Int. J. Appl. Math. Comput. Sci., 2020, Vol. 30, No. 4, 717–731
DOI: 10.34768/amcs-2020-0053

ASA–GRAPHS FOR EFFICIENT DATA REPRESENTATION AND PROCESSING

ADRIAN HORZYK [a,*], DANIEL BULANDA [a], JANUSZ A. STARZYK [b,c]

[a]Department of Biocybernetics and Biomedical Engineering
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: horzyk@agh.edu.pl, daniel@bulanda.net

[b]Faculty of Applied Computer Science
University of Information Technology and Management in Rzeszów
ul. Sucharskiego 2, 35-225 Rzeszów, Poland

[c]School of Electrical Engineering and Computer Science
Ohio University
Athens, OH 45701, USA
e-mail: starzykj@gmail.com

Fast discovering of various relationships in data is an important feature of modern data mining, cognitive, knowledge-based, and explainable AI systems, including deep neural networks. The ability to represent a rich set of relationships between stored data and objects is essential for fast inferences, finding associations, representing knowledge, and extracting useful patterns or other pieces of information. This paper introduces self-balancing, aggregating, and sorting ASA-graphs for efficient data representation in various data structures, databases, and data mining systems. These graphs are smaller and use more efficient algorithms for searching, inserting, and removing data than the most commonly used self-balancing trees. ASA-graphs also automatically aggregate and count all duplicates of values and represent them by the same nodes, connecting them in order, and simultaneously providing very fast data access based on a binary search tree approach. The proposed ASA-graph structure combines the advantages of sorted lists, binary search trees, B-trees, and B+trees, eliminating their weaknesses. Our experiments proved that the ASA-graphs outperform many commonly used self-balancing trees.

Keywords: self-balancing trees, self-sorting trees, self-aggregating data structures, associative structures, graphs, data access efficiency, representation of relationships.

1. Introduction

With the rapidly growing amount of data, we need to search for new algorithms and data structures that will allow for efficient storage and fast data processing. New data structures should also represent frequently used data relationships to avoid looking for these relationships in many nested loops. Known relationships allow us to find related data and achieve other search objectives quickly.

For data mining and knowledge discovery, we need efficient structures to find not only data and their frequent patterns, but also their relationships quickly. The most commonly used structures are arrays, matrices (n-dimensional arrays), lists, queues, stacks, heaps, hashing structures, trees, and graphs of various kinds (Cormen *et al.*, 2009). Every structure has its strengths and weaknesses, which determine its use in various applications according to the most frequent operations that will be processed using them. Data structures are typically efficient only for a subset of operations, e.g., search, while for the other operations, e.g., insert or remove, are less efficient depending on the stored data. A satisfying compromise is desirable (Baran, 2018).

Another important factor that differentiates various data structures is the number and kind of represented relationships between stored data. For example, when data are sorted, the subsequent array fields or list

[*]Corresponding author

nodes define the neighborhood relationship of the stored values. Sorted data speed up the calculation of sums, averages, medians, minima, maxima, count up or remove duplicates, filter data according to some conditions or perform data mining computations, searching for similarities, differences, clusters, frequent patterns, associative rules, etc. In all these operations, we need to have fast access to data and relationships (Lewicki and Pancerz, 2020). Today, we have many complex neural network structures which we want to work faster to solve the problems of our civilisation (Wieczorek *et al.*, 2020). On the other hand, we have different complex neural network solutions which can work faster when appropriately associating sensors with neural networks (Woźniak *et al.*, 2020).

This paper introduces a new type of aggregating and sorting associative graph structures (ASA-graphs). ASA-graphs store data in sorted order, allowing for very fast data access, search, insertion, removal, updating, and representing information about a number of duplicates, aggregating their representations and stable sorting of their dependencies, making often used relationships instantly accessible. They also improve the performance and storage capacity of commonly used B-trees (Bayer and McCreight, 1972) in datasets with many duplicated values. This structure not only represents data consistently (automatically aggregating and counting duplicates) but also carries important relationships between stored data like order and counts of duplicates, which accelerates filtering and calculations of sums, averages, and medians. Hence, we can use this associative structure to represent frequent data patterns together with frequent relationship patterns, which might be an important goal of various knowledge-based search and relationship mining algorithms.

The ASA-graphs link stored values in sorted order similarly to B+trees (Comer, 1979; Cormen *et al.*, 2009; Wu *et al.*, 2010) that are commonly used as the backbone data structure of database management systems (DBMSs), but unlike B+trees the proposed structure stores data in all tree nodes spanned over this graph, not wasting either memory for the guidepost nodes or time for targeting values stored only in leaves of B+trees. The ASA-graph is a combination of a sorted list and a self-balancing search tree. Therefore, all operations take at most the logarithmic time as a function of the number of unique (not all) keys in the collection. When the collection contains many duplicates, the computational complexity of the operations on ASA-graphs is independent of the number of stored data and can be processed in constant time, competing with the fastest self-balancing trees and hash maps. The comparisons of ASA-graphs with these structures are presented in this paper.

The contribution of this paper is the presentation of a new graph structure that combines, expands, and improves the other self-balancing tree structures, allows for the efficient representation of the cardinality of duplicates, which can accelerate many operations. The unique combination of the sorted list woven into the tree structure creates the graph structure whose elements can be searched in different ways, taking advantage of both a sorted order of elements in the list and logarithmic search using the balanced search tree. This combination also enabled us to develop new, more efficient insert and remove operations using both the search tree and sorted list structures. The aggregated representation of identical values (duplicates) in the collection allows for the quicker association of objects defined by the same keys in databases, and the connections to neighbor elements expand this quick search also to the objects defined by similar (not only the same) values.

2. Data tree structures for efficient search

In computer science, we use various data structures like hash maps (Mehta and Sahni, 2004; Sedgewick and Wayne, 2011), binary search trees (Cormen *et al.*, 2009; Hibbard, 1962), Red-Black-trees (Chen and Schott, 1996; Guibas and Sedgewick, 1978; Sedgewick and Wayne, 2011; Sharma *et al.*, 2018), AVL-trees (Adel'son-Vel'skii and Landis, 1962), B-trees (Bayer and McCreight, 1972; Graefe, 2011), B+trees (Comer, 1979; Cormen *et al.*, 2009; Wu *et al.*, 2010), etc., which accelerate search and other operations. There are many variants of B-trees: B*-trees (Sagiv, 1986), B**-trees (Toptsis, 1993), UB-trees (Comer, 1979), BUB-trees (Fenk, 2002), R-trees (Guttman, 1984), Bx-trees (Jensen *et al.*, 2004), (Dan, 2007), Tango-trees (Demaine *et al.*, 2007), and WAVL-trees (Haeupler *et al.*, 2015).

Binary search trees (BSTs) (Cormen *et al.*, 2009; Hibbard, 1962) are widely used in practice. This data structure is a rooted binary tree wherein each node stores a key (and optionally a value). Apart from the leaves, every node has two pointers to the left and right subtrees. The key in a node must be greater than or equal to any key stored in the left subtree, and less than or equal to any key stored in the right subtree. One of the reasons for the popularity of this structure is simplicity. Because BST are not self-balancing, in typical situations, the expected height of the tree is about $\sqrt{n}$ (where n is a number of keys), which grows much faster than $\log n$. In the worst-case, the time complexity of the search operation is of $O(n)$ because a tree can degenerate and become a list.

Adel'son-Vel'skii and Landis (1962) invented self-balancing binary search tree (AVL-tree) that achieve the $O(\log n)$ computational complexity for insert, search, and remove. The most significant constraint of this tree is that the heights of the two-child subtrees of any node differ by at most one – if not, rebalancing is done to restore the balanced structure. Rebalancing is performed using one or more subtree rotations.

A popular tree-like structure is the Red-Black-tree (RBT) invented by Guibas and Sedgewick (1978), who derived the RBT from the symmetric binary B-tree, also known as 2-3-4 tree. The RBT is a self-balancing binary search tree with one extra bit per node interpreted as the red or black color. The color bit helps to ensure that the tree remains balanced during deletions or insertions (Sedgewick and Wayne, 2011). The balancing of the tree is not perfect but still guarantees worst-case complexity $O(\log\ n)$ for insertion, searching, and deletion. The RBT is similar to a B-tree of order 4 (Knuth's definition of order: the maximum number of children (Knuth, 1998)). Each node stores at most 3 keys, only one key-node marked as black, with an optional red key-nodes before and after the black one.

Another popular solution is a hash map (also called a hash table) (cf. Mehta and Sahni, 2004; Sedgewick and Wayne, 2011) that implements an associative array of abstract data types, mapping keys to values. Hash maps use hash functions to compute an index (hash code) into an array of buckets (slots), where the desired value can be found in constant time if the number of values in buckets is not too big.

B-trees are the simplest self-balancing binary search trees invented in Boeing Research Labs introduced by Bayer and McCreight (1972). They are a generalization of binary search trees with nodes that can have more than two children. The order of this tree, according to the Knuth definition (Knuth, 1998), is defined as the maximal number of node children as used in this paper. Another popular measure of the capacity of B-tree nodes is the CLRS degree (often denoted as t) (Cormen *et al.*, 2009) where the possible number of node's children k, except the root, is in the range of t and $2t$, i.e., $t \leq k \leq 2t$.

One of the first mentions of B+trees came from Comer (1979). B+trees modify a B-tree structure to store all keys in leaf nodes only and order these keys, adding extra connections, and thus removing one of the disadvantages of B-trees. Such an upgraded structure is commonly used in file systems stored on disks and databases. Because all the keys are in leaves, internal nodes are used to navigate through this structure only. It simplifies disk operations but produces additional signpost nodes (navigating the search algorithm to the leaves), and therefore these trees grow faster than B-trees, and the search always requires to go through all nodes on the path from the root to a leaf. Every leaf node stores a pointer to the next leaf if it exists so that we can move over the leaves in sorted order. A big advantage over B-trees is that B+trees store the information about the relationship of order between data. These structures guarantee logarithmic time complexity for typical operations at the cost of more memory usage and longer time of search and insertion on average.

WAVL-trees, invented by Haeupler *et al.* (2015), are designed to combine some of the best properties of both AVL-trees and Red-Black-trees. Their height is at most $1.44\log_2 n$ and only a constant number of tree rotations is used to balance themselves after some insert or remove operations. They can be used in various applications instead of Red-Black-Trees or AVL-Trees.

AVL-trees are very popular, especially for search-intensive tasks, because they are more strictly balanced (Pfaff, 2004). Applications of Red-Black-trees are varied, including Completely Fair Scheduler in Linux kernel or Java HashMap, as a better alternative to LinkedList to store different elements with colliding hash codes. In the work of Sharma *et al.* (2018), the Red-Black-trees make routing in partitioned networks feasible. The distributed priority tree-based routing protocol (DPTR) utilizes features provided by the trees to manage communication between nodes in different networks. Hash maps are widely used in many solutions, particularly for associative arrays, database indexing, caches, and datasets.

B-trees are used in column-oriented storage formats for query processing in relational databases and warehouses (Graefe, 2011). B-trees can also be used in data warehouses (Sun *et al.*, 2016), where a B-tree-based tool enhances metadata management to overcome a bottleneck in the performance of metadata operations. This tool creates a B-tree based overlay network, supplying inefficient lookup operations. Shen *et al.* (2018) used the Locally-Sensitive B-tree (LSB-tree) for anomaly saliency determination by providing information about the nearest neighbors in the LSB-forest. Kim *et al.* (2018) adapted B-trees called cache-line friendly B-trees (clfB-trees) to achieve better performance than B-trees. B+trees find use in file systems and databases (Wu *et al.*, 2010). Cloud computing is an area where B-trees and B+trees are also deployed successfully. The feature described by Fan *et al.* (2018) is a perfect example of the topical Internet of Things (IoT) adaptation in the automotive industry where the designer settled three main goals of his project: user privacy, security, and efficiency.

Real data collections usually contain many duplicates that are not aggregated and counted, e.g., in B-trees or original B+trees since they assume storing the unique keys. To prove that duplicates are common in many datasets used in data mining related tasks, we downloaded 85 randomly selected datasets from UCI ML Repository and Kaggle containing from 57 to 2299651 records to obtain some statistics about duplicates. The mean value of the number of duplicates for the tested datasets was 71%, with the number of duplicates for each dataset varying from 7% to 99%. This experiment has shown that duplicates are very common in data from various domains. To benefit from the aggregations of duplicates in data science, we introduce a special data structure in the following section.

3. ASA-graphs

In this paper, we propose aggregative sorting associative graphs (ASA-graphs), cf. Fig. 1. They are a combination of a sorted list of elements and a third-order self-balancing search tree (similar to a B-tree) spanned over all stored elements in this list. Every element stored in these graphs can be reached using at least two paths, namely, using the connections between elements of the bidirectionally sorted list or the connections between nodes storing elements of the spanned self-balancing tree.

Definition 1. (*ASA-graph element*) Each element of ASA-graphs stores (Fig. 1):

- a unique key of the represented data collection,
- the number of this key duplicates in the collection,
- two pointers or indices to the previous and next elements in the sorted order,
- the list of indices or pointers to the objects sharing this key.

Elements of ASA graphs are combined into a sorted list and self-balancing search tree, defined next.

Definition 2. (*ASA-graph bidirectional sorted list*) A bidirectional sorted list used in ASA-graphs is a classic sorted list storing elements which contain unique keys and two pointers or indices to the previous and next elements in the sorted order (Fig. 1).

To complete the definition of ASA-graphs, we must define its self-balancing search tree storing all elements of the ASA-graph in its nodes.

Definition 3. (*ASA-graph self-balancing search tree*) A self-balancing search tree of the 3rd order of the ASA-graph satisfies the following properties (Fig. 1):

1. Every node contains one or two elements.
2. Every non-leaf node stores one child more than the number of the stored elements.
3. All leaves always are in the last level of this tree.
4. Keys stored in the leftmost subtree of each node are smaller the key(s) stored in this node.
5. Keys stored in the rightmost subtree of each node are bigger the key(s) stored in this node.
6. Keys stored in the middle subtree of the node, if this subtree exists, are larger than the smallest key and smaller than the largest key stored in this node.

Based on the previous three definitions, we can define ASA-graphs.

Definition 4. (*ASA-graph*) An ASA-graph is a hybrid structure consisting of special nodes storing all elements representing unique keys of the collection (Definition 1) that are connected in sorted order using bidirectional connections of the sorted list (Definition 2) and also connected and accessible through the connections of the self-balancing search tree (Definition 3) spanned over all these elements (Fig. 1).

There are alternative paths between elements stored in ASA-graphs because both the connections of a bidirectional list and the connections of the self-balancing search tree can be used to move between every two elements. ASA-graphs are a hybrid graph structure because it encompasses two kinds of connections between stored elements (graph nodes), which are connected directly using bidirectional sorting connections and extra, higher-level node-connections between ASA-graph nodes containing one or two elements (Fig. 1).

3.1. ASA-graph properties.

ASA-graphs aggregate and count duplicates, sort them bidirectionally, and accelerate the search, insert, remove, and other operations thanks to the use of this tree that stores elements in a similar way to the keys stored by self-balancing B-trees. Unlike B+trees, the spanned trees of the ASA-graphs do not use extra signpost nodes. They are usually smaller and faster than B-trees and B+trees thanks to the built-in aggregating and counting mechanisms of duplicates, which substantially reduce the size and height of such spanned trees in comparison with B-trees, B+trees, and other similarly used trees.

The ASA-graphs allow for searching for the keys starting from the root and going along the edges, benefiting from the logarithmic search time of the number of all unique keys stored in this tree, or moving along the sorted collection of elements using additional connections between elements. This design ensures the efficiency of various operations. ASA-graphs are associative because they associate neighboring keys, aggregate and count duplicates, and connect keys to similar objects sharing these keys. Associative means related to something with the defined strength of the relationship usually expressed by the connection weights between graph nodes that define related elements. Relationships may be of various types, e.g., one-to-one, one-to-many, or many-to-many. Associations can be of various kinds and have varying strengths as introduced by Horzyk (2013), e.g., associations come from similarities, sequences, proximity, inversion, or inclusion. The more associations between data and objects we store, the more pieces of information are directly accessible in the data structure, neural network, or database.

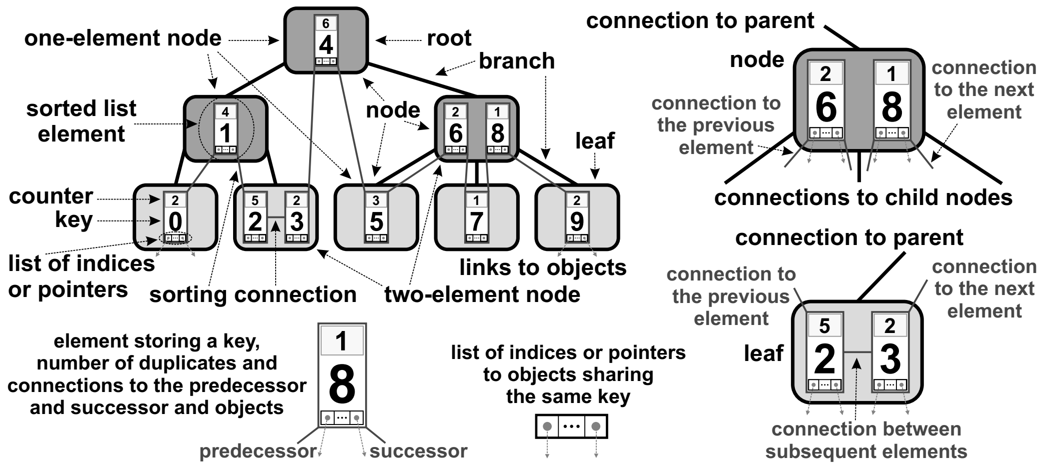
one-element node
root
branch
sorted list element
node
leaf
counter
key
list of indices or pointers
sorting connection
two-element node
links to objects
element storing a key, number of duplicates and connections to the predecessor and successor and objects
predecessor
successor
list of indices or pointers to objects sharing the same key
connection to parent
node
connection to the previous element
connection to the next element
connections to child nodes
connection to parent
connection to the previous element
connection to the next element
leaf
connection between subsequent elements

Fig. 1. ASA-graph structure consisting of nodes, branches, elements, keys, counters, connections, and links, combining a sorted list with a self-balancing search tree spanned over all elements.

In databases, where objects (entities) are defined by several attributes, every attribute can be represented by such an ASA-graph, accelerating various operations and indexation. ASA-graphs have been designed to support the other associative structures and neural networks like AGDS (Horzyk, 2018), APNN (Horzyk and Starzyk, 2018), DASNG (Horzyk, 2017), ANAKG (Horzyk, 2015), and associative semantic memories (Horzyk *et al.*, 2017) to accelerate various search, data mining, and knowledge exploration operations.

Assuming that h is the height of the spanning tree, where $h = 0$ for the root of the spanning tree, ASA-graphs have the following properties:

1. The minimal number of unique elements in the graph $n_{\min}$ equals 2^h.

2. The maximal number of unique elements in the graph $n_{\max}$ equals $3^h - 1$.

In the search, insert, remove, and the other operations on the keys, we use both a sorted list of elements and a self-balancing search tree structure (called a tree in short in the following descriptions). The algorithms of these operations are presented in the subsections that follow.

Algorithm 1. ASA-graph search.

```
Require: $r, k$ {$r$, root of tree; $k$, key sought}
1: $n = r$ {$n$, current node}
2: while true do
3:   if contains($r, k$) then
4:     return el {el, element storing key $k$}
5:   else if is_leaf($n$) then
6:     return null
7:   else if $k <$ leftmost_key($n$) then
8:     $n =$ leftmost_child($n$)
9:   else if $k >$ rightmost_key($n$) then
10:    $n =$ rightmost_child($n$)
11:  else
12:    $n =$ middle_child($n$)
13:  end if
14: end while
```

3.2. Algorithm 1: Search operation in ASA-graphs.

Step 1. Start from the root of the tree.

Step 2. Check if the current node stores the searched key. If so, return this element and finish this operation.

Step 3. If the current node is a leaf, return no element and finish this operation.

Step 4. If the searched key is smaller than the key of the leftmost element, go to the leftmost child; otherwise, if the searched key is larger than the key of the rightmost element, go to the rightmost child; otherwise, go to the middle child node.

Step 5. Go to Step 2.

When searching for the previous or the next key in order, only ASA-graphs, B+trees, and bidirectionally sorted lists allow for quick, constant complexity access to it, while B-trees, AVL-trees, BST-trees and Wavl-trees have at least logarithmic complexity because it is necessary to go along the edges of these trees. Searching for such neighbor values in hash maps is also difficult because neighbor values are not directly or indirectly pointed.

3.3. Algorithm 2: Insert operation in ASA-graphs.

Step 1. Start from the root of the tree.

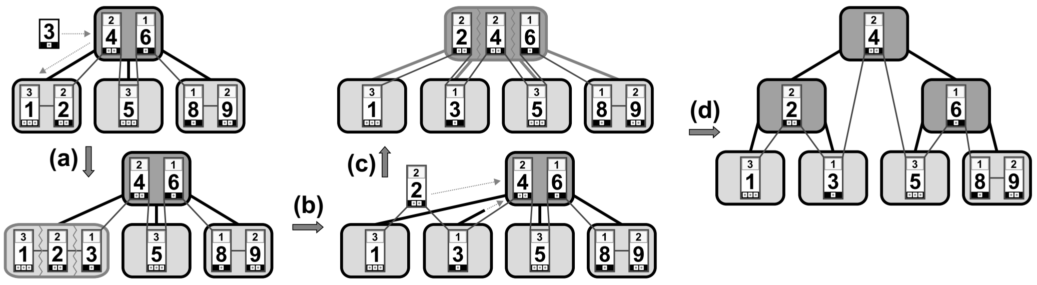

Fig. 2. ASA-graph insert operation of a key that requires splitting the overfilled nodes.

Step 2. Check if the current node already stores the element with the same key as the inserted one. If so, increase the element's key counter, add the value to the list of values associated with the key, and finish this operation.

Step 3. If the current node is not a leaf, go to one of the children by the following rules: If the inserted key is smaller than the key of the leftmost element, go to the leftmost child; otherwise, if the inserted key is larger than the key of the rightmost element, go to the rightmost child; otherwise, go to the middle child node. Go to Step 2.

Step 4. Insert a new element in sorted order to this leaf, initialize its counter to one (Fig. 2(a)). Set its previous and next element pointers accordingly to the elements between which it has been inserted, and update the previous and next element pointers of those elements to point out this new element, updating the bidirectional list of elements accordingly.

Step 5. If this node contains at most two elements, finish this operation.

Step 6. Split the overfilled node (containing more than 2 elements) in the following way (Fig. 2):

(a) Create a new node and the right element together with two rightmost children (when this node is not a leaf) is passed to the newly created node, and the left element together with two leftmost children (when this node is not a leaf) is kept in the split node (Fig. 2(b)).

(b) Forward the middle element of the overfilled node to the parent if it exists (Fig. 2(b)). Insert the middle element to the parent node at the position of the child that forwarded the element (Fig. 2(c)). Insert a new child connection from this parent to the new node right after the connection to this child that forwarded the element (Fig. 2(c)). Next, go to the parent node and go to Step 5.

Algorithm 2. ASA-graph insert.

```
Require: $r, k, v$ {$r$, root of tree; $k$, key to insert; $v$, value to insert}
1:  $n = r$ {$n$, current node}
2:  while true do
3:    if contains$(r, k)$ then
4:      increment_counter$(el)$ {$el$, element storing key $k$}
5:      add_value$(el, v)$
6:      return
7:    else if is_leaf$(n)$ then
8:      $ne$ = create_element$(k, v)$
9:      connect_neighboring_elements$(ne)$
10:     while elements_count$(n) <= 2$ do
11:       if has_parent$(n)$ then
12:         split$(n)$
13:       else
14:         create_parent$(n)$
15:         split$(n)$
16:       end if
17:       $n$ = parent$(n)$
18:     end while
19:     return
20:   else if $k <$ leftmost_key$(n)$ then
21:     $n$ = leftmost_child$(n)$
22:   else if $k >$ rightmost_key$(n)$ then
23:     $n$ = rightmost_child$(n)$
24:   else
25:     $n$ = middle_child$(n)$
26:   end if
27: end while
```

(c) If the parent does not exist, create a new parent node and connect it to the split node and the new node, establishing them as its children Fig. 2(d)). This new parent node becomes a new root of the tree as well. Next, finish this operation.

During the insert operation, the connections to the previous and next elements are preserved, so the bidirectional list included in the ASA-graph is always up-to-date.

3.4. Algorithm 3: Remove operation in ASA-graphs.

1. SEARCH: Use the search operation to find an element containing the key intended for removal. If this key is not found, finish this operation with no effect.

2. DECREMENT: If the counter of the element storing the removed key is greater than one, decrement this counter, remove the link to the bound object, and finish the remove operation.

3. REMOVE: If the element storing the removed key is in a leaf, remove this element from this leaf, switch pointers from its predecessor and successor to point themselves as direct neighbors. Next, if this leaf is not empty, finish the remove operation (Fig. 3(A)), otherwise go to Step 5 (Fig. 3(B)) to fill or remove this empty leaf.

4. REPLACE: The element storing the removed key is a non-leaf node that must be replaced by the previous or next element stored in a leaf. If the previous or next leaf contains more than one element, replace the removed element in the non-leaf node by the element from the leaf containing more than one element, and finish the remove operation (Fig. 3(Ca)), otherwise continue to choose the element of the leaf which parent contains more elements. Next, replace the removed element by the selected leaf element (Fig. 3(Cb)), which produces the empty leaf that will be filled or removed in Step 5.

5. FILL EMPTY LEAF: If one of the nearest siblings of the empty leaf contains more than one element, replace its ancestor element from the empty leaf side by the closest element from this sibling and shift this ancestor element to the empty node (Fig. 3(D)), and finish the removal operation.

6. REMOVE LEAF: If the parent of the empty leaf stores more than one element, move the nearest parent element to the closest sibling of the empty leaf, remove the empty leaf (Fig. 3(E)), and finish this operation.

7. COLLAPSE: When both the parent of the empty leaf and the only sibling store only single elements, move the element from this sibling to the parent and remove both children of this parent (Fig. 3(F)).

8. REBALANCE: If one of the sibling nodes of the reduced subtree root contains more than one element, shift its closest element to the parent replacing the closest element in it, create a new child and move down the replaced element from the parent to this new child. Connect this new child to the reduced subtree and switch the closest child of the sibling to this new child as well (Fig. 3(G)) and finish this operation.

9. REBALANCE: If all sibling nodes of the reduced subtree root contain only one element, move the parent element to the closest one-element sibling of the reduced subtree root and switch this reduced subtree root from its parent to this sibling. (Fig. 3(H)). Next, finish this operation if the parent node still contains one element. If not, in the parent node of the empty node (if exists) swap the pointer to

Algorithm 3. ASA-graph remove.

```
Require: $r, k$ {$r$, root of tree; $k$, key to remove}
n = r {n, current node}
while true do
  if contains(r, k) then
    if counter(el) > 1 then
      el {el, element storing key k}
      decrement_counter(el)
      remove_value(el)
      return
    else if is_leaf(n) then
      disconnect_neighbours(el)
      remove_value(el)
      if is_not_empty(n) then
        return
      else
        balance_empty_leaf(n) {see Steps 5–9 and
          Fig. 3 for details}
      end if
    else
      rel = find_candidate_to_replace(el)
      replace_empty_element(el, rel)
      if is_not_empty(node(rel)) then
        return
      else
        balance_empty_leaf(node(rel)) {see Steps
          5–9 and Fig. 3 for details}
      end if
    end if
  else if is_leaf(n) then
    return
  else if k < leftmost_key(n) then
    n = leftmost_child(n)
  else if k > rightmost_key(n) then
    n = rightmost_child(n)
  else
    n = middle_child(n)
  end if
end while
```

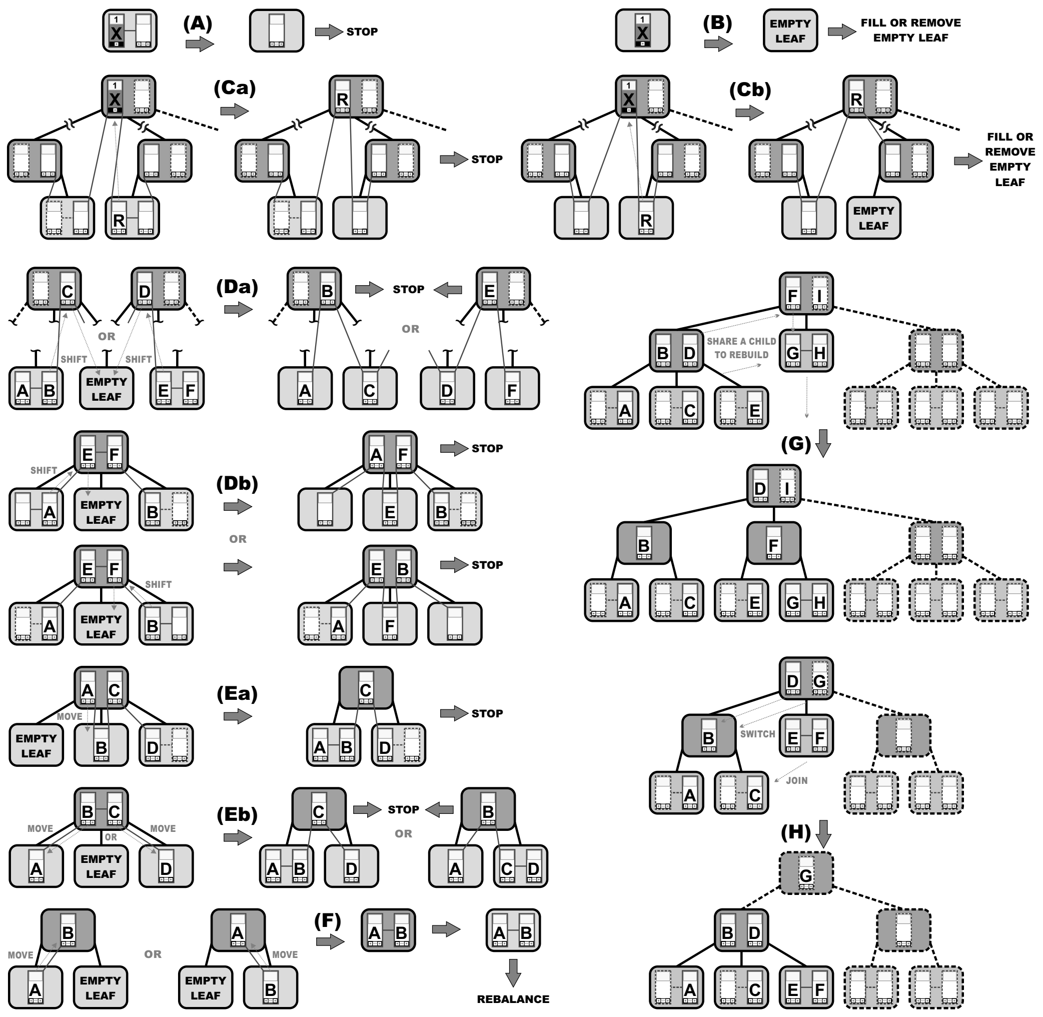

Fig. 3. ASA-graph remove operation: removing the element from the leaf or non-leaf node (A)–(C), filling or collapsing the empty leaf (D)–(F), rebalancing the tree to retrieve leaves to be at the last layer only (G)–(H).

the empty parent node with the pointer to the newly merged node, remove the empty node and perform rebalancing for the root of the reduced subtree by going to Step 8 until the main root of the tree is not achieved. If the main root is achieved, set the main root pointer as the pointer to the newly merged node and remove the empty old main root node and finish this operation. If the parent node of the empty node does not exist, set the newly merged node as the main root of the tree, remove the empty node, and finish this operation.

The insert and remove operations of ASA-graphs are defined in such a way that the connections (represented by the pointers or indices) to the elements representing the neighbor keys are updated automatically in constant time. Hence, the order of all keys in these graphs is always guaranteed with a constant computational complexity cost. This is possible thanks to the movement of pointers to the next and previous elements inside the element structure (Fig. 1), so no sorting operations on these graphs are necessary. That is why we say that ASA-graphs are self-sorting structures.

The remove operation is very challenging because it requires to keep elements in order and rebalance the tree spanned over the remaining elements.

3.5. Computational complexity of ASA-graphs.

Time computational complexities (called in short time complexities) of the search, insert, and remove operations in ASA-graphs are logarithmic of the number of unique keys in the collection in the worst case. The features that support the computational superiority of ASA graphs are the following:

1. The height of the spanned tree of an ASA-graph is always less than or equal to the height of a B-tree or a B+tree constructed for the same data collection

Table 1. Comparison of the time complexities of operations.

Data Structure	Search	Insert	Remove	Min/Max	Med/Avr/Sum
Hash map average	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N \log N)$
Hash map worst case	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N \log N)$
B-tree	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(N)$
B+tree *	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(N)$
RB-tree	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(N)$
AVL-tree	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(N)$
WAVL-tree	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(\log\ N)$	$O(N)$
ASA-graph	$O(\log\ \widehat{N})$	$O(\log\ \widehat{N})$	$O(\log\ \widehat{N})$	$O(\log\ \widehat{N})$	$O(\widehat{N})$

$\widehat{N}$ is the number of unique keys, thus $\widehat{N} \leq N$.
* The classic, non-aggregating version of the B+tree is used for the comparisons in the table.

because ASA-graphs aggregate the representation of all duplicates while B-trees do not.

2. B+trees use extra sign-point nodes that are not used in the trees of ASA-graphs. Therefore, the number of steps along the edges of trees used in ASA-graphs is always less than or equal to the number of such steps in B+trees.

3. B+trees store keys only in leaves, so the search operation must always go along the whole height of these trees to perform most of the operations; therefore, it is slower than similar operations in ASA-graphs.

4. The greatest number of all insert and remove operations in ASA-graphs are finished in insert step 6 and remove steps 8 and 9 without the rebalancing operations because only counters are incremented or decremented. Therefore, it is often faster than the insert or remove algorithms of B-trees or B+trees of the same order, where the aggregations and counting are not used.

The time complexity of ASA-graphs depends on the combined time efficiencies of sorted lists and self-balanced search trees, taking into account that duplicates are aggregated, so the efficiency does not depend on the number of all stored keys but only on the number of all unique keys. Therefore, the following efficiencies are characteristic for ASA-graphs:

- Access to the neighbor keys takes constant time.
- Search for a given key takes logarithmic time of the number of unique keys in the collection.
- Search for a minimum or maximum key takes logarithmic time of the number of unique keys in the collection.
- Insert a key with or without a rebalancing operation takes logarithmic time of the number of unique keys in the collection.
- Remove a key with or without a rebalancing operation takes logarithmic time of the number of unique keys in the collection.
- Computation of a sum, a product, an average, or a median takes linear time of the number of unique keys in the collection.
- Retrieving the number of duplicates of a given key takes logarithmic time of the number of unique keys.
- Moving between objects (entities) defined by the same or similar keys takes constant time.

Note that for the presented operations, the logarithmic time of the number of unique keys ($\widehat{N}$) in the collection can be constant from the point of view of all keys (N) if the number of duplicates in the collection is big. Finally, if we assume that the number of unique keys ($\widehat{N}$) is much less ($\widehat{N} \ll N$) and independent of the number of keys (N) in the collection, then the given operations of ASA-graphs have constant computational complexity and can be successfully compared with specialized structures like hash maps, etc. Table 1 compares the computational complexities of all similarly used structures.

3.6. Data mining examples. An ASA-graph stores more relationships between data than any other data structure discussed in this paper, without a significant loss of performance for typical operations. It performs even faster in some knowledge engineering-related tasks like finding the means, medians, standard deviations, sums, counts, frequent patterns, minima, maxima, similarity, neighborhood, etc. Examples of such operations include the following:

- Calculation of a sum, product, mean, standard deviation, or median can be accelerated by the bidirectional sorted list and duplicate aggregation, and in the worst case is $O(\widehat{N})$.

- Direct connections to neighbors result in instant access to connected elements without in-order searching as is typically performed for binary search trees. This makes finding similarity or neighborhood more efficient.

- ASA-graphs can accelerate frequent and rare patterns finding as a part of more complex graph data structures, like AGDS (Horzyk, 2018). The aggregated elements are connected with object nodes representing patterns, sequences, or other types of relationships, also giving those object nodes direct connections to similar or neighboring values represented by the ASA-graph.

- These complex graphs can be used as the very efficient data structure for the modified k-nearest neighbor algorithm (Altman, 1992), utilizing direct connections between neighboring elements for each data dimension to propose and compute the distance between nearest neighbors in an n-dimensional space (Horzyk and Starzyk, 2019).

- ASA-graphs can also be very useful for clustering in such algorithms as K-means or K-medians thanks to very efficient mean and median calculation.

4. Experiments and efficiency comparisons

In our experiments, we focused on the real-time efficiency comparisons between popular self-balancing trees (natural competitors) and ASA-graphs. We also compared them with other simpler but widely used data structures like hash maps or binary search trees. The computational complexity of operations on AVL-trees, RBT, B-trees, and original B+trees depends on the number of all keys inserted to these structures, while the computational complexity of operations on ASA-graphs depends on the number of unique keys that may be much smaller than the number of all keys (Table 1). Therefore, if data contain a sufficient number of duplicated keys, the operations on ASA-graphs should be more efficient than the same operations on the other structures.

On the other hand, if the number of duplicates is low, then the operations can be a bit less efficient because ASA-graphs are more complex and require some additional operations on pointers and counters. Therefore, the theoretical comparisons of complexities of all mentioned structures are supplemented by the practical experiments that show us the true level of efficiency for each structure according to the size of the collection and the number of duplicates in the collection.

The final time efficiency of all compared structures very much depends on the way how they are implemented and optimized. Therefore, we implemented all the compared structures ourselves to be sure that all these structures are optimized in the same way to get trustful results of the comparisons.

Experiments presented in this section are set to check and confirm the presented theoretical predictions and the analysis of the computational complexities of the self-balancing trees considered. The conducted experiments measured real differences between most frequent operations like insert, search, and remove performed on the mentioned structures in the previous sections. For benchmarks, we chose the detection of IoT botnet attacks dataset from the UCI ML Repository and the Electric Motor Temperature dataset from Kaggle repository. The first dataset is a large collection (above 7 million of instances) of floating-point numbers divided into many files and grouped in 115 features. We used Philips benign, scan, and ack data files (Meidan *et al.*, 2018) to achieve a sufficient amount of records and duplicates for each key. The test scenario includes translating the dataset stored as CSV files into a tested data structure. We chose one column for a selected dataset with a given number of duplicates and treated it as a key. The whole record is transformed into value represented by this key. Because of imprecise floating-point numbers representation in most computers, we converted each key to an integer. Every row is represented as a string. To achieve a representative distribution of duplicates among keys, we merged benign, ack, and scan subsets from the selected dataset into two working datasets.

The first tested dataset has 369 984 records containing features that have the following numbers of duplicates: 2%, 9%, 20%, 32%, 40%, 48%, 56%, 67%, 76%, 84%, and 93% respectively. The second tested dataset has 266 363 records and features of 3%, 12%, 17%, 23%, 29%, 36%, 42%, 50%, 57%, 67%, 73%, 80%, and 89% duplicates respectively.

The second dataset is a collection of 998 070 examples described by 13 numerical features. We narrowed down this collection to 420 000 of examples to obtain the desired number of duplicates and speed up experiments. Columns which contain 2%, 11%, 24%, 41%, 67%, 79%, and 99% were selected and used as keys in our experiments.

The implementation of the tested data structures was developed in C++17 using the GCC 7.3.0 x64 compiler. Every test was performed ten times to ensure that the standard deviation of all measurements is below 5%, and then we present the average results for all tests. We used $std :: multimap$ to test RBT because the GCC 7.3.0 x64 compiler implements this interface using RBT. Similarly, $std :: unordered\_multimap$ is implemented as a hash map in the GCC compiler, and this optimized implementation was used in our tests. B-trees, B+trees, AVL-trees, Wavl-trees, and ASA-graphs have been implemented using best programming practices and optimizations. We implemented B+trees with duplicates

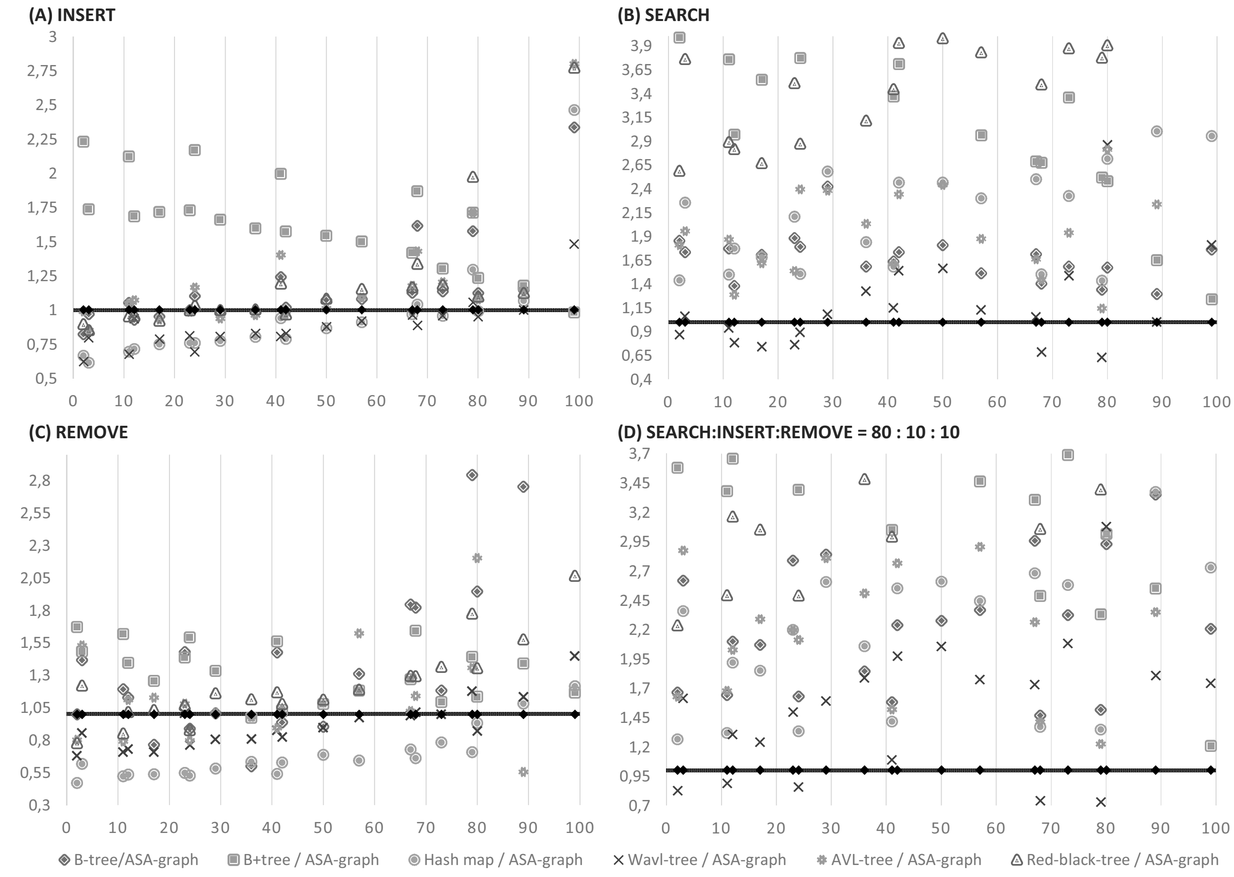

Fig. 4. Comparison of time efficiency of search (a), insert (b), remove (c) operations, and the typical ratio of the search, insert, and remove (d) operations on data. The thick black line is a reference time needed for the ASA-graph. Thus, in all results above, the thick black lines indicate situations where ASA-graphs are faster than the other solutions. The x-axes present the number of duplicates in the compared datasets. The y-axes represent the ratio of the execution time of the compared algorithm to the execution time of the ASA-graph algorithm.

counting and aggregation, not to store duplicates in separate nodes, which would require a large number of recursive non-leaf nodes checking during removal.

4.1. Testing time efficiency of the search, insert, and remove operations. The charts presented in Fig. 4 show the ratios of the time used by all tested data structures to the ASA-graphs as a function of the percentage of the above-considered numbers of duplicates. When these ratios are greater than 1.0, ASA-graphs achieve better efficiency than the compared reference data structures. The high efficiency of ASA-graphs is gained thanks to the aggregation of duplicates (x-axes). The datasets with no duplicates are very rare in real applications, so the achieved efficiencies are representative for the most real data. ASA-graphs perform faster insertion than hash maps if the number of duplicates is greater than approximately 70% even though hash maps store data unordered. The efficiency is slightly lower for the datasets with fewer duplicates.

ASA-graphs are usually smaller than the compared data structures, which affects the memory efficiency. There is also no need to reorganize this structure every time, as every duplicate only increases the counter, not affecting nodes, edges, order of elements, and memory usage. In our experiment, the memory was saved in comparison with other structures when the dataset contained a sufficient number of duplicates (about 70%). The amount of saved memory is larger if the number of duplicates is larger, and savings are greater if each stored key occupies a larger chunk of memory (which depends on its data type).

The next experiment compares the measured performance of a search operation for all keys (Fig. 4(a)). ASA-graphs outperform hash maps because of the hashing collisions that often occur for large datasets. B-trees look competitive in this test, but they were worse in the insertion operation (Fig. 4(b)).

For the removal operation, every value from the previously constructed structures was removed in random order (but the order was the same for every data structure in each test case) until they became empty. All tests proved (Fig. 4(c)) that ASA-graphs are faster than most of the other structures, especially when the number of duplicates is high. The main reason is that the remove operation is much more complex and time-consuming than any other basic operations on those trees. Another reason is that ASA-graphs can finish this operation almost immediately when removing duplicated keys, decrementing the key counters only when they are greater than one. Moreover, ASA-graphs optimize the removal operation much better (thanks to the connections to neighbor-key elements) than B-trees, which must perform more complex operations. Only hash maps outperformed ASA-graphs because of the simpler structure and the lack of representation of relationships between data, resulting in a faster removal without the cost of the rebalancing operation.

Comparing the results of the search, insert, and remove operations (Fig. 4(d)) together and assuming that during the time the number of duplicates increases, we can claim that ASA-graphs almost always outperform efficiencies of all tested data structures except for the situations when the data contain very few duplicates. In these quite rare situations, the other structures are only slightly better than ASA-graphs, so in total, the use of ASA-graphs is still beneficial.

4.2. Testing typical data mining tasks. In typical data mining tasks, we often measure properties of sorted subsets of data. In the performed experiments, we measured the time efficiency of calculation of medians, means, and standard deviations for the numbers of key duplicates between fifty and ninety-fifth percentile. The results presented in Fig. 5 show that ASA-graphs outperform tested data structures in the major part of all test cases. Only B+trees achieved comparable results (in finding means and standard deviations in the lower range of the percentage of duplicated keys), and only after we improved the original code by aggregating duplicates and direct connections between leaves. However, the overall performance of B+trees was still worse than ASA-graphs due to the repetitions of keys in signposts which are present in B+trees.

The results illustrated in Fig. 5 demonstrate a strong advantage of the proposed ASA-graphs in several useful arithmetic operations performed on the stored data. Such operations are often used in data mining, machine learning, and artificial intelligence, therefore, their efficient performance is significant.

The comparisons showed that ASA-graphs and B+trees are more efficient than B-trees in accessing neighbor keys because they use bidirectional lists of elements spanned over the nodes organized in these trees. However, the B+trees access time of all objects by keys is always logarithmic because all objects are stored only in leaves, while for ASA-graphs and B-trees, the access time is shorter because objects are directly accessible from all nodes of these trees. B+trees duplicate many keys to navigate through the structure, so they are the least memory efficient.

ASA-graphs do not duplicate the representation of any key, which can save memory when the stored collection contains a lot of duplicates. However, the memory efficiency is not a primary goal of the use of these graphs, but the time efficiency of all operations and the ability to sort keys and associate objects (entities) defined by the same or similar keys. ASA-graphs span bidirectional lists over all elements and nodes and automatically move and update connections to neighbor keys in constant time when elements are inserted or removed, which is impossible in the commonly used B-trees. Moreover, the aggregations of duplicates can significantly decrease the number of rebalancing operations when the first instance of the key is inserted, or when the last instance of the key is removed. Thanks to these aggregations, ASA-graphs are smaller than B-trees and the other self-balancing trees for data collections containing many duplicates, save memory, and accelerate all operations. When the number of unique keys is much smaller than the number of all keys, the time efficiency of all operations is constant and independent of the number of all keys. ASA-graphs can also support and accelerate various filtering operations thanks to the aggregations of the same keys and storing them in order. This order is always preserved and updated in constant time during every operation, so the sorting is very efficient, and all keys are always available in sorted order.

5. Conclusion

This paper presented novel ASA-graphs and compared them with the commonly used structures in databases for indexing, optimizing disk operations, IoT, and data mining applications. The experiments showed that ASA-graphs are very efficient and provide extra relationships unavailable in B-trees and many other data structures like direct connections to the predecessors and successors or the counts of duplicates. ASA-graphs are usually faster than the other structures in the search, insert, and remove operations and can accelerate various data mining operations. The theoretical analysis of the computational complexities and performed experiments proved this hypothesis. In addition, ASA-graphs save memory in comparison with B-trees and B+trees when data contain many duplicates.

The performed experiments proved that commonly used B-trees and B+trees might be beneficially replaced

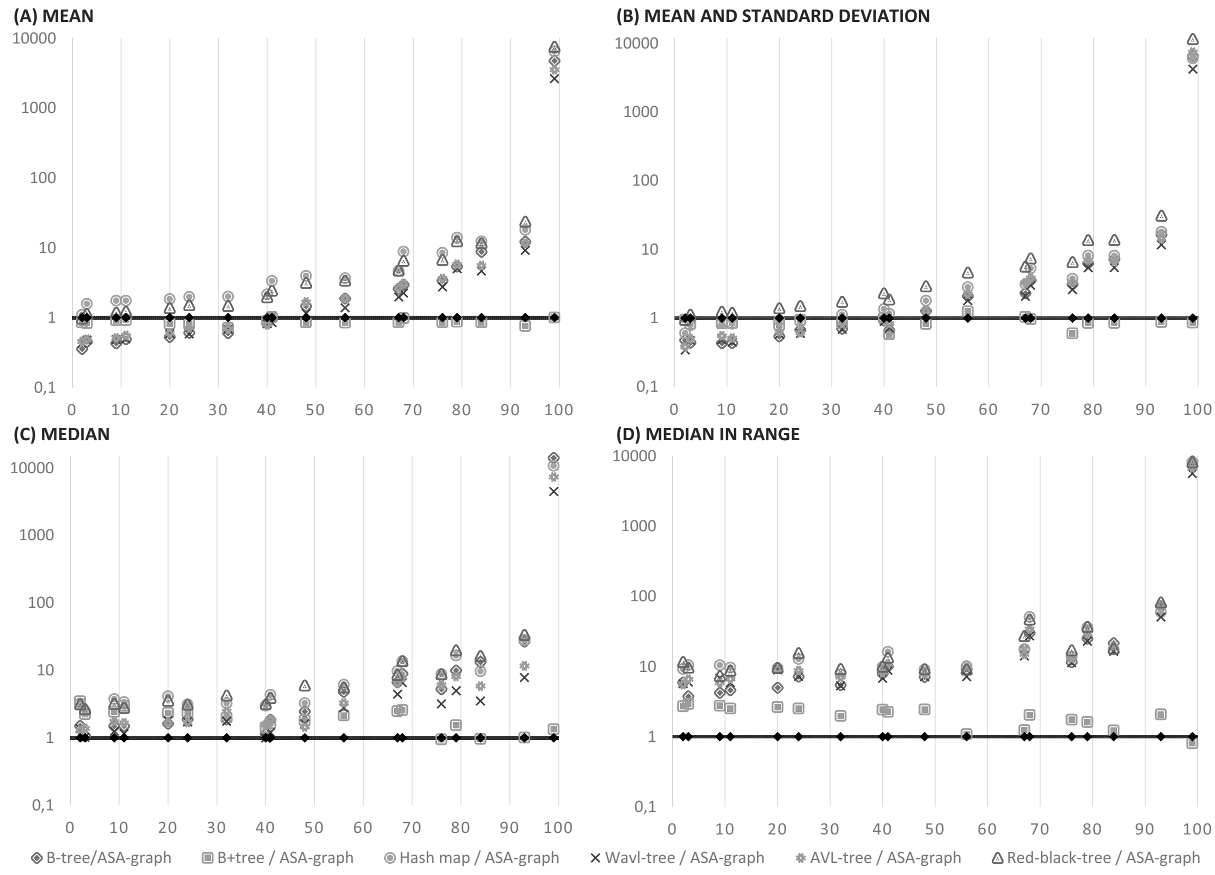

Fig. 5. Comparison of time efficiency of calculation of means (a), mean and standard deviation in the range between the 50th and the 95th percentile of the tested sets (b), medians (c), and medians between the 50th and the 95th percentile of the tested sets (d), where all results above the thick black lines indicate situations where the ASA-graphs are faster than the other solutions. The x-axes present the number of duplicates in the compared datasets. The y-axes (in the logarithmic scale) represent the ratio of the execution time of the compared algorithm to the execution time of the ASA-graph algorithm.

by ASA-graphs. This can substantially accelerate complex operations in big data applications and data mining systems where quick access to the elements, selected or similar keys is needed. The proposed ASA-graphs are not only memory and time-efficient but create new, interesting opportunities in cognitive and knowledge-based applications for fast searching for frequent relationships and collocations between multiple attributes and objects in big data collections.

In our experiments, there was no significant loss of efficiency of ASA-graphs even if the data contained a small number of duplicates or no duplicates at all, so that they can replace other self-balancing trees without risking degradation of performance regardless of the data collection. However, all real-world datasets used in our experiments showed that most of the data collections contain a significant number of duplicated values. Therefore, aggregating duplicates makes sense and reduces the data structure size and memory usage. The greater the reduction, the bigger collection. In the future, we plan to use ASA-graphs to enhance big data processing, data mining, knowledge representation, cognitive and knowledge-based artificial intelligence inferences, memories, and systems.

Acknowledgment

This work was supported by the AGH grant no. 16.16.120.773 and the National Science Centre of Poland grant no. DEC2016/21/B/ST7/02220.

References

Adel'son-Vel'skii, G.M. and Landis, E.M. (1962). An algorithm for organization of information, *Doklady Akademii Nauk* **146**(2): 263–266.

Altman, N.S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression, *The American Statistician* **46**(3): 175–185.

Baran, M. (2018). Closest paths in graph drawings under an elastic metric, *International Journal of Applied Mathematics and Computer Science* **28**(2): 387–397, DOI: 10.2478/amcs-2018-0029.

Bayer, R. and McCreight, E. (1972). Organization and maintenance of large ordered indices, *Acta Informatica* **1**(3): 173–1.

Chen, L. and Schott, R. (1996). Optimal operations on red-black trees, *International Journal of Foundations of Computer Science* **7**(03): 227–239.

Comer, D. (1979). Ubiquitous B-tree, *ACM Computing Surveys* **11**(2): 121–137.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009). *Introduction to Algorithms*, MIT Press, Cambridge, MA.

Dan, L. (2007). *Indexing and Querying Moving Objects Databases*, PhD thesis, National University of Singapore, Singapore.

Demaine, E.D., Harmon, D., Iacono, J. and Pătraşcu, M. (2007). Dynamic optimality—almost, *SIAM Journal on Computing* **37**(1): 240–251.

Fan, K., Wang, X., Suto, K., Li, H. and Yang, Y. (2018). Secure and efficient privacy-preserving ciphertext retrieval in connected vehicular cloud computing, *IEEE Network* **32**(3): 52–57.

Fenk, R. (2002). The BUB-tree, *Proceedings of the 28th VLDB International Conference on Very Large Data Bases (VLDB'02), Hongkong, China*, `https://www.cse.ust.hk/vldb2002/VLDB2002-proceedings/papers/S34P16.pdf`.

Graefe, G. (2011). Modern B-tree techniques, *Foundations and Trends® in Databases* **3**(4): 203–402.

Guibas, L.J. and Sedgewick, R. (1978). A dichromatic framework for balanced trees, *19th Annual Symposium on Foundations of Computer Science (SFCS 1978), Ann Arbor, MI, USA*, pp. 8–21.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching, *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA*, pp. 47–57.

Haeupler, B., Sen, S. and Tarjan, R.E. (2015). Rank-balanced trees, *ACM Transactions on Algorithms* **11**(4): 1–26.

Hibbard, T.N. (1962). Some combinatorial properties of certain trees with applications to searching and sorting, *Journal of the ACM* **9**(1): 13–28.

Horzyk, A. (2013). *Artificial Associative Systems and Associative Artificial Intelligence*, Academic Publishing House EXIT, Warsaw, (in Polish).

Horzyk, A. (2015). Innovative types and abilities of neural networks based on associative mechanisms and a new associative model of neurons, *Proceedings of the International Conference on Artificial Intelligence and Soft Computing, Zakopane, Poland*, pp. 26–38.

Horzyk, A. (2017). Deep associative semantic neural graphs for knowledge representation and fast data exploration, *Proceedings of the 9th International Conference on Knowledge Engineering and Ontology Development, Santa Cruz/Funchal, Madeira, Portugal*, pp. 67–79.

Horzyk, A. (2018). Associative graph data structures with an efficient access via AVB+trees, *Proceedings of the 11th International Conference on Human System Interaction, Gdańsk, Poland*, pp. 169–175.

Horzyk, A. and Starzyk, J.A. (2018). Multi-class and multi-label classification using associative pulsing neural networks, *2018 IEEE World Congress on Computational Intelligence (WCCI 2018)/2018 International Joint Conference on Neural Networks (IJCNN 2018), Rio de Janeiro, Brazil*, pp. 427–434.

Horzyk, A. and Starzyk, J.A. (2019). Associative data model in search for nearest neighbors and similar patterns, *Proceedings of the 2019 IEEE Symposium Series on Computational Intelligence, Xiamen, China*, pp. 932–939.

Horzyk, A., Starzyk, J.A. and Graham, J. (2017). Integration of semantic and episodic memories, *Transactions on Neural Networks and Learning Systems* **28**(12): 3084–3095.

Jensen, C.S., Lin, D. and Ooi, B.C. (2004). Query and update efficient B+-tree based indexing of moving objects, *Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, Canada,* Vol. 30, pp. 768–779.

Kim, W.-H., Seo, J., Kim, J. and Nam, B. (2018). clfB-tree: Cacheline friendly persistent B-tree for NVRAM, *ACM Transactions on Storage* **14**(1): 1–17.

Knuth, D.E. (1998). *Sorting and Searching. The Art of Computer Programming*, Addison-Wesley, Boston, MA.

Lewicki, A. and Pancerz, K. (2020). Ant-based clustering for flow graph mining, *International Journal of Applied Mathematics and Computer Science* **30**(3): 561–572, DOI: 10.34768/amcs-2020-0041.

Mehta, D. and Sahni, S. (2004). *Handbook of Datastructures and Applications*, CRS Press Tylor & Francis Group, Boca Raton, FL.

Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Shabtai, A., Breitenbacher, D. and Elovici, Y. (2018). N-BAIOT—network-based detection of IOT botnet attacks using deep autoencoders, *IEEE Pervasive Computing* **17**(3): 12–22.

Pfaff, B. (2004). Performance analysis of BSTs in system software, *ACM SIGMETRICS Performance Evaluation Review* **32**(1): 410–411.

Sagiv, Y. (1986). Concurrent operations on B*-trees with overtaking, *Journal of Computer and System Sciences* **33**(2): 275–296.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*, Addison-Wesley, Upper Saddle River, NJ.

Sharma, V., Kumar, R. and Kumar, N. (2018). DPTR: Distributed priority tree-based routing protocol for FANETs, *Computer Communications* **122**: 129–151.

Shen, M., Jiang, X. and Sun, T. (2018). Anomaly detection based on nearest neighbor search with locality-sensitive B-tree, *Neurocomputing* **289**: 55–67.

Sun, P., Wen, Y., Ta, D.N.B. and Xie, H. (2016). Metaflow: A scalable metadata lookup service for distributed file systems in data centers, *IEEE Transactions on Big Data* **4**(2): 203–216.

Toptsis, A.A. (1993). B**-tree: A data organization method for high storage utilization, *Proceedings of ICCI'93: 5th International Conference on Computing and Information, Sudbury, ON, Canada*, pp. 277–281.

Wieczorek, M., Siłka, J., and Woźniak, M. (2020). Neural network powered COVID-19 spread forecasting model, *Chaos, Solitons & Fractals* **140**, Article no. 110203.

Woźniak, M., Wieczorek, M., Siłka, J. and Połap, D. (2020). Body pose prediction based on motion sensor data and recurrent neural network, *IEEE Transactions on Industrial Informatics*, DOI: 10.1109/TII.2020.3015934.

Wu, S., Jiang, D., Ooi, B.C. and Wu, K.-L. (2010). Efficient B-tree based indexing for cloud data processing, *Proceedings of the VLDB Endowment* **3**(1–2): 1207–1218.

Adrian Horzyk received his MS degree in computer science from Jagiellonian University, Kraków, Poland, and his PhD and DSc degrees in computer science from the AGH University of Science and Technology, Kraków, where he is currently an associate professor. His present research interests encompass the development of knowledge-based models and methods of artificial and computational intelligence, associative and spiking neurons and their networks and memories, new machine learning strategies and algorithms, data mining, knowledge engineering, and cognitive systems. He is a co-founder and has been a member of the Polish Association of Artificial Intelligence since 2009 and a board member of the Polish Neural Network Society (PTSN) since 2011. He has been a deputy team leader of the AGH University of Science and Technology in the CERN Alice experiments and projects since 2017. He is also an IEEE Senior Member.

Daniel Bulanda received his MD degree from Jagiellonian University, Kraków, Poland, in 2013. He is a PhD student of biomedical engineering at the Faculty of Electrical Engineering, Automatics, Computer Science, and Biomedical Engineering at the AGH University of Science and Technology, Kraków. He is also a computer science student at the Faculty of Electrical Engineering of the Warsaw University of Technology, Poland. His scientific interests include computational intelligence, associative models, computer vision, and application of machine learning techniques in medicine, especially in radiology and cardiology.

Janusz A. Starzyk received his MS degree in applied mathematics and his PhD degree in electrical engineering from the Warsaw University of Technology, Poland, and his DSc degree in electrical engineering from the Silesian University of Technology, Gliwice, Poland. He has been an assistant professor with the Institute of Electronics Fundamentals, Warsaw University of Technology. He has also been a professor of electrical engineering and computer science with Ohio University, USA. Since 2007, he has been the head of the Information Systems Applications Department of the University of Information Technology and Management, Rzeszów, Poland. His current research interests include embodied machine intelligence, motivated goal-driven learning, self-organizing associative spatiotemporal memories, active learning of sensory-motor interactions, machine consciousness, and applications of machine learning to autonomous robots and avatars. He is an IEEE Life Member.

Received: 3 August 2020
Revised: 15 November 2020
Accepted: 20 November 2020