Research Article

A Method for Fast Leaderboard Calculations in Massive Online Game-Based Environments

Julian Moreno Cadavid and Hernán Darío Vanegas Madrigal

Universidad Nacional de Colombia, Bogotá, Colombia

Correspondence should be addressed to Julian Moreno Cadavid; jmoreno1@unal.edu.co

Received 9 November 2017; Accepted 27 March 2018; Published 3 May 2018

Academic Editor: Michael J. Katchabaw

Copyright © 2018 Julian Moreno Cadavid and Hernán Darío Vanegas Madrigal. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Leaderboards and other game elements are present in many online environments, not just in videogames. When such environments have relatively few users, the implementation of those leaderboards is not usually a problem; however, that is no longer the case when they have dozens of thousands or more. For those situations we propose a method that is easy and cheap to implement. It is based on two particular data structures, a Self-Balanced Ordering Statistic Tree and a hash table, to perform proper leaderboard calculations in a fast and cheap way. More specifically, our proposal has $O(\log_2 N)$ time complexity, whereas other approaches also based on in-memory data structures like linked lists have $O(N)$, and others based on Hard Disk Drive operations like a relational database have $O(N \log_2 N)$. Such improvement with regard to the other approaches is corroborated with experimental results for several scenarios, also presented in this paper.

1. Introduction

The online games industry is in constant growth. According to NewZoo [1], between China and the US only, there were more than 1 billion online gamers in 2016, who represented altogether almost \$48 billion in revenues from the \$99.6 billion worldwide, up 8.5% compared to 2015. Just as example, according to SteamCharts [2], Dota 2, a Multiplayer Online Battle Arena (MOBA) game for the Steam platform, had a peak of over a million players in January 2017. That number seems a lot but it looks pale compared with League of Legends (LoL), another MOBA game for Microsoft Windows and MAC OS, which in January 2014 had up to 67 million monthly players, with 27 million playing daily [3].

But not only games themselves are a force to consider, also other online environments are now adopting game-based features to improve users' experiences. A clear example of that is the use of gamification, that is, the use of game elements within nongame contexts, in a great diversity of contexts [4, 5] including education [6, 7], health [8], and marketing [9].

One particular element of game-based environments, and therefore of gamified environments, is leaderboards, a common mechanism to rank players according to their

relative success. Leaderboards measure players against a particular criterion, usually the underlying score, and are thus indicators of progress that relate the player's performance to the performance of others looking for intrinsic motivation [10–14].

Most games and gamified environments use leaderboards and, in order to implement them, as well as for solving persistence, a common solution is using a database approach. A database, or more precisely a Data Base Management System (DBMS) generally stores data in the Hard Disk Drive (HDD) using certain data structures, typically n -ary trees of several types, which allows for fast insertion, deletion, and search, but not so much for ordering (referring specifically to the ORDER BY clause). When the number of players is low, that is, up to hundreds or few thousands, this does not represent a problem for online environments. However, when they have dozens or even hundreds of thousands of players, the response times become a major issue. As an anecdotic reference, that was exactly what motivated this research in the first place: we were working on a gamified learning environment called TICademia (<https://ticademia.com>) and when the number of users only came to a thousand, the leaderboard based on a relational data base approach worked

just fine. However, when such a number raised to the several dozens of thousands, a lot of response time problems started to appear. In rush hours, with almost 25000 active students in the course “pre-calculus,” the response time for the leaderboard page was larger than a minute. It might not sound like too much for most users, but it certainly is for the more “enthusiastic.” Of course, such response time would depend on the technology used in the online environment but, any case, it would never scape the intrinsic time of the ORDER BY task.

A very similar situation happened to Applibot, one of the major social apps providers in Japan. With popular games like Legend of the Cryptids and Gang Road, they were able to scale smoothly and handle the massively growing traffic but found some troubles to maintain up-to-date player rankings or, at least, with their initial database approach [15].

An alternative, not necessarily to replace but to complement the database approach, is to store particular information in faster memory schemes like in the Random-Access Memory (RAM) and use efficient data structures to manipulate it. What we propose is exactly the following: to use an order-statistic tree jointly a hash table, both in RAM, to obtain considerably lower response times.

The rest of the paper is structured as follows. In Section 2, we present some related works. In Section 3, we describe our proposal and then in Section 4 we show and discuss some experimental results. Finally, in Section 5, we present the conclusions of this research.

2. Related Works

When we searched the scientific literature about how other researchers addressed the leaderboard implementation problem, we found three main obstacles. First, we did not find anything in games themselves, nor commercial, neither other kinds. That does not mean they do not deal with the problem. Our educated guess is that they are not interested in showing it, or at least in that context. Second, what we did find were some works on gamified learning environments that use leaderboards, but most of them focused on the motivational or learning outcomes, not in implementation details. Third, just a few of those works present validation scenarios with real users, but in all cases with a reduced number of them, dozens or hundreds at most.

In the case of ALEPS, for example, a gamified learning environment for physics problem solving, the leaderboard shows the top students based on the results of various game elements such as the score, levels, experience, and number of badges. Even if they did not explicitly state using a database approach for the leaderboard calculation, they did state using SQL Server as DBMS for manipulating all user data [16].

The same happened with a gamified learning environment for solving computer programming assignments. Here, they implemented two leaderboards, one that shows the overall score and one that shows the score for the current week. In this case, they mention that the system incorporates a leaderboard calculation service in the application layer. Again, they did not explicitly state the database approach, but they did report the use of Hibernate for data storage,

an Object-Relational Mapping (ORM) library that supports multiple relational database systems, such as HSQLDB and MySQL [17].

Another example is a gamified online course for multimedia content production, implemented in the Moodle Learning Management System. In this case, they use a leaderboard to display all enrolled students sorted in descending order by level and then by experience points (XP). Because they used Moodle instead of creating their own system, they do not present any implementation details. However, looking into Moodle’s documentation, it turns out that it uses XMLDB, a library in the abstraction layer that lets Moodle interact with and access the database, which may be managed by several DMBS like Postgres, MySQL, MSSQL, and Oracle [18].

Outside the scientific literature, it is possible to find some interesting works. In [15], for example, they rightly point out that maintaining a real-time leaderboard is not an easy task because (a) the game environment may have hundreds of thousands players; (b) whenever a player fights enemies or performs other activities, their score changes; and (c) you want to show the latest ranking for the player usually on a web page. They even contrast the simplicity of the implementation with a relational database approach to its poor performance as the scale grows. As a solution, they use an algorithm with similar time complexity as the one we propose in the next section, $O(\log_a N)$. We tried to use this work as a reference in Section 4, but there are two main difficulties. Conceptually, they only use the player score for the ranking function so, unlike our proposal, multiple players may have the same rank. Technically, its throughput has a limitation of about 300 updates/second due to its cloud architecture.

There are also other commercial approaches with similar time complexity, like Amazon ElastiCache for Redis, but they work only in the cloud and are not necessarily cheap.

3. Method Proposed

In order to handle a leaderboard, we first assume that players have at least two attributes, one related to their identifiers and another related to their scores. The identifier, that we will call from now on id, may be numeric or alphanumeric, but in any case, unique. The score may come from a single or multiple data, but in any case, must be comparable. Particularly, we assume that it refers to a single numeric value and that the higher, the better. These two attributes should be stored together in RAM into two different data structures: a Self-Balanced Order-Statistic Tree (SBOST) and a hash table [19–22]. In both cases, the two attributes are part of an object that we will call player. The same two attributes, in addition to all other relevant information like name, alias, avatar, and so on should be stored separately in a database.

A SBOST is a particular kind of a binary search tree. A binary search tree stores nodes, in this case of player type, and each node is linked at most to two subtrees, commonly denoted left and right. A node with no subtrees is called leaf, and the unique top of the whole tree is called root. Now, besides these features, a binary search tree must fulfill a condition: each node must be greater than all nodes in its left subtree, and not greater than any in the right subtree. There

ALGORITHM 1

must be, of course, a way to determine which of two nodes is greater than the other. In our case, this comparison is made according to Algorithm 1.

Notice that this algorithm returns the player with the higher score and, in case of a tie, it uses as an untying criterion the higher id. The reason for doing so is that we assume that a higher id means that such a player started later the game or at least checked in later so he/she had less advantage. Of course, this criterion is completely subjective and may be altered according to the designer needs just by adjusting Algorithm 1. For instance, it might use extra information of the player like time spent on the game or time since the last login, and so on. It might also use more information of the score in the case that it refers to multiple, instead of single, data.

For being balanced, a binary search tree must fulfill another condition: the difference between the heights of the left and right subtrees of any node must be at most 1. The height of a subtree is the maximum number of jumps between the root of the subtree and its deepest leaf. Being balanced has a critical repercussion: considering the binary layout of the tree, as well as the relation of each node with its two subtrees, all the basic operations insertion, deletion, and search can be achieved in $O(\log_2 N)$ time complexity where N is the number of elements stored.

Now, for addressing self-balancing, there are several alternatives, being two of the most popular the Adelson-Velsky and Landis (AVL) tree and the Red-Black Tree. In the second, formerly known as symmetric binary B-tree [23], each node has an extra bit which is often interpreted as the color red or black of the node, so that explains its name. Despite all the particular algorithms for doing so, these color bits are used to ensure the tree remains balanced during insertions and deletions without affecting the $O(\log_2 N)$ complexity.

Finally, being an order-statistic tree means that it supports two additional operations beyond the three mentioned above: selection and ranking. The first one refers to finding the i th smallest element stored in the tree, whereas the second one refers to the opposite, finding the rank, or position in a linear order, of a given element in the tree. For our proposal, we are only interested in the ranking operation. Nevertheless, both can also be performed in $O(\log_2 N)$ when a self-balancing tree is used. However, for doing so, all nodes must store one additional attribute, which is the size of the subtree starting at that node. In other words, it refers to the number of

FIGURE 1: Order-statistic tree.

nodes below and including it. All operations that modify the tree (insertion and deletion, knowing that update may be implemented as a composite function of the two) must consider this attribute and preserve the relation presented in the following without altering the time complexity:

$$\text{node.size} = (\text{node.left})\text{.size} + (\text{node.right})\text{.size} + 1. \quad (1)$$

An example of an order-statistic tree, using the comparison between nodes according to Algorithm 1, is presented in Figure 1.

Considering the previous structure, we can compute the *ranking* operation as presented on Algorithm 2. Notice that besides the *left* and *right* attributes, each node contains a reference to its *parent*. All nodes have a *parent* except for the *root*. Here, we assume that *search()* returns the node of the player we are looking for and that *NULL.size* = 0.

To demonstrate that Algorithm 2 works correctly, we may think of node x 's rank as the number of nodes preceding x in an in-order tree walk, plus 1 for x itself. Then, Algorithm 2 maintains the following loop invariant.

At the start of each iteration of the while loop of lines (8)–(13), r is the rank of x in the subtree rooted at node y .

And we use this loop invariant to show that the algorithm works correctly as follows [19].

Initialization. Prior to the first iteration, line (6) sets r to be the rank of player x within the subtree rooted at x . Setting $y = x$ in line (7) makes the invariant true the first time the test in line (8) executes.

Maintenance. At the end of each iteration of the **while** loop, we set $y = y.\text{parent}$. Thus we must show that if r is the rank of x in the subtree rooted at y at the start of the loop body, then r is the rank of x in the subtree rooted at $y.\text{parent}$ at the end of the loop body. In each iteration of the **while** loop, we consider the subtree rooted at $y.\text{parent}$. We have already counted the number of nodes in the subtree rooted at node y that precede x in an in-order walk, and so we must add the nodes in the subtree rooted at y 's sibling that precede x in an in-order walk, plus 1 for $y.\text{parent}$ if it, too, precedes x . If y is a right child, then neither $y.\text{parent}$ nor any node in $y.\text{parent}$'s left subtree precedes x , and so we leave r alone. Otherwise, y is a left child and all the nodes in $y.\text{parent}$'s right subtree precede x , as does $y.\text{parent}$ itself. Thus, in line (10), we add $y.\text{parent}.\text{right.size} + 1$ to the current value of r .

```

ranking (Integer score, Integer id)
(1) Let x be a node
(2) x = search(score, id)
(3) if x = null
(4)     return -1
(5) end if
(6) r = (x.right).size + 1
(7) y = x
(8) while y ≠ root:
(9)     if y = (y.parent).left:
(10)        r = r + ((y.parent).right).size + 1
(11)     end if
(12)     y = y.parent
(13) end while
(14) return r
end ranking

```

ALGORITHM 2

Termination. The loop terminates when $y = \text{root}$, so that the subtree rooted at y is the entire tree. Thus, the value of r is the rank of x in the entire tree.

Now, if the tree is self-balanced, we already discuss that *insertion*, *deletion*, and *search* operations are performed in $O(\log_2 N)$. Now, notice that, in the previous algorithm, once the node x is found it goes up to the *root* one level at the time, which means that the running time is proportional to the tree height. Therefore, that is why the *ranking* operation also runs in $O(\log_2 N)$.

Now, this *ranking* operation requires both attributes *score* and *id*. In a typical situation, however, we would have only the *id* of the player, and that is where that hash table enters the scene. A hash table is a data structure that can map keys to values using a hash function and is able of doing so in $O(1)$ amortized time. In this case, the key refers to the *id* and the value to the corresponding *score*.

If these two data structures are stored in the RAM (of the environment server) and contain the relevant information of all players, performing a complex task as “increment the score of the player with *id* 1002 in 50 points and determine how many positions he/she gained and the corresponding final position in the leaderboard” could be made in the following manner.

Steps 1 and 5 have $O(1)$ time complexity, whereas steps 2, 3, and 4 have $O(\log_2 N)$; therefore, the entire operation requires $O(\log_2 N)$. Of course, in a real game environment, some of this data should be stored additionally in the database which has a complexity on its own. We are not actually discarding this task, what we are doing is performing the most “expensive” operations, that is, inner ordering and therefore ranking, in a lot more efficient way.

In order to clarify all the algorithms presented so far, consider the next hypothetical situation. There are six players with *scores* and *ids* as shown in Figure 1. Then, as described in Algorithm 3, the player with *id* 1002 increments its *score* in 50 points. Before the increasing, and considering the comparison criteria described in the Algorithm 1, the corresponding leaderboard would stand as presented in Table 1.

TABLE 1: Leaderboard before increasing.

Position	id	Score
1	1006	460
2	1003	450
3	1001	430
4	1002	400
5	1004	380
6	1005	350

FIGURE 2: Order-statistic tree after increasing.

Once the increasing has been done, the corresponding order-statistic tree stands as shown in Figure 2, whereas the corresponding leaderboard changes as presented in Table 2.

Notice that player 1002 moved up one position in the leaderboard. Checking Algorithm 2, it passes from position 4 to 3. It is important however to make a clarification. The *ranking* operation returns the current position of a player given its *id* and *score* (the *score* may be obtained from the *id* with the hash table), but there is no need of running it N times to get the whole leaderboard. Instead, taking advantage of the binary search structure of the tree, it is possible to just traverse the tree in backward “in-order” as in Algorithm 4.

4. Experimental Results

To validate the method proposed, we performed a comparison against two other approaches. The first of those

Step 1. Search for the score s of the player id in the Hash Table
 Step 2. Use $ranking(s, id)$ to obtain current position p
 Step 3. Update score $s' = s + 50$ for player id in both structures, tree and Hash Table
 Step 4. Use $ranking(s', id)$ to obtain new position p'
 Step 5. Return $p' - p$, and p'

ALGORITHM 3

Step 1. Set the *root* as the current node
 Step 2. Check if the current node is null, if not, proceed to step 3
 Step 3. Traverse the right subtree by recursively calling the in-order function
 Step 4. Display information of the current node
 Step 5. Traverse the left subtree by recursively calling the in-order function

ALGORITHM 4

TABLE 2: Leaderboard after increasing.

Position	id	Score
1	1006	460
2	1003	450
3	1002	450
4	1001	430
5	1004	380
6	1005	350

TABLE 3: Average running time (in seconds).

Input size	SBOST	Linked list	Database
1,000	0.0272	0.0591	1.280
2,000	0.0442	0.1426	3.610
5,000	0.0726	0.6631	21.0
10,000	0.1083	2.764	95.2
20,000	0.2738	13.6	342.0
50,000	0.5106	164.6	3,398
100,000	0.8825	844.1	13,555

approaches, as we stated earlier, corresponds to the typical solution used in most online game-based environments, which is a database. More specifically, we use a relational database assuming that all relevant information, that is, player's *score* and *id*, is stored in a single table, so no additional operations like joins are required. As for DBMS we used PostgreSQL 9.6. As for the second approach, we used a linked list data structure running in RAM. As in the method proposed, it does not require HDD operations. However, it differs in several aspects. From the technical point of view, its implementation is a lot easier. In fact, just a few Java code lines are needed considering that a native class *LinkedList* is available, including methods for the insertion, deletion, update, and search. From the algorithms point of view, the time complexities of the required operations are entirely different. The native insertion is $O(1)$ if made at the beginning or end of the structure. However, with a few modifications, it can be done orderly in $O(N)$. When doing so, the ranking operation can take place also in $O(N)$, as well as deletion and search.

For the method proposed, we made the corresponding implementation also in Java. For the hash table we used native classes, but for the SBOST we implemented it from scratch. More specifically, we used a Red-Black Tree scheme to achieve self-balancing of the binary search tree and performed the SBOST operations according to the algorithms described in the previous section.

Now, to compare the three approaches, we arranged a scenario in which there are N players, each one with a unique

id and an initial *score*. After that, there are up to N queries. Similar to the example presented in the previous section, a query refers to the increment of an individual player *score*, expecting as a result the corresponding new position in the leaderboard, as well as the number of positions gained. With the aim of determining the scalability of each alternative, we used random values for queries using N from 1,000, to 2,000, 5,000, 10,000, 20,000, 50,000, and 100,000. For a more robust statistical comparison, we run each case at least ten times and then present the corresponding mean. All runs were made using the same conditions and equipment: Java SE1.8 with Eclipse 4.5.0 in an Intel Core i7-4710HQ at 2.50 GHz, 8 Gb RAM, and 64-bit Windows 8.1.

The results obtained are presented in Table 3. Even with the lower input size, there is a considerable difference between “in-memory” approaches and the HDD-based, that is, between the data structures running in the RAM and the database solution. The difference between the SBOST and the linked list is not too high at the beginning but, as the input size raises, such a difference becomes bigger and bigger. With $N = 100,000$, the SBOST solution is almost 1,000 : 1 faster than the linked list and 15,000 : 1 faster than the database.

To visualize these results, but particularly the relation with the input size, Figure 3 presents them as a chart. For the SBOST solution, this chart bears out that running time exhibits a $O(N \log_2 N)$ behavior, whereas the linked list exhibits $O(N^2)$ behavior and the database $O(N^2 \log_2 N)$

FIGURE 3: Running times for SBOST (a), linked list (b), and database (c). x-axis is for number of queries (in thousands and in logarithmic scale), and y-axis is for average running time (in seconds).

TABLE 4: Multiple linear regression results.

Method	Model	R^2	ANOVA's P value
SBOST	$aN\log_2(N) + b$	0.9823	<0.0001
Linked list	$aN^2 + bN + c$	0.9998	<0.0001
Database	$aN^2\log_2(N) + b$	0.9996	<0.0001

behavior. Even when the shapes in the three cases seem similar, the scales show how different they are. In fact, when running multiple linear regressions for the outcomes of the three solutions, the results presented in Table 4 are obtained whereas the dots on Figure 3 represent the predicted values. In other words, there is empirical evidence for the theoretical time complexity of the three approaches evaluated, including the one of our proposal.

Using those models to extrapolate running times, it results in the fact that, with an input size $N = 1$ million (nothing too outlandish considering the game related numbers presented at the very beginning of the introduction), the SBOST would obtain nearly 10 seconds, whereas the linked list solution would obtain approximately 28 hours and the database 19 days.

5. Concluding Remarks

Leaderboards are a common element of both online games and game-based environments. Even if there is a lot of evidence of their psychological effects on users, we in this research did not focus on that aspect. Instead, considering such importance, what we proposed is an efficient way to implement them.

Our method has three main features. First, it runs “in-memory,” so it exploits fast data access, unlike slower HDD

solutions. Although it might sound as a disadvantage as well, because the reduced space may limit how many users you can have, it is not so problematic considering that in its basic form only the user score and its identifier are needed. For instance, if a four-byte unsigned integer is used for both attributes, a scenario with 100,000 players would require 800,000 bytes, which is less than 1Mb. Second, it uses specific data structures, so no ordering at all is actually needed for obtaining players positions. More specifically, it uses an SBOST jointly with a hash table which allows for performing all important operations in $O(\log_2 N)$ time complexity. The SBOST was implemented from a Red-Black Tree, but other alternatives for Self-Balanced Binary Search Trees could be adopted as well. Third, the comparison criterion, which ultimately defines the rank of a player, may be easily modified in order to adjust to the designer needs. For instance, it could incorporate more information about the player, rather than just a single score and an identifier.

From the algorithmic point of view, such a proposal surpasses typical solutions as the ones based on databases, as well as other “in-memory,” simpler, alternatives as ordered linked lists. As presented in the experimental results section, we achieved speedups on all the scenarios we tested. In fact, the more difficult the scenario, the higher the speedup. For example, such a speedup with an input size $N = 100,000$ was nearly 1,000:1 and 15,000:1 compared to the other two approaches presented. With the forecast coming from a multiple linear regression with $N = 1$ million (actual running of such a scenario would be impractical) the corresponding speedups would be nearly as large as 10,000:1 and 160,000:1. This finding turns out to be very relevant in massive environments where dozens or even hundreds of thousands of users are common.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

- [1] NewZoo, "2016 Global games market report," <http://resources.newzoo.com/hubfs/Reports/Newzoo-Free-2016-Global-Games-Market-Report.pdf>.
- [2] SteamCharts, "Dota 2," <http://steamcharts.com/app/570>.
- [3] Forbes, "Riot Games Reveals 'League of Legends' Has 100 Million Monthly Players," <https://www.forbes.com/sites/insert-coin/2016/09/13/riot-games-reveals-league-of-legends-has-100-million-monthly-players>.
- [4] K. Seaborn and D. I. Fels, "Gamification in theory and action: A survey," *International Journal of Human-Computer Studies*, vol. 74, pp. 14–31, 2015.
- [5] G. Zichermann and J. Linder, *The gamification revolution: How leaders leverage game mechanics to crush the competition*, McGraw-Hill Education, New York, NY, USA, 1st edition, 2013.
- [6] M. Morales, H. R. Amado-Salvatierra, R. Hernández, J. Pirker, and C. Gütl, "A practical experience on the use of gamification in MOOC courses as a strategy to increase motivation," *Communications in Computer and Information Science*, vol. 620, pp. 139–149, 2016.
- [7] R. N. Landers, "Developing a Theory of Gamified Learning: Linking Serious Games and Gamification of Learning," *Simulation & Gaming*, vol. 45, no. 6, pp. 752–768, 2014.
- [8] B. A. Jones, G. J. Madden, and H. J. Wengreen, "The FIT Game: Preliminary evaluation of a gamification approach to increasing fruit and vegetable consumption in school," *Preventive Medicine*, vol. 68, pp. 76–79, 2014.
- [9] J. Hamari, "Transforming homo economicus into homo ludens: a field experiment on gamification in a utilitarian peer-to-peer trading service," *Electronic Commerce Research and Applications*, vol. 12, no. 4, pp. 236–245, 2013.
- [10] C. Butler, "The Effect of Leaderboard Ranking on Players' Perception of Gaming Fun," in *Online Communities and Social Computing*, vol. 8029 of *Lecture Notes in Computer Science*, pp. 129–136, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [11] C. Christy and J. Fox, "Leaderboards in a virtual classroom: A test of stereotype threat and social comparison explanations for women's math performance," *Computers & Education*, vol. 78, pp. 769–785, 2014.
- [12] R. N. Landers and A. K. Landers, "An Empirical Test of the Theory of Gamified Learning: The Effect of Leaderboards on Time-on-Task and Academic Performance," *Simulation & Gaming*, vol. 45, no. 6, pp. 769–785, 2014.
- [13] S. Nebel, M. Beege, S. Schneider, and G. D. Rey, "The higher the score, the higher the learning outcome? Heterogeneous impacts of leaderboards and choice within educational videogames," *Computers in Human Behavior*, vol. 65, pp. 391–401, 2016.
- [14] M. Sailer, J. U. Hense, S. K. Mayr, and H. Mandl, "How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction," *Computers in Human Behavior*, vol. 69, pp. 371–380, 2017.
- [15] Google Cloud Platform, "Fast and Reliable Ranking in Datastore," <https://cloud.google.com/datastore/docs/articles/fast-and-reliable-ranking-in-datastore/>.
- [16] Z. Rasool, A. Bimba, N. F. Mohd Noor, H. Affal, M. N. Ayub, and N. Husin, "Gamification of Web Based Learning Environment for Physics Problem Solving," in *Proceedings of the Second Asian Conference on Society, Education and Technology (ACSET)*, Osaka, Japan, 2014.
- [17] C. Rasmussen and D. Ase, *A Web-Based Code-Editor For Use in Programming Courses [Dissertation thesis for the Master of Science in Computer Science]*, Norwegian University of Science and Technology, 2014.
- [18] G. Barata, S. Gama, J. Jorge, and D. Goncalves, "Engaging engineering students with gamification," in *Proceedings of the 2013 5th International Conference on Games and Virtual Worlds for Serious Applications, VS-GAMES 2013*, UK, September 2013.
- [19] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, UK, 2009.
- [20] D. Knuth, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Mass, USA, 3rd edition, 1997.
- [21] M. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, Hoboken, USA, 5th edition, 2010.
- [22] S. S. Skiena, *The Algorithm Design Manual*, Springer-Verlag, London, UK, 2008.
- [23] R. Bayer, "Symmetric binary B-Trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.