchanged, with a bound adjustment in cases 3 and 4. As with a self-balancing BST, rotations must be performed whenever a node becomes unbalanced because of an insertion or withdrawal below it.

We have implemented an AC-Tree library by modifying the Set module of the OCaml standard library [4], which provides a simple functional implementation of AVL trees where the height discrepancy between children of each node is bounded by 2 (instead of 1 for the original AVL trees). Algorithm 1 shows in pseudocode how the addition of a single element to an AC-Tree can be (functionally) implemented, using a few auxiliary functions:

- node(l, a, b, r) returns a new node with interval [a, b], left child l and right child r;
- is_empty(s) returns a boolean indicating if set s is empty;
- bounds(s) returns the bounds at the root of set s and children(s) returns its children;
- balance(l, a, b, r) [2]is similar to node but checks for a possible imbalance between l and r and performs one step of rebalancing (with rotations) if required;
- min(s) (resp. max(s)) returns the minimal (resp. maximal) element of set s in constant time, as AC-Trees maintain extrema at each node (see section 3);
- remove_get_min_inter(s) (resp. remove_get_max_inter(s)) returns the couple of the left-most (resp. right-most), and therefore minimal (resp. maximal), interval of set s and a new balanced set representing s without its minimal (resp. maximal) interval.

```
1  set add(int x, set s):
2    if is_empty(s) then return node(empty,x,x,empty) else
3    (a,b) := bounds(s)
4    (l,r) := children(s)
5    if x < a-1 then return balance(add(x,l),a,b,r)
6    else if x > b+1 then return balance(l,a,b,add(x,r))
7    else if x = a-1 then
8      if is_empty(l) then return node(empty,x,b,r)
9      else if x = max(l)+1 then
10       ((lmaxa, _), newl) := remove_get_max_inter(l)
11       return balance(newl,lmaxa,b,r)
12     else return node(l,x,b,r)
13   else if x = b+1 then
14     if is_empty(r) then return node(l,a,x,empty)
15     else if x = min(r)-1 then
16       ((_, rmaxb), newr) := remove_get_min_inter(r)
17       return balance(l,a,rmaxb,newr)
18     else return node(l,a,x,r)
19   else return s
```

Algorithm 1 Addition of a single elemen

The main difficulty occurs when the element to add may merge the interval at the node being visited and the preceding or succeeding one. On line 9 for example, we check if the interval at the root node must be merged with the preceding interval, and if it is the case, we compute and withdraw in logarithmic time the maximal interval of the left subtree l (line 10). The balance of the resulting new subtree must then be checked (line 11) as the height of l may have been reduced. Note that this can happen only once, as there are no recursive calls in this case, such that the time complexity of the addition of a single element remains O(log k) for an AC-Tree.

2 This function is provided by the original Set module of the OCaml standard library.

2.2 Removal of a Single Element

When removing an element x from a set S, the following cases must be considered:

(1)if such that and , then interval [ai, bi] must be removed;

(2)else if such that , then interval [ai, bi] must be replaced by [x+1, bi];

(3)else if such that , then interval [ai, bi] must be replaced by [ai, x-1];

(4)else if such that , then interval [ai, bi] must be replaced by intervals [ai, x - 1] and [x+1, bi];

(5)otherwise and the RS is left unchanged.

Note that in cases 2 and 3, only a bound adjustment must be performed, whereas in case 1 a singleton interval must be removed, and in case 4, an interval must be split in two parts.

```
1  set remove(int x, set s):
2    if is_empty(s) then return s else
3    (a,b) := bounds(s)
4    (l,r) := children(s)
5    if x < a then return balance(remove(x,l),a,b,r)
6    else if x > b then return balance(l,a,b,remove(x,r))
7    else if x = a then
8      if a = b then return merge(l,r)
9      else return node(l,a+1,b,r)
10   else if x = b then return node(l,a,b-1,r)
11   else
12     if h(l) <= h(r) then return balance(add_right(a,x-1,l),x+1,b,r)
13     else return balance(l,a,x-1,add_left(x+1,b,r))
```

Algorithm 2 Removal of a single element

Algorithm 2 shows how this operation can be implemented, using some auxiliary functions:

- merge(s1, s2) returns the set s1 ∪ s2, provided that max(s1) + 1 < min(s2) and that their height discrepancy is bounded by 2 (performing one step of rebalancing if required);
- add_right(a,b,s) (resp. add_left(a,b,s)) returns a new set where interval [a, b], with a 1 > max(s) (resp. b+1 < min(s)), has been added to set s as the right-most (resp. left-most) node (performing one step of rebalancing if required);
- h(s) returns the height of set s.

The algorithm is somewhat simpler than for addition, as there is no need to find successive intervals that could be merged. The most delicate case arises when an interval is split (case 4 corresponding to lines 12 to 13): a new interval [a, x 1] is inserted in logarithmic time in the shortest subtree of the visited node and the result is joined with interval [x + 1, b] at the root (performing one step of rebalancing if required). As for AVL, if a singleton interval must be removed (case 1 corresponding to line 8), a logarithmic amount of rotations may be performed in the worst case (one at each recursive call on line 5 or 6), such that the overall time complexity remains in O(log k).

2.3 Interval Operations

An interval [x, y] could be added to an AC-Tree by iterating the addition of one element from x to y with algorithm 1, but at a cost of O((y x + 1) log k). But an AC-Tree can benefit from its interval structure to support more efficient interval operations. The addition of an interval [x, y] to a RS is a generalization of the case of a single element where an arbitrary number of intervals can be merged instead of two at most:

1.if such that ai ≤ x and y ≤ bi, i.e. x S, then the RS is left unchanged;

2.else if such that x 1 ≤ bi and, if i > 1, bi 1 < x 1, and , such that aj ≤ y + 1, and, if j < k, y + 1 < aj+1, then intervals [ai, bi] to [aj, bj] must be merged and replaced by [min(ai, x), max(bj, y)];

3.otherwise interval [x, y] must be inserted in the RS as in case 5 of algorithm 1.

```
1  set add_interval(int x, int y, set s):
2    if is_empty(s) then return node(empty,x,y,empty) else
3    (a,b) := bounds(s)
4    (l,r) := children(s)
5    x := x = b+1 ? b : x
6    y := y = a-1 ? a : y
7    if y < a then return join(add_interval(x,y,l),a,b,r)
8    else if x > b then return join(l,a,b,add_interval(x,y,r))
9    else if x < a then
10     if y > b then
11       (lx,linter) := left_of(x-1,l)
12       (rinter,ry) := right_of(y+1,r)
13       if is_none(linter) then
14         if is_none(rinter) then return join(lx,x,y,ry)
15         else return join(lx,x,ub(rinter),ry)
16       else
17         if is_none(rinter) then return join(lx,lb(linter),y,ry)
18         else return join(lx,lb(linter),ub(rinter),ry)
19     else
20       (lx,linter) := left_of(x,l)
21       if is_none(linter) then
22         if is_empty(lx) then return add_left(x,b,r)
23         else
24           (lxa,lxb) := max_interval(lx)
25           if x = lxb+1 then return join(remove_max_inter(lx),lxa,b,r)
26           else return join(lx,x,b,r)
27       else return join(lx,lb(linter),b,r)
28   else
29     if y > b then
30       (rinter,ry) := right_of(y,r)
31       if is_none(rinter) then
32         if is_empty(ry) then return add_right(a,y,l)
33         else
34           (rya,ryb) := min_interval(ry)
35           if y = rya-1 then return join(l,a,ryb,remove_min_inter(ry))
36           else return join(l,a,y,ry)
37       else return join(l,a,ub(rinter),ry)
38     else return s
```

Algorithm 3 Addition of an interval

Algorithm 3 shows how the addition of an interval to an AC-Tree can be implemented, using some auxiliary functions:

- join(l, a, b, r) returns a new set equal to the union of sets l, r and interval [a, b], provided that max(l) + 1 < a and b < min(r) 1, without assumption on the height discrepancy between l and r;
- left_of(x,s) (resp. right_of(x,s)) returns in logarithmic time the couple of the subset of all elements of s strictly less (resp. greater) than x, without the possible interval [a, b] to which x belongs, and interval [a, b] if it exists as an optional value;
- is_none(opt) returns a boolean indicating if an optional value opt exists or not;
- lb(opt) (resp. ub(opt)) returns the lower (resp. upper) bound of optional interval opt;
- max_interval(s) (resp. min_interval(s)) returns the maximal (resp. minimal) interval of set s;
- expr1 ? expr2 : expr3 stands for the classic ternary conditional operator as in C. Note that function join may use a logarithmic amount of calls to balance, and function left_of (and right_of) can have the same behavior as an arbitrary number of intervals may be withdrawn from the tree.

This algorithm basically has the same structure as algorithm 1 for the first three cases (line 2 to 8), except that an arbitrary number of intervals can be removed on one side, so the join function must be called instead of balance. Besides, we adjust the bounds of the added interval on lines 5 and 6 to avoid a couple of tests in the contiguous case: if [x, y] is contiguous with [a, b] on its left side (i.e. y = a 1) then y can be incremented (i.e. y := a) without changing the result (and the same adjustment is performed symmetrically on x). The remaining of the algorithm consists in finding intervals i and j of case 2 if they exist and joining the left and right remaining subtrees to the merged intervals. When such intervals are found, the computation of

the left and right subtrees (by functions left_of and right_of) takes logarithmic time w.r.t. the height of the remaining subtrees l and r, with possibly the same amount of rotations. The overall complexity remains in O(log k), which is more efficient than the naive insertion in O((y x + 1) log |S|) for a classic AVL tree.

For the removal of an interval [x, y], one interval may be added to the AC-Tree at most (if such that ai < x and y < bi) and an arbitrary number of intervals can be withdrawn if [x, y] spans over multiple intervals. However, the algorithm is simpler than the one for the addition, but is not shown here because of lack of space.

2.4 Sets Operations

Set operations like intersection, union, difference or dis-jointness can also be efficiently implemented for AC-Trees by simultaneously traversing both sets to obtain a linear-time algorithm, rather than the O(k log k) one that would result from the processing of each interval of one set w.r.t. the other. We will here only consider intersection for the sake of brevity, but the performances of each of these operations are presented in section 4.1.3.

3 CARDINAL AND EXTREMA

In some applications, like CP, other operations such as obtaining the cardinal of a set or its extrema must also be performed efficiently. For a classic BST, computing the cardinal comes down to counting the number of nodes in the tree, which can be done in linear time w.r.t. the size of the set, and extrema can be returned in logarithmic time. However, these operations can be performed much more efficiently, i.e. in constant time, if the corresponding data are stored at each node, at the cost of a linear amount of extra storage: e.g. for a node with interval [a, b], left child l and right child r, the maximum is equal to the expression is_empty(r) ? b : max(r) and the cardinal to b-a+1+card(l)+card(r) (with auxiliary function card(s) returning the cardinal of s).

AC-Trees store at each node the minimal and maximal values as well as the cardinal of the subtree, which are easily maintained in constant time by all the algorithms corresponding to operations returning a new set. Furthermore, operations on AC-Trees can benefit from this direct access to extrema, used as an approximation of the set represented by the subtree rooted at each node to reduce computation times in simple cases: e.g. the remove function described in algorithm 2 can test whether y < min(s) || x > max(s) just after line 2 to immediately return s, and the inter function of algorithm 4 can test max(s1) < min(s2) || max(s2) < min(s1) after line 2 to immediately return empty.

4 EXPERIMENTS

We compare in this section the performances of set and interval operations on random sets with AVL trees and AC-Trees, implemented on top of the Set module of the OCaml standard library [4]. We also report our first experiments on the application of AC-Trees as a replacement for RSs to represent the finite domains of the FaCiLe CP solver [1]. All the experiments were carried out on an Intel Xeon at 3.4GHz with 16GB of RAM, running Debian GNU/Linux with kernel 3.16 and the OCaml 4.01.0 compiler. Execution times are given in seconds and obtained from the mean of 100 runs for each operation.

4.1 Random Sets

To compare the performances of AVL trees and AC-Trees on typical set and interval operations, we have randomly generated sets of a given cardinal n with a given number of holes k, because the complexity of operations on AC-Trees only depends on k. But the maximal number of holes depends on n (and is exactly n 1), so to be able to compare instances of various sizes on the same graph, our plots are given as a function of the “holes density” which is defined as the ratio of the actual number of holes over the maximal

number of holes, i.e. k/n 1, which ranges from 0 for a plain interval to 1 for sets made of dis-contiguous singletons.

On each figure, we show three graphs corresponding to different sizes (indicated by thousands of elements, e.g. “50K” means 50, 000 elements) for AVL trees (labelled AVL and pictured with a dashed line) and AC-Trees (labelled ACT, with a plain line).

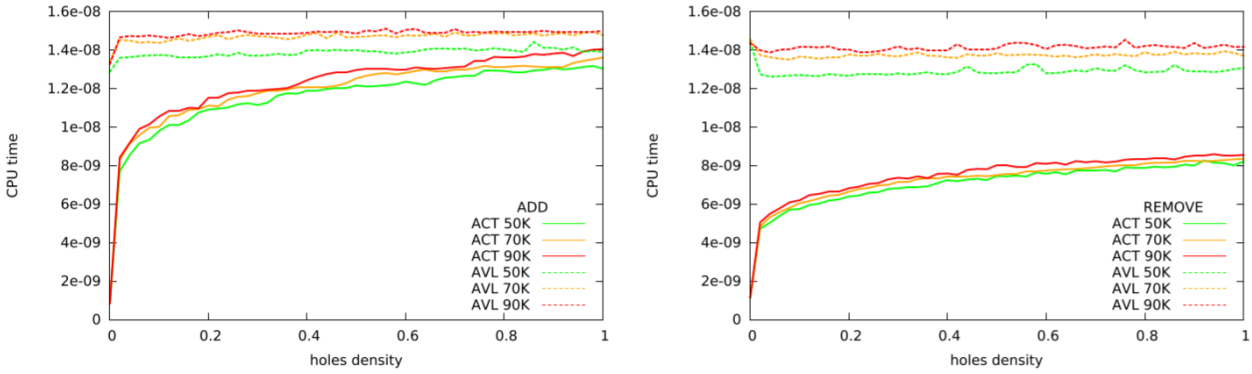

Figure 2 Addition and removal of a single element.

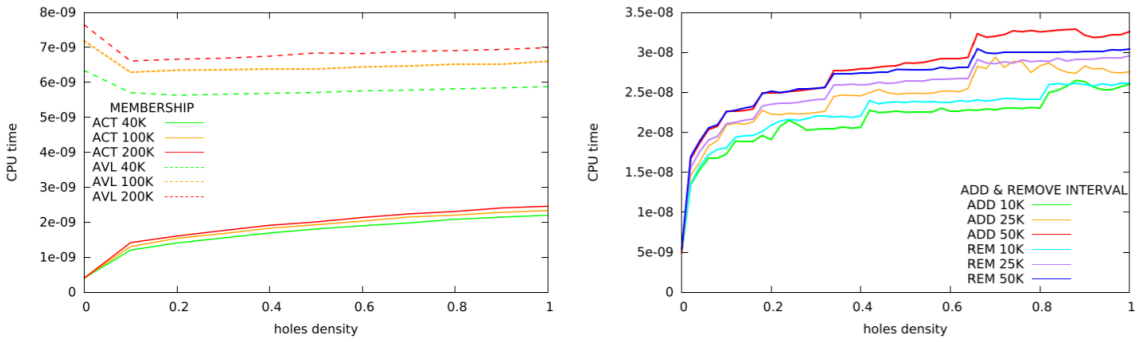

Figure 3 Membership of an element and interval operations.

4.1.1 Operations on Single Elements

As shown on figure 2, addition and removal of a single element is always faster with AC-Trees than with AVL trees, especially when the holes density is low, as expected. For densities close to one, the performances of both data structures are similar for addition, as an AC-Tree made of dis-contiguous singletons has the same structure as an AVL tree.

However, for removal, AC-Trees are much faster than AVL trees for all densities, as the extrema maintained at each node allow to decide efficiently whether the subtrees must be explored or not: if the element to remove is less than the minimal element or greater than the maximal one, the corresponding recursive call immediately returns.

Membership, shown on the left graph of figure 3, exhibits the same properties but is even faster (at least three times than AVL trees). Finally, these graphs confirm the expected logarithmic complexity of single element operations w.r.t. the number of holes for AC-Trees.

4.1.2 Interval Operations

As mentioned in section 2.3, AC-Trees support interval operations by design, such that the addition (union) or removal (difference) of an interval is orders of magnitude faster than with AVL trees (1000 times faster at least for the instances shown on figure 3). Indeed, for the latter, interval operations come down to the iteration of addition or removal for each element of the considered interval [x, y], i.e. O((y x+1) log n) operations, whereas the time complexity remains O(log k) for AC-Trees, as confirmed by the graphs of figure 3.

4.1.3 Set Operations

The performances of set operations like intersection, union, difference and subset on AC-Trees reported on figure 4 do not outperform so clearly AVL trees: they are much faster for low holes densities and become similar or slower above 0.6 to 0.8 (depending on the operation considered). Indeed, when the number of nodes of an AC-Tree gets closer to the one of the corresponding AVL tree, the overhead induced by interval merging or splitting to maintain dis-contiguity becomes too costly compared to single nodes insertion and removal. These graphs also confirm the expected linear time complexity of these operations w.r.t. k (and therefore to the holes density for a given cardinal).

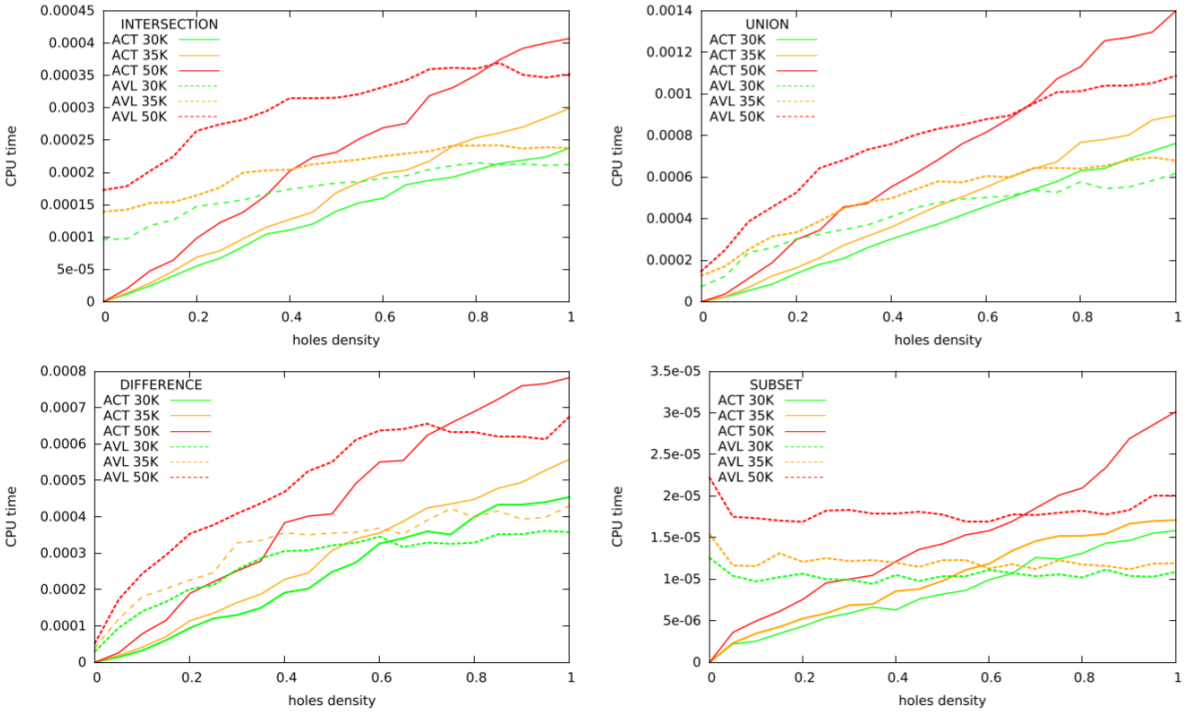

Figure 4 Intersection, union, difference and subset.

4.2 Constraint Programming

We have used AC-Trees to replace the RSs[3] that represent finite domains in the FaCiLe CP solver [1] and solved the same bioinformatics problem[4] of RNA motifs detection than with the one presented in [7]. The complete problem is based on the genome of bacteria E. coli made of a sequence of more than 4.6 × 106 nucleotides, and the CP model uses variables that are indices of this sequence, therefore with a domain size equal to its length.

Figure 5 shows the comparison of execution time and maximal memory usage between RSs (in green/light grey) and AC-Trees (in red/dark grey) to solve subproblems that only considers the first n (noted as "Size" on the graphs) nucleotides of the genome, starting from 16, 000 (which is enough to find a solution) to 1.5 × 106. With AC-Trees, FaCiLe is able to solve the problem up to 2.5 times faster than with RSs, though the memory consumption can be twice as much (but only 50% more for the largest instance[5]). However, with enough RAM, the complete instance can be solved in about 1h45.

3 Note that the RSs of FaCiLe maintain their cardinals and extrema, but only at the root node.

4 The ILOG Solver and Naxos source codes, as well as the corresponding data, can be found at http://di.uoa.gr/~pothitos/setn2010/HairpinDetection.

5 OCaml uses a garbage collector (like most functional languages), so the memory usage is less predictable than with manual memory management.

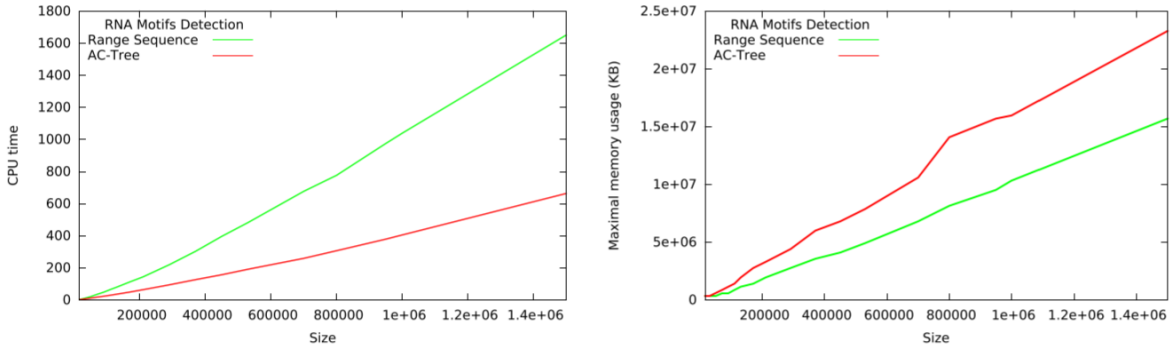

Figure 5 Execution time and maximal memory usage to solve the RNA motifs detection problem.

5 CONCLUSION AND FURTHER WORKS

We have presented a new data structure, AC-Trees, similar to self-balancing BSTs but holding dis-contiguous intervals at each node, in order to reduce the storage of integers sets with few holes and to improve standard and interval set operations, which must be efficiently supported in applications like CP solvers. For integers sets with k holes, AC-Trees need only O(k) storage, and single element and interval operations can be performed in O(log k) time.

Moreover, cardinals and extrema are also stored at each node to further optimize set operations, at the cost of a linear extra storage. First experiments show that AC-Trees outperform classic AVL trees on random sets in most cases, and that an application to a bioinformatics problem can be solved twice faster when the finite domains of a CP solver are implemented with AC-Trees instead of standard range sequences.

However, further works need to be done to assess more precisely the advantages of ACTrees. The benefit of maintaining extrema should be analysed by comparing with AC-Trees devoid of these extra data and with AVLs augmented with them. Other self-balancing BSTs like Red-Black trees could also be used instead of AVL trees. Eventually, classic self-balancing BSTs should also be compared to AC-Trees to represent finite domains in a CP solver.

ACKNOWLEDGMENTS

[1]This article is supported by diversified investment fund of Tianjin under Grant 21JCQNJC00790.

REFERENCES

Barnier, N., & Brisset, P. (2001). FaCiLe: A Functional Constraint Library. In CICLOPS – Colloquium on Implementation of Constraint and Logic Programming Systems, CP'01 Workshop, Paphos, Cyprus, December 2001.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press. Chapter 14, pp. 339–355.

Knuth, D. E. (1998). The Art of Computer Programming: Sorting and Searching (Vol. 3). Pearson Education.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., & Vouillon, J. (2013). The OCaml System (Release 4.01): Documentation and User's Manual. Institut National de Recherche en Informatique et en Automatique.

Pfaff, B. (2004). Performance Analysis of BSTs in System Software. ACM SIGMETRICS Performance Evaluation Review, 32(1), 410–411.

Pothitos, N. (2013). Naxos Solver. http://di.uoa.gr/ ˜pothitos/naxos.

Pothitos, N., & Stamatopoulos, P. (2010). Flexible Management of Large-Scale Integer Domains in CSPs. In Artificial Intelligence: Theories, Models and Applications: 6th Hellenic Conference on AI, SETN 2010, Athens, Greece, pp. 405–410. Springer.

Schulte, C., & Carlsson, M. (2006). Finite Domain Constraint Programming Systems. In Handbook of Constraint Programming, Chapter 14, pp. 495–526. Elsevier.