MAP-MIND: An Offline Algorithm for Optimizing Game Engine Module Placement in Cloud Gaming

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier

MAP-MIND: An Offline Algorithm for Optimizing Game Engine Module Placement in Cloud Gaming

IMAN LOTFIMAHYARI[1] (Member, IEEE), LUIGI DE GIOVANNI[2], DAVIDE GADIA[3], PAOLO GIACCONE[1] (Senior Member, IEEE), DARIO MAGGIORINI[3], and CLAUDIO E. PALAZZI[2] (Member, IEEE)

[1]Politecnico di Torino, Torino, Italy
[2]Università degli Studi di Padova, Padova, Italy
[3]Università degli Studi di Milano, Milano, Italy

ABSTRACT Online gaming has seen a significant surge in popularity, becoming a dominant form of entertainment worldwide. This growth has necessitated the evolution of game servers from centralized to distributed models, leading to the emergence of distributed game engines. These engines allow for the distribution of game engine modules (GEMs) across multiple servers, improving scalability and performance. However, this distribution presents a new challenge: the game engine module placement problem. This problem involves strategically placing GEMs to maximize the number of accepted placement requests while minimizing the delay experienced by players, a critical factor in enhancing the gaming experience. The problem can be formulated as an Integer Linear Programming (ILP) model, which provides an optimal solution but suffers from high computational complexity, making it impractical for real-world applications. To address this challenge, this paper introduces two novel heuristic algorithms, MAP-MIND and MAP-MIND*. The MAP-MIND algorithm demonstrates superior performance, achieving near-optimal delay and more than 92% GEM request acceptance in the worst heterogeneous scenarios. The MAP-MIND* algorithm, while slightly under-performing MAP-MIND in terms of delay, proves to be significantly faster, making it a viable alternative for real-world applications with equal GEM request acceptance. The trade-off between the two algorithms offers a flexible approach to GEM placement, balancing performance and computational efficiency.

INDEX TERMS Cloud gaming, distributed game engines, placement algorithm

I. INTRODUCTION

IN recent years, there has been a notable trend in the world of video games: they are increasingly evolving into online services. In an online game, the gaming experience relies on a server providing a shared exchange point for many players. This central hub may provide a single-user experience as well as a shared virtual environment for the users to interact with each other. In any case, the server is responsible for efficiently managing the shared resources among the player population, independently from game mechanics.

Online games' evolution has seen many stages. In the beginning, a centralized (physical) server was in charge of managing all gaming sessions, with obvious scalability issues. In the second generation, with the increasing availability of cloud computing, gaming servers have been distributed in the cloud and run as a combination of virtualized software services. The adoption of cloud computing to implement cloud gaming allowed for a significant cost reduction and an improvement in scalability. In the last generation, game streaming has been introduced. In game-streaming, the computation is completely offloaded to the remote (cloud-based) server, and a video stream is sent to the player [1], [2]. A client handles the player's inputs, which are sent back to the game server. This paradigm shift eliminates the need for gamers to acquire and maintain cutting-edge hardware while making high-quality gaming experiences more accessible than ever before. As a result, the number of players using game streaming is ramping up, opening up new business opportunities for modern game companies. Services like Google Stadia [3] (now discontinued), Amazon Luna [4], GeForce Cloud Gaming [5], and Microsoft Xbox Cloud [6] exemplify this model, where the player's device is only responsible for displaying the game scene streamed from private data centers equipped with

dedicated hardware.

Hopefully, the new way to provide gaming services will move millions of existing and next-generation players from legacy platforms to cloud gaming. Anyway, such a large-scale migration might generate a workload able to overtax even hyper-scaling cloud architectures. While the capability to scale up in terms of raw computation power can be given for granted, modern cloud architectures are not designed to organize internal modules while taking into account strict Quality of Service (QoS) requirements. Currently, in today's cloud architectures, gaming services are placed where resources are available using legacy optimization policies and relying on over-provisioning to fulfill the QoS required by the user for an optimal gaming experience. We anticipate that the future of gaming services will demand the distribution of game server software, operating within game engines, across cloud infrastructure. Accommodating this shift while prioritizing the gaming experience and resource optimization presents a substantial challenge, particularly for conventional monolithic game engines.

The primary research challenge at present revolves around the design and development of a new generation of game engines capable of harnessing the dynamic resource allocation provided by cloud architecture. These next-generation game engines are envisioned as being composed of discrete modules, each handling a specific aspect of the game's functioning, named Game Engine Modules (GEMs) [7]. GEMs can be launched on-demand, dispersed throughout the cloud continuum based on resource availability, and strategically located to reduce gaming delay for the user [8].

Strategically positioning these modules within the cloud continuum is an intricate endeavor that directly impacts the system's performance and user experience. One distinct feature of GEM placement, which differentiates it from virtual machine placement, is that GEMs can be utilized by multiple players simultaneously. As a result, the placement process necessitates a continuous, synchronous backstream for all participating players. This situation is comparable to synchronously streaming a video to a multitude of viewers, with the unique twist that these viewers can ''interact'' with the watching video but are unable to reverse the stream.

The GEM placement challenge entails determining the most advantageous positions for these modules within the cloud continuum. The aim is to minimize the delay perceived by the gamer, providing a smooth and seamless game-play experience. However, this is a complex task that presents various challenges. While the dynamic nature of cloud resources complicates the GEM placement problem, the focus is on the challenges associated with the varying requirements of different GEMs, such as compute power, memory, and network bandwidth. An equally critical consideration is the network delay between the GEMs and the gamers. Optimal GEM placement should prioritize the reduction of the player's experienced delay, ensuring a seamless and highly responsive gaming experience. However, achieving this requires a thorough understanding of the network structure and the geographical distribution of players. It is worth mentioning that the aforementioned cloud gaming platforms are closed-source and do not disclose the techniques they employ for the optimal placement of the monolithic games they offer.

In this study, we define the optimal GEM placement problem, considering each GEM's bandwidth and resource needs. Initially focusing on a single-player per-game session, our approach is then extended to accommodate multiple players within the same game arena, both scenarios being of significant practical relevance. Our model aims to maximize the acceptance of GEM placement requests while minimizing the delay experienced by players. The first optimization goal leads to both optimized resource utilization and enhanced player satisfaction due to the acceptance of more play requests.

To solve this optimization problem, we develop a two-phase approximation algorithm, denoted as MAximized-Placement with MINimized-Delay (MAP-MIND), which runs on a set of GEM requests. In the first phase, MAP-MIND places the GEMS to maximize the number of accepted GEMs. In the second phase, MAP-MIND revises the placement to minimize delay. Subsequently, we introduce MAP-MIND*, a faster variant of MAP-MIND, which serves as a practical alternative for real-world applications, albeit with a slight trade-off in terms of delay compared to the original MAP-MIND. We compare MAP-MIND and MAP-MIND* with the optimal solver, which suffers from limited scalability. Given that no existing solutions share our specific optimization objective, we also compare them with standard placement algorithms that either minimize delay (derived from the scenario of Virtual Machine placement in cloud computing systems) or maximize placement through bin-packing standard approaches. Through extensive simulations, we show that MAP-MIND closely approximates the optimal solution achieved by the solver in terms of both the delay and the number of accepted GEMs.

Furthermore, it also outperforms the other algorithms in terms of delay, number of accepted GEMs, and running time. On the other hand, MAP-MIND* reduces the execution time at the cost of degraded delays. This demonstrates the effectiveness of our approach in tackling the GEM placement problem. It is important to highlight that our model can handle scenarios involving dynamic resource usage. Our resource utilization strategy is based on estimating the worst-case resource requirements for the games.

The remainder of this paper is organized as follows: Sec. II describes the architecture of distributed game engines. In Sec. III, we provide a formal description of the problem and formulate a mathematical optimization model for the scenario of both a single player per GEM and many players per GEM (i.e., a game arena scenario). In Sec. IV, we propose MAP-MIND and MAP-MIND* as approximation algorithms to solve the problem presented in Sec. III. In Sec. V, we assess by simulation the performance of both MAP-MIND and MAP-MIND* by comparing them with some proposed

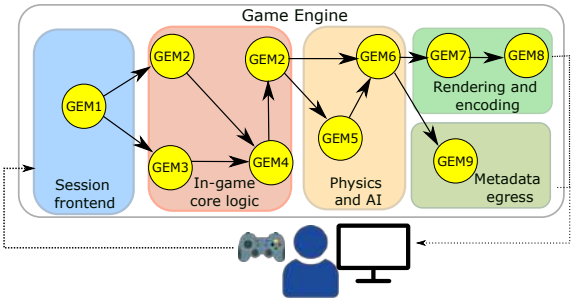

FIGURE 1. Example of game session workflow in CODEG [8].

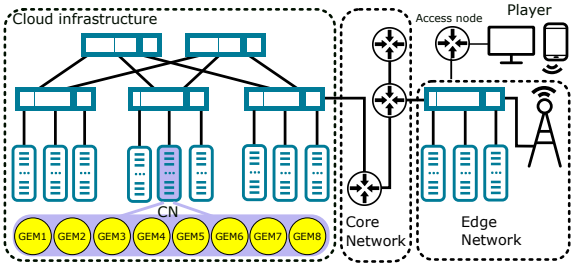

FIGURE 2. Example of compute nodes (CN) and GEM co-location.

TABLE 1. Notation.

Notation	Description
N	Set of network nodes (i.e., compute nodes or access nodes)
E	Set of network links
$\mathcal{T}$	Network topology
K	Types of resources available at the nodes
r_n^k	Amount of resource of type k available at node n
ρ_s^k	Amount of resource of type k needed by game session s
S	Set of game sessions to place
GEM_s	Group of co-located GEM running game session s
P	Set of all the players
P_s	Set of players in the game session s
$h(p)$	Access node of player p
$d(p,n)$	Delay experienced by player p if GEM_s is placed in node n
$d_{\max}(s,n)$	Maximum delay experienced by players in P_s if GEM_s is placed in node n
$d_{\text{tot}}(s,n)$	Summation of the delays experienced by the players in P_s if GEM_s is placed in node n
$d_s^{\max}$	Maximum delay allowed for each player of the game session s
$N(s)$	Set of nodes for which the maximum delay is satisfied for all the players in the game session s
λ_s	Reserved bandwidth for game session s
b_{uv}	Available bandwidth on link (u,v)
$\mathcal{P}(s,n)$	Set of links used by the traffic for game session s if GEM_s is placed in node n
ϕ_{sn}^{uv}	$=1$ if the link (u,v) belongs to $\mathcal{P}(s,n)$, 0 otherwise
x_{sn}	$=1$ if GEM for game session s is placed at node n, 0 otherwise

variants and with the state-of-the-art approaches. In Sec. VI, we discuss the related work in the state-of-the-art literature. Finally, we draw our conclusions in Sec. VII.

II. CODEG: A CLOUD-ORIENTED DISTRIBUTED GAME ENGINE APPROACH

The Cloud-Oriented Distributed Engine for Gaming (CODEG) model, as presented in [8], introduces a groundbreaking approach to cloud gaming by fully utilizing the capabilities of modern cloud infrastructure. This model redefines game engines as network-wide operating systems, which are partitioned into functional units known as Game Engine Modules (GEMs). Fig. 1 shows an example of a game session in which GEMs are grouped into different functionalities necessary to implement the game service. The request for new game sessions triggers the request to start new GEMs. All the requests that arrive during a predefined observation time window are allocated based on the placement algorithm.

Each GEM represents a basic component of a game engine, activated as needed, and eventually linked to other GEMs to implement the whole game service. This modular approach allows for greater flexibility and scalability, as GEMs can be deployed based on real-time game demand. The CODEG model operates under the assumption that modern cloud infrastructures are composed of multiple Compute Nodes (CNs). These CNs could be physical servers in a data center or virtual servers in a cloud environment, as shown in Fig. 2. The model utilizes this distributed architecture to host the GEMs, efficiently spreading the workload of the game engine across multiple CNs.

The model also assumes that all players are connected to the same network provider and that the data center hosting the game engine is optimally located to minimize delay. This is vital for ensuring a smooth gaming experience, as significant delays can result in interruptions and disruptions during gameplay. Furthermore, the CODEG model takes advantage of advanced network virtualization techniques. These techniques allow for the creation of logical networks with specific QoS levels. This ensures that game-related traffic has guarantees of minimum bandwidth and maximum delay. In the CODEG model, the servers hosting the GEMs are connected through the network. The nodes and edges of this network represent the CNs and the logical or physical links between them, respectively.

Each game session in the CODEG model has a unique maximum tolerable delay. This is defined based on the nature of the game and is set to maximize fairness among players of the same multiplayer game session and to satisfy the Quality of Experience (QoE) perceived by each player.

III. OPTIMAL GEM PLACEMENT

In this section, we describe our system model, and then we formulate the placement problem, whose objective is to maximize the number of accepted game sessions while minimizing the overall delay experienced by their players. This combined objective function is designed to maximize the profits of the game providers while maximizing the QoE perceived by the players as well. Each game can be implemented by composing multiple GEMs. However, for the sake of simplicity, this work considers games that contain a single GEM or multiple GEMs that are co-located, as shown in Fig. 2. We now introduce the adopted notation, reported also in Table 1. Let $\mathcal{T} = (N, E)$ represent the graph describing the network topology, where E is the set of network links and N is the set of network nodes, which can be

either CNs or access nodes, where the players connect to the network. We define K as the set of resource types available in the nodes, including CPU, memory, and storage. We use r_n^k to denote the amount of resources of type $k \in K$ available at node $n \in N$.

Let S be the set of game sessions to place. We assume multiple players for each game session; as a special case, a game session could be played by a single player. We also assume a single GEM or a group of co-located GEMs per game session s, and we denote it by GEM_s. Let P be the set of all the players, whereas $P_s \subseteq P$ be the subset of players associated with the game session $s \in S$. Let ρ_s^k, for any $k \in K$, be the amount of resource of type k that is required by GEM_s. Note that such value may depend on the actual number of players (e.g., according to a proportional law), serving as an upper bound for the worst-case scenario of the required resource amount. Let $h(p) \in N$ be the access node of player $p \in P$. Assume now that GEM_s is placed on node $n \in N$. Let $d(p,n)$ be the delay experienced by player p, computed as the sum of the communication delay from $h(p)$ to n, the GEM_s processing delay, and the communication delay from n to $h(p)$. Let $d_{\max}(s,n)$, for any $s \in S$, be the maximum delay experienced by any of the players in P_s: $d_{\max}(s,n) = \max_{p \in P_s} d(p,n)$. Similarly, let $d_{\text{tot}}(s,n)$ be the summation of all the delays experienced by each of the players in P_s: $d_{\text{tot}}(s,n) = \sum_{p \in P_s} d(p,n)$. Note that it is possible to pre-compute the set of all $d_{\max}(s,n)$ and $d_{\text{tot}}(s,n)$ with $O(N^2|P|)$ operations. Let $d_s^{\max}$ be the maximum allowed delay for game session s, which has been devised to satisfy the expected performance at the endpoint. This parameter expresses the main threshold related to the QoS constraint. Given $d_s^{\max}$, it is possible to pre-compute the set $N(s) \subseteq N$ of nodes where GEM_s can be placed such that all the players experience a delay compatible with the maximum allowed one: $N(s) = \{n \in N \mid d_{\max}(s,n) \leq d_s^{\max}\}$.

We assume single-path routing according to the shortest path. Let $\mathcal{P}(s,n)$ be the overall routing paths, i.e., the set of links used to route traffic from all the access nodes of the players in the games session s to node n and back from n to all the access nodes. We define an indicator function ϕ_{sn}^{uv} equal to 1 iff the link (u,v) belongs to $\mathcal{P}(s,n)$. We assume that the bandwidth is reserved for each game session through a dedicated network slice. The bandwidth reserved for the game session s along the links belonging to the routing paths (i.e., for any $e \in \mathcal{P}(s,n)$) is denoted as λ_s, and b_{uv} represents the available bandwidth on link $(u,v) \in E$. Notably, λ_s is equal to the maximum bandwidth, among all the links in $\mathcal{P}(s,n)$, required by the aggregate traffic for game session s, and it typically grows with $|P_s|$. Note that all the parameters defined so far are pre-computed based on the network topology and the maximum allowed delay.

The decision variables are denoted by binary variables x_{sn}, defined as equal to 1 iff GEM_s is placed at node n. Whenever there are not enough resources in a CN for a game session or the corresponding constraint on the maximum delay cannot be met, the game session request is dropped, and all the corresponding players abort the game.

We formulate the problem as an Integer Linear Programming (ILP) model. The problem can be viewed as a generalized assignment problem, as the sole decision revolves around identifying the CN where each game session GEM should be placed. The optimal GEM placement for multi-player GEMs can be formulated as follows:

$$\max_{\{x_{sn}\}} \left(\alpha \sum_{s \in S} \sum_{n \in N(s)} x_{sn} - \sum_{s \in S} \sum_{n \in N(s)} x_{sn} d_{\text{tot}}(s,n)\right) \tag{1}$$

subject to

$$\sum_{n \in N(s)} x_{sn} \leq 1, \qquad \forall\, s \in S \tag{2}$$

$$\sum_{s \in S : n \in N(s)} \rho_s^k x_{sn} \leq r_n^k, \qquad \forall\, k \in K, n \in N \tag{3}$$

$$\sum_{s \in S, n \in N(s)} \lambda_s \phi_{sn}^{uv} x_{sn} \leq b_{uv}, \qquad \forall\, (u,v) \in E \tag{4}$$

$$x_{sn} \in \{0,1\}, \qquad \forall s \in S, n \in N(s) \tag{5}$$

The objective function (1) combines two components and aims to maximize the total number of accepted game sessions (first term), and secondly, to minimize the overall delay experienced by the players (second term). Indeed, α is a large enough constant such that the objective function will always prefer solutions with a larger number of accepted game sessions, independently from the overall delay. To achieve this, we choose $\alpha = |P|(2d^{\text{net}} + d^{\text{proc}})$, where d^{net} is the diameter of the network in terms of delay and d^{proc} is the maximum processing delay per game session. Constraint (2) assigns the GEM for each game session to at most one CN; indeed, a game session may be dropped due to a lack of resources in the CNs compatible with the related maximum delay requirement. Constraint (3) ensures that the CN resources are not exceeded. Constraint (4) ensures that the reserved bandwidth for each game session does not exceed the network link capacity. Finally, (5) defines the domain of the decision variables. We now claim the following:

Property 1. *The optimal GEM placement problem is NP-hard.*

Proof. We show the problem complexity by reduction from the multiple knapsack problem, which is known to be NP-hard, in a weak sense. By assuming no constraints on bandwidth (i.e., $b_{uv} = \infty, \forall (u,v) \in E$), (4) can be omitted. By assuming no constraints on the maximum delay, all CNs become delay-eligible for all GEMs (i.e., $N(s) = N$). Now, the placement of GEMs is decided solely by the resources on each CN. We can simplify these resources to just one type, such as CPU. The problem of optimizing GEM placement then becomes identical to the multiple knapsack problem, where N represents the set of knapsacks, $\mathcal{S}$ the set of items, the CN's CPU capacity represents the capacity of each knapsack, and the CPU demand of the GEMs represents the sizes of the items to be selected, and the profit for item s in knapsack n is represented by $\alpha - d_{\text{tot}}(s,n)$. □

IV. APPROXIMATED ALGORITHMS

Property 1 implies that an optimal solver for the GEM placement problem is likely not scaling well with problem instance size, and this motivates the design of an approximated approach able to solve large instances of the problem within a constrained time-frame. In response, we have introduced two approximate search algorithms inspired by local search methods, MAP-MIND and MAP-MIND*, each striking a distinct balance between efficiency and computational complexity.

We note that our devised approach is general and implementation-independent. It can be integrated with any resource orchestrator, e.g., Kubernetes for the case of virtualization obtained with containers.

A. THE MAP-MIND ALGORITHM

We are interested in scalable algorithms able to cope with large input instances. We propose an approximate search algorithm called *MAximize-Placement and MINimize-Delay* (*MAP-MIND*). *MAP-MIND* aims to simplify and solve the problem of optimized placement by addressing the multi-criteria objective function (1) through two distinct phases. The first phase, denoted as MAP, is devoted to maximizing the acceptance of the game sessions' requests, and the second one, denoted as MIND, aims at minimizing the average delay experienced by the players. Coherently with (1), MAP-MIND prioritizes maximizing acceptance over minimizing the average delay.

In the first phase, the algorithm strives to accommodate as many game session requests as possible, regardless of the incurred delay. To do so, the GEMs are sorted based on the maximum delay allowed by their players (i.e., $d_s^{\max}$ for GEM$_s$). This choice is motivated by the fact that it gives more chances to the GEMs with small delay requirements to be placed in nodes closer to the access nodes compared to other GEMs with large delay requirements. The algorithm starts to place the sorted GEMs using a Best-Fit approach [9], where the GEMs' resource requirements are considered with respect to the available resources on the computing nodes. The algorithm divides the sum of each resource requirement by the total available resources of the same type within the network. The resource with the highest ratio is deemed the *decision-maker resource* by the Best-Fit algorithm. This algorithm prioritizes nodes with smaller available values of the decision-maker resource. As a result, it optimizes the utilization of available resources throughout the network.

For the second phase, the placement solution of the first phase is used as the initial solution to minimize the delays. This phase is iterative and adopts two possible steps to modify the current solution: (i) *move*, (ii) *swap*. A node n is said to be eligible for GEM$_s$ if it belongs to $N(s)$, it has enough resources to run GEM$_s$, and there is enough bandwidth along the routing path, during the current iteration of the algorithm. The *move* step tries to move each placed GEM to another eligible node, if available, which gives the players of that GEM the minimum experienced delay. This procedure will be repeated until no GEM movement is available to reduce the players' experienced delay. In the subsequent *swap* step, the algorithm tries to swap the position of each GEM with another GEM to reduce the average delay experienced across all players. The reason behind prioritizing the *move* step over the *swap* step is due to the Best-Fit approach used in the first phase, which may lead to having some empty nodes (especially at low loads) that could be efficiently used by the *move* steps rather than the *swap* steps.

The pseudocode of MAP-MIND is provided in Fig. 3. It takes as input S with the game session requests, the graph describing the network topology, the available bandwidth on the links, and the available resources on the nodes.

Input: S: Set of game session requests
Input: $\mathcal{T} = (N, E)$: Network topology
Input: Available bandwidths and resources in the network
procedure MAP-MIND
  `PHASE 1: MAximize Acceptance (MAP)`
  $Y \leftarrow \emptyset$ ▷ Initialize temporary placement
  $x_{sn} \leftarrow 0 \ \ \forall s \in S, n \in N$ ▷ Initialize GEMs placement
  $k_1 = \underset{k \in K}{\operatorname{argmax}} \left(\frac{\sum_{s \in S} \rho_s^k}{\sum_{n \in N} r_n^k} \right)$ ▷ Identify the relative bottleneck resource
  sort $s \in S$ w.r.t. $d_s^{\max}$ in incr. order ▷ Sort by maximum tolerable delay
  for $s \in$ sorted S **do** ▷ For each game session request s
    $n' \leftarrow -1$, score $\leftarrow \infty$ ▷ Initialize best node and its score
    for $n \in N$ **do** ▷ Find the best eligible node
      if (n is eligible for s) $\wedge$ ($r_n^{k_1} <$ score) **then**
        score $\leftarrow r_n^{k_1}, n' \leftarrow n$ ▷ Update selected node and score
    if $n' \neq -1$ **then** ▷ If the best node is found
      $Y \leftarrow Y \cup \{(s, n')\}$ ▷ Update the temp placement
      $x_{sn'} \leftarrow 1$ ▷ Update placement variables
      Update the available bandwidth along $\mathcal{P}(s, n')$ and $r_{n'}^k \ \ \forall k \in K$
  `PHASE 2-1: MINimize average Delays (MIND:Move step)`
  sort Y w.r.t. $d_{\text{tot}}(s, n)$ in desc. order ▷ Sort by total delay
  Improved = True
  while Improved = True **do** ▷ Iterate until no improvement is experienced
    Improved = False
    for $(s, n) \in$ sorted Y **do** ▷ For each GEM
      $n'' \leftarrow -1, \Delta \leftarrow 0$ ▷ Initialize best destination node and its score
      for $n' (\neq n) \in N$ **do**
        if (n' is eligible for s) $\wedge$ ($d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n') > \Delta$) **then**
          $n'' \leftarrow n', \Delta \leftarrow d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n')$ ▷ Update node, score
      if $\Delta \neq 0$ **then**
        $Y \leftarrow Y \setminus \{(s, n)\} \cup \{(s, n'')\}$ ▷ Move from n to n''
        $x_{sn''} \leftarrow 1, x_{sn} \leftarrow 0$ ▷ Update placement variables
        Update the available bandwidth along $\mathcal{P}(s, n)$ and $\mathcal{P}(s, n'')$
        Update $r_n^k, r_{n''}^k \ \ \forall k \in K$
        Improved = True
  `PHASE 2-2: MINimize average Delays (MIND:Swap step)`
  Improved = True
  while Improved = True **do** ▷ Iterate until no improvement is experienced
    Improved = False
    for $(s, n) \in$ sorted Y **do** ▷ For each GEM
      $(s'', n'') \leftarrow (-1, -1), \Delta \leftarrow 0$ ▷ Initialize best swap and its score
      for $(s', n')(\neq (s, n)) \in$ sorted Y **do** ▷ For each GEM
        if (n' is eligible for s) $\wedge$ (n is eligible for s') **then**
          $\Delta' \leftarrow d_{\text{tot}}(s, n) + d_{\text{tot}}(s', n') - d_{\text{tot}}(s, n') - d_{\text{tot}}(s', n)$
          if $\Delta' > \Delta$ **then**
            $(s'', n'') \leftarrow (s', n')$ ▷ Update candidate
            $\Delta \leftarrow \Delta'$ ▷ Update score
      if $\Delta \neq 0$ **then**
        $Y' \leftarrow Y' \setminus \{(s, n), (s'', n'')\} \cup \{(s, n''), (s'', n)\}$ ▷ Swap
        $x_{sn} \leftarrow 0, x_{s''n''} \leftarrow 0$ ▷ Update old placement variables
        $x_{sn''} \leftarrow 1, x_{s''n} \leftarrow 1$ ▷ Update new placement variables
        Update the available bandwidth along $\mathcal{P}(s, n)$ and $\mathcal{P}(s'', n'')$
        Update the available bandwidth along $\mathcal{P}(s'', n)$ and $\mathcal{P}(s, n'')$
        Update $r_n^k, r_{n''}^k \ \ \forall k \in K$
        Improved = True
  return $\{x_{sn}\}_{s \in S, n \in N}$

FIGURE 3. Pseudocode of `MAP-MIND`**.**

In the first phase, after the initialization of the placement variables and a temporary placement set, the decision-maker resource will be determined (ln. 3-5). Now, GEMs are sorted in increasing order based on their maximum tolerable end-to-end delay (ln. 6). Then, for each GEM in sorted order, the best destination node and its score will be initialized (ln. 7-8). Then, all the nodes are checked to find the eligible node with the minimum available decision-maker resource, coherently with a Best-Fit approach (ln. 9-11). The GEM and the selected node are then added to the set of accepted GEMs, while the related placement variable and the available bandwidth and resources of the network are updated accordingly (ln. 12-15). If no eligible node is found, the GEM's related placement variables will remain unassigned.

Then, the second phase starts. During the first step (*move*), the accepted GEMs are sorted based on their average players' delay (ln. 17). For each accepted GEM, the algorithm tries to find another eligible node that minimizes the end-to-end delay compared to the current GEM placement. If such a node is found, the GEM placement is updated accordingly, and the resources of the source and destination nodes along with the corresponding bandwidths are updated (ln. 19-31). This process is repeated until no improvement *move* is found.

During the second step (*swap*), the algorithm finds pairs of accepted GEMs whose swap will be feasible in terms of maximum delay, bandwidth, and resources, and the overall delay will be decreased (ln. 33-43). If such a pair is found, it updates the placement allocation and the corresponding resource availability (i.e., bandwidth, memory, CPU, and storage) (ln. 45-50). This process is iterated until no improving swaps can be executed.

We also present a simplified version of MAP-MIND, called *MAP-MIND**, whose pseudocode is reported in Fig. 4. It mimics the original MAP-MIND structure in two phases, but in the second phase, it combines the *move* step (ln. 19-27) and the *swap* step (ln. 29-41) at each iteration. This approach allows us to reduce the running time compared to MAP-MIND but at the cost of slightly more delay, as shown in the following section.

Input: S: Set of game session requests
Input: $\mathcal{T} = (N, E)$: Network topology
Input: Available bandwidths and resources in the network
1: **procedure** MAP-MIND*
2: `PHASE 1: MAximize Acceptance (MAP)`
3: $Y \leftarrow \emptyset$ ▷ Initialize temp placement
4: $x_{sn} \leftarrow 0 \quad \forall s \in S, n \in N$ ▷ Initialize GEMs placement
5: $k_1 = \underset{k \in K}{\operatorname{argmax}} \left(\frac{\sum_{s \in S} \rho_s^k}{\sum_{n \in N} r_n^k} \right)$ ▷ The relative bottleneck resource
6: **sort** $s \in S$ w.r.t. $d_s^{\max}$ in incr. order ▷ Sort by maximum tolerable delay
7: **for** $s \in$ sorted S **do** ▷ For each game session request s
8: $n' \leftarrow -1$, score $\leftarrow \infty$ ▷ Initialize best node and its score
9: **for** $n \in N$ **do**
10: **if** (n is eligible for s) $\wedge$ ($r_n^{k_1} <$ score) **then**
11: $n' \leftarrow n$, score $\leftarrow r_n^{k_1}$ ▷ Update selected node and score
12: **if** $n' \neq -1$ **then**
13: $Y \leftarrow Y \cup \{(s, n')\}$ ▷ Update the temp placement
14: $x_{sn'} \leftarrow 1$ ▷ Update placement variables
15: **Update** the available bandwidth along $\mathcal{P}(s, n')$ and $r_{n'}^k \quad \forall k \in K$
16: `PHASE 2: MINimize average Delays (MIND:MOVE or SWAP)`
17: **sort** Y w.r.t. $d_{\text{tot}}(s, n)$ in desc. order ▷ Sort by total delay
18: **for** $(s, n) \in$ sorted Y **do**
19: $n'' \leftarrow -1, \Delta \leftarrow 0$ ▷ Initialize best destination node and its score
20: **for** $n' (\neq n) \in N$ **do** ▷ Check for possible MOVE
21: **if** (n' is eligible for s) $\wedge$ ($d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n') > \Delta$) **then**
22: $n'' \leftarrow n', \Delta \leftarrow d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n')$ ▷ Update node, score
23: **if** $\Delta \neq 0$ **then** ▷ MOVE
24: $Y \leftarrow Y \setminus \{(s, n)\} \cup \{(s, n'')\}$ ▷ Move from n to n''
25: $x_{sn''} \leftarrow 1, x_{sn} \leftarrow 0$ ▷ Update placement variables
26: **Update** the available bandwidth along $\mathcal{P}(s, n)$ and $\mathcal{P}(s, n'')$
27: **Update** $r_n^k, r_{n''}^k \quad \forall k \in K$
28: **else** ▷ Check for possible SWAP
29: $(s'', n'') \leftarrow (-1, -1), \Delta \leftarrow 0$ ▷ Initialize best swap and its score
30: **for** $(s', n') (\neq (s, n)) \in$ sorted Y **do** ▷ For each GEM
31: **if** (n' is eligible for s) $\wedge$ (n is eligible for s') **then**
32: $\Delta' \leftarrow d_{\text{tot}}(s, n) + d_{\text{tot}}(s', n') - d_{\text{tot}}(s, n') - d_{\text{tot}}(s', n)$
33: **if** $\Delta' > \Delta$ **then**
34: $(s'', n'') \leftarrow (s', n'), \Delta \leftarrow \Delta'$ ▷ Update GEM, score
35: **if** $\Delta \neq 0$ **then** ▷ SWAP
36: $Y' \leftarrow Y' \setminus \{(s, n), (s'', n'')\} \cup \{(s, n''), (s'', n)\}$
37: $x_{sn} \leftarrow 0, x_{s''n''} \leftarrow 0$ ▷ Update old placement variables
38: $x_{sn''} \leftarrow 1, x_{s''n} \leftarrow 1$ ▷ Update new placement variables
39: **Update** the available bandwidth along $\mathcal{P}(s, n)$ and $\mathcal{P}(s'', n'')$
40: **Update** the available bandwidth along $\mathcal{P}(s'', n)$ and $\mathcal{P}(s, n'')$
41: **Update** $r_n^k, r_{n''}^k \quad \forall k \in K$
42: **return** $\{x_{sn}\}_{s \in S, n \in N}$

FIGURE 4. Pseudocode of `MAP-MIND*`.

V. NUMERICAL EVALUATION

A. SIMULATION METHODOLOGY

To evaluate the performance of our proposed algorithms, we developed an event-driven simulator using Python 3.8 on an Ubuntu 20.04.6 LTS system with 16GB RAM and an 8X Intel Core-i5-10210U-1.60GHz CPU. We also run an optimal solver (OPT) obtained by implementing the ILP formulation presented in Sec. III using AMPL as modeling language and Cplex 12.8.0 as a solver. The solver runs on an Ubuntu 18.04.6 LTS system with 32GB of RAM and a 12X Intel Core-i7-8700-3.20GHz CPU.

The network topology is a random geometric graph with a specified number of nodes ($|N|$) and average degree (g). We only considered connected instances of the graph. The edges of the graph represent the communication links between pairs of nodes, and the propagation delays are set proportional to the link lengths. For generality, we normalized the delay of each link to the average of all shortest paths in the same graph. We assumed a negligible processing delay for all GEMs at each node and null network access delay, since for simplicity (but without loss of generality) we defined $d_s^{\max}$ for game session s at the net of the processing delays and access delays. We also assumed unlimited bandwidth and routing based on the shortest path. We considered two main settings for the maximum tolerable delay of the GEMs: (i) No Delay Constraint (NDC) in which $d_s^{\max} = \infty$ to model all the games for which the perceived delay is not a relevant constraint (e.g., chess); (ii) Uniform Delay Constraint (UDC) in which $d_s^{\max}$ is uniformly distributed between 0 and maximum Round Trip Time (RTT) in the network graph, to model a large variety of game scenarios with different sensitivity to delays. Players' access nodes for each game session are selected uniformly from the available nodes, where the same node can be selected more than once.

Each node is equipped with CPU, memory, and storage capacities set to 5 GHz, 32 GB, and 512 GB, respectively.

For every GEM, the requirements for CPU, memory, and storage are uniformly distributed between 0 and 1 GHz, 0 and 1 GB, and 0 and 1 GB, respectively. We determine the Utilization Factor (UF) based on the system's bottleneck resource. According to the above scenario settings, we expect the CPU to be such a bottleneck. Our choice is influenced by the observation that, in real-world scenarios, one resource will invariably become the limiting factor in placement decisions. While our primary focus is on the bottleneck resource, we also factor in other potential non-bottleneck resources in our requirements. This approach ensures that we consider the influence of all resources on the algorithms. To achieve the desired UF, we generate GEMs sequentially. This generation continues until the cumulative bottleneck resource requirements of the produced GEMs in S either match or surpass the product of the target UF and the total CPU capacity across the entire network.

B. TEST SCENARIOS

We explore the following test scenarios:

- Scenario 1: *Varying Number of Players per GEM* - We examine the influence of changing the number of players per GEM, i.e., for each game session, varying it in the set $\{1, 2, 10, 50\}$, with $|N| = 32$, $g = 4$, and UF $= 0.8$. We chose these values based on modern game genres. Single-player games like Red Dead Redemption II [10] and Horizon Forbidden West [11] often require substantial resources due to large maps or complex AI. Two-player games can be competitive, like Street Fighter 6 [12] and FIFA 23 [13], or collaborative like Portal 2 [14]. In eSports, games typically involve 10 players with two teams of 5 competing, as seen in Dota 2 [15], League of Legends [16], Valorant [17], and Counter-Strike [18]. Battle royales like Fortnite [19], Player Unknown Battle Grounds (PUBG) [20], and Fall Guys [21] fall in the 50-player category, although Fortnite and PUBG can start with up to 100 players, and Fall Guys with 40 to 60, depending on the map. However, matchmaking timeout issues often reduce the starting player count, making an average of 50 players a reasonable assumption for our simulations.
- Scenario $2a$: *Varying Number of CNs* - We explore the system's behavior with different network sizes by varying $|N| \in \{16, 32, 48\}$, with $g = 4$, and UF $= 0.8$.
- Scenario $2b$: *Scaling resources of CNs* - We explore the system's behavior where the resources of nodes are scaled by a factor in $\{1, 4, 16\}$ relative to a base resource amount for the compute nodes (CPU, memory, and storage) with $|N| = 32$, $g = 4$, and UF $= 0.8$.
- Scenario 3: *Varying Average Graph Degree* - We investigate the impact of varying connectivity levels within the network, which is a primary determinant of RTT latency in a shortest-path routed network. This investigation involves varying $g \in \{3, 4, 5\}$, with $|N| = 32$ and a utilization factor of UF $= 0.8$.
- Scenario 4: *Varying UF* - To study how different levels of resource utilization affect the system's performance, we vary the UF $\in \{0.1, 0.5, 0.8, 0.85, 0.9, 0.95, 0.97, 0.99\}$, with $|N| = 32$, and $g = 4$. It is worth noting that by considering the variation in UF, we inherently incorporate the impact of changing the number of game session placement requests.
- Scenario 5: *Varying UF with Heterogeneous Resources* - We consider a scenario where the nodes' resources are randomly selected to be either in $\{1\,\text{GHz}, 8\,\text{GB}, 128\,\text{GB}\}$ or in $\{5\,\text{GHz}, 32\,\text{GB}, 512\,\text{GB}\}$ (i.e., CPU, memory, and storage respectively), simulating a more heterogeneous environment and reflecting variations in different resources across nodes. Similar to scenario 4, we vary the UF $\in \{0.1, 0.5, 0.8, 0.85, 0.9, 0.95, 0.97, 0.99\}$, with $|N| = 32$, and $g = 4$.
- Scenario 6: *Multiple-resource Bottleneck* - We consider a scenario to assess how the algorithms behave in uncommon situations where more than one resource acts as a bottleneck. To create this scenario, we replicated scenario 4 but introduced two resource bottlenecks by maintaining the same ratio between the two resource requirements of the GEMs (e.g., CPU and memory) and the corresponding available resources in the network. For each GEM, the CPU, memory, and storage requirements are uniformly distributed between 0 and 1 GHz, 0 and 6.4 GB, and 0 and 1 GB, respectively.
- Scenario 7: *Batch-placement in an online scenario* - We explore applicability to dynamic resource availability in an online scenario showcasing a realistic system with game sessions arriving and leaving. We consider in total 10^5 requests for GEM to arrive following a Poisson process. Following the realistic cases, the game session associated with each GEM has a random duration in the interval $[60, 3600]$ s (e.g., the average game session length in PUBG [20]), after which the GEM releases the occupied resources. We assume that $|N| = 32$ and $g = 4$. The CPU is considered the sole bottleneck, creating a single-bottleneck resource scenario even with multiple resources, reflecting a realistic situation. We set the offered load to the network to be 0.5, resulting in average inter-arrival times for the GEMs of 11.25 s. We run the placement algorithm every Δt time, denoted as "batch-window", on all the GEMs that arrived during the last batch-window, considering the available resources. All the unplaced GEMs are dropped. We set $\Delta t \in \{0, 11.25, 25, 50, 100, 200, 400\}$ s, where the value 0 means that the placement algorithm runs GEM by GEM. The corresponding average batch sizes are shown in $\{0, 1, 2, 4.5, 9, 18, 36\}$ GEMs respectively. The different values for Δt reflect different tradeoffs between computation complexity and reactivity to changes in the workload. The number of players for each incoming GEM is uniformly selected from $\{1, 2, 4, 10, 50\}$ to showcase a real system in which different kinds of games exist. When each game session

finishes, the corresponding resources are released. This scenario showcases our approach's adaptability to online environments, using the batch-window concept to balance complexity and workload responsiveness in fluctuating cloud resource availability scenarios.

We considered both 1-player GEMs and 2-player GEMs as 2 sub-scenarios for each of scenarios 2 to 6. These scenarios collectively provide a comprehensive system examination, considering various parameters and configurations. By exploring these different aspects, including the specific cases of 1-player and 2-player GEMs, we can derive a robust understanding of the system's characteristics and performance under diverse conditions.

For each test, we generated different random network topologies and for each topology, we generated multiple random sets S according to the mentioned parameters.

We measured two main metrics: the *acceptance probability* for the GEMs and the *average normalized delay*, averaged across all the players and normalized based on the average RTT of each network topology. Finally, we compared the complexity and execution time of all the algorithms. It is important to note that the described scenarios represent a selected subset of multiple test scenarios we considered to examine the algorithms' behavior in various situations. We excluded test cases that showed behavioral results similar to those already reported.

C. ALTERNATIVE ALGORITHMS

For comparison, we considered the optimal algorithm, denoted as OPT, solving the problem (1)-(5). To our knowledge, prior research has not delved into heuristic algorithms for the game engine module placement problem, rendering our study pioneering in this area. Thus, for comprehensive comparison, we evaluated the following alternative algorithms.

- *RaNDom (RND)*: It selects the GEMs in random order. For each selected GEM, it finds an eligible node in random order.
- *Quality-driven heuristic* (QDH*)*: It selects the GEMs in random order. For each selected GEM, it finds an eligible node by searching the nodes with $d_{\text{tot}}(s,n)$ in increasing order. The algorithm is an extension for multiplayer to QDH proposed in [22], which was designed for the placement of single-tenant containers.
- *First-Fit-Decreasing (FFD)*: It sorts the GEMs based on the CPU requirement in decreasing order. For each GEM, it finds an eligible node by searching the nodes in random order. The algorithm belongs to the First-Fit-Decreasing (FFD) heuristics that have been shown to have given often good results for one-dimensional bin-packing problems [23].
- *MAP-RaNDom_First (MAP-RNDF)*: It is a two-phase algorithm, with the first phase identical to MAP in MAP-MIND. For the second phase, it selects the GEMs in random order, and for each selected GEM, it checks the nodes in random order to find an eligible node that reduces the overall delay if the GEM is moved there. Then it checks the other GEMs in random order to find the first one that is eligible to swap with and reduces the overall delay.
- *MAP-RaNDom_Greedy (MAP-RNDG)*: It is a two-phase algorithm similar to MAP-RNDF. But in the second phase, for each randomly selected GEM, it checks all the nodes for the move or swap steps (as in MAP-MIND) and chooses the one that minimizes the overall delay.
- *MAP-STeepest_Descent (MAP-STD)*: It is a two-phase algorithm extending the search options compared to MAP-RNDG. At each iteration, for each GEM, it considers all possible eligible nodes for the move and all possible GEMs to swap with and selects the option that minimizes the overall delay. It keeps iterating until no improvement on the overall delay is possible.

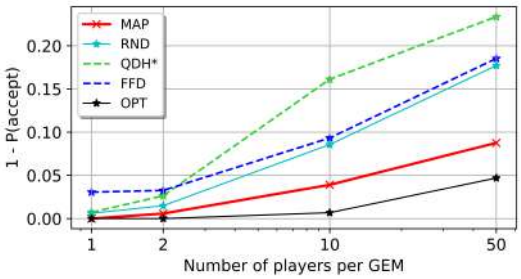

FIGURE 5. Probability of dropping game session requests under scenario 1 for UDC.

D. SIMULATION RESULTS

It is important to note that, in all experiments, the dropping probability and delay graphs should be analyzed in conjunction, with a particular emphasis on the dropping probability as the key objective. Additionally, for all two-phase algorithms where MAP serves as the first phase (i.e., MAP-MIND, MAP-MIND*, MAP-RNDF, MAP-RNDG, and MAP-STD), the acceptance probability is the same. To maintain clarity in the probability graphs across all experimental scenarios and avoid redundancy, we represent all these algorithms solely with MAP.

Scenario 1: Varying the number of players per GEM

In Fig. 5, we report the acceptance ratio for the UDC setting, whereas Fig. 6 shows the average normalized delay for NDC and UDC settings, under scenario 1. We have omitted the dropping probability for NDC since it consistently remains at zero.

In both UDC and NDC settings, each algorithm individually exhibits similar behavior. Generally, in the case of multi-player GEM with a high number of players, all algorithms tend to exhibit similar delays. This phenomenon occurs because, in high-player GEM scenarios, player distribution across different nodes forces GEM placement onto nodes with greater network centrality. This can result in

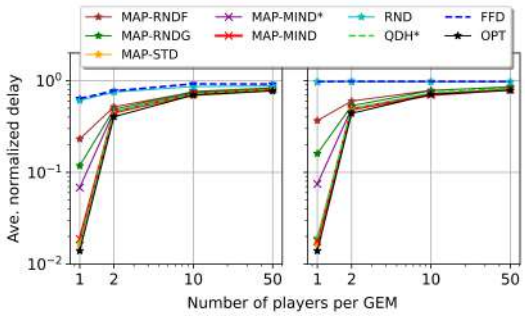

FIGURE 6. **Average normalized delay under scenario 1 for UDC (left) and NDC (right).**

TABLE 2. **Average execution time [s] under scenario 1 for UDC.**

Algorithm	1-player	2-player	10-player	50-player
RND	0.010	0.012	0.035	0.143
FFD	0.016	0.019	0.048	0.179
QDH*	0.020	0.028	0.112	0.442
MAP	0.027	0.032	0.09	0.332
MAP-RNDF	0.813	0.830	1.05	2.05
MAP-RNDG	0.908	1.03	1.49	3.17
MAP-MIND*	0.873	0.969	1.54	3.71
MAP-MIND	1.37	1.96	6.30	15.6
MAP-STD	39.8	48.6	166	613
OPT	0.209	0.264	3.44	840 (i)

(i) Calculated only for the instances that were finished before 900 s wall-clock time (i.e., 88% of them).

higher dropping in less efficient algorithms, particularly in the case of UDC. However, certain algorithms opted to drop some GEMs, which subsequently led to lower delays in UDC, as we will elaborate on shortly. Lastly, as the number of players per GEM is reduced, higher efficiency algorithms demonstrate a greater reduction in the delay while maintaining a low dropping probability.

From the figures, as expected, all two-phase algorithms that use MAP as their first placement phase show the closest results for dropping probability to OPT. They have no dropping for 1-player GEMs, while they show a very small dropping probability for 2-player GEMs and a moderate increase in dropping probability up to 50-player GEMs. Among them, in terms of delay, MAP-MIND and MAP-STD show the closest delay to OPT while starting with MAP-MIND*, MAP-RNDG, and MAP-RNDF the experienced delay becomes worse. On the other hand, all the one-phase algorithms such as RND, QDH*, and FFD start dropping GEMs even for 1-player GEMs. For GEMs with a higher number of players, their dropping probability increases noticeably. RND shows the best probability of dropping among the three mentioned ones as it performs load balancing, which results in the highest delay as shown in the figures as well. QDH* achieves lower delay as it prioritizes reducing delay over dropping, so it drops more than RND while achieving near-optimal delay. As anticipated, FFD displays the worst delays since its primary goal is to enhance the acceptance probability. Interestingly, it presents the highest drop probability for 1-player and 2-player GEM scenarios. This is because it overlooks another crucial GEM placement constraint: the GEMs' maximum tolerable delay. However, for a large number of players per GEM, specifically more than 10 players, the suitable nodes based on delay for different GEMs primarily narrow down to the network central nodes. Now, its dropping probability becomes less than QDH* algorithm and converges to the RND algorithm as it does a simple load balancing. Notably, all algorithms, except RND and FFD, converge to a similar value for the delay as the number of players per GEM increases.

Table 2 presents the execution times for different algorithms under scenario 1. Notably, MAP-STD, the heuristic akin to MAP-MIND, exceeds typical setup times for real game environments. In contemporary multiplayer games, a wait time of approximately 10 s (up to 60 s before gameplay) is deemed acceptable for establishing co-players or rivals in an arena. Conversely, MAP-MIND operates under a worst-case scenario involving over $12,000$ players across approximately 256 games, each accommodating 50 players with diverse delay thresholds. Despite this complexity, the algorithm achieves near-optimal delay and less than 8% dropping. MAP-MIND*, while providing expedited execution at the expense of slightly increased delay, demonstrates the potential for further time reduction through efficient implementation. A comparison of MAP with other 2-phase algorithms underscores the negligible time required for the MAP phase relative to the subsequent phase, focused on refining placement solutions through moves and swaps to minimize delays.

Optimal results for both 1-player and 2-player GEM scenarios typically require less than 1 s of computation time. However, for the 10-player GEM scenario, the average execution time extends to a few tens of seconds, with some instances taking approximately 40 s, indicating notable variability. Remarkably, for 50-player GEM scenarios, computation time varies significantly, with instances being resolved in under 10 s or requiring over 900 s.

Scenario 2a: Varying the number of CNs

In this experiment, we report results under scenario 2a and only for the UDC case. This is because, in the case of NDC, all algorithms' dropping probability is zero, and the delay results are similar to the UDC case. We report for the 1-player and 2-player GEM cases. The dropping probability and the delay experienced by the GEMs for UDC are shown in Fig. 7 and Fig. 8 respectively. The results regarding the experienced delay under scenario 2a are entirely consistent with the results of the 1-player and 2-player GEMs in the previous experiments under scenario 1. An interesting observation is that by increasing the number of CNs, the dropping probability of one-phase algorithms (i.e., FFD, QDH*, and RND), decreases for both 1-player and 2-player GEMs. This is due to the increase in possible CNs for GEM placement. For 1-player GEMs, the delay for smarter algorithms reduces by increasing the CNs, but in 2-player GEMs, only the central CNs are eligible due to the possible positions of the

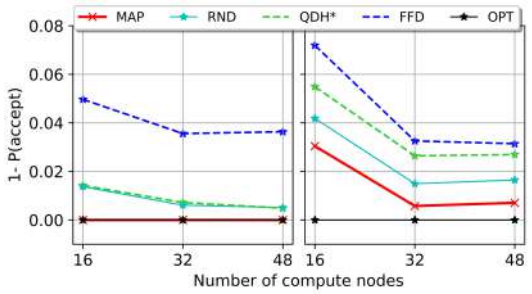

FIGURE 7. Probability of dropping under scenario 2a for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

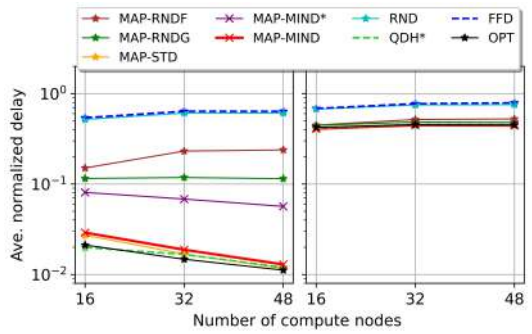

FIGURE 8. Average normalized delay under scenario 2a for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

TABLE 3. Average execution time [s] under scenario 2a for UDC.

Algorithm	$\|N\| = 16$		$\|N\| = 48$	
	1-player	2-player	1-player	2-player
RND	0.005	0.007	0.019	0.023
FFD	0.006	0.008	0.034	0.041
QDH*	0.009	0.013	0.040	0.059
MAP	0.010	0.013	0.055	0.073
MAP-RNDF	0.238	0.225	2.04	2.09
MAP-RNDG	0.246	0.274	2.46	2.78
MAP-MIND*	0.226	0.253	2.33	2.63
MAP-MIND	0.387	0.425	4.39	5.70
MAP-STD	4.41	4.94	158	206
OPT	0.084	0.098	0.489	0.653

2 players of each GEM. In summary, MAP-based algorithms experience the closest dropping probability to OPT, while one of our proposed algorithms, MAP-MIND, is also close to OPT in terms of experienced delay.

We conducted a comparison of algorithm execution times for this scenario, where the average number of GEMs is 128, 256, and 384 respectively. The results are presented in Table 3. Interestingly, MAP-STD, the heuristic most similar to MAP-MIND, requires 10 to 30 times longer to achieve similar delay results. OPT consistently achieves results in less than 1 s on average, irrespective of the number of CNs. MAP-MIND* demonstrates lower computation times compared to MAP-MIND. QDH* and MAP-STD yield improved delay results, while the first exhibits increased GEM dropping and the second significantly longer computation time.

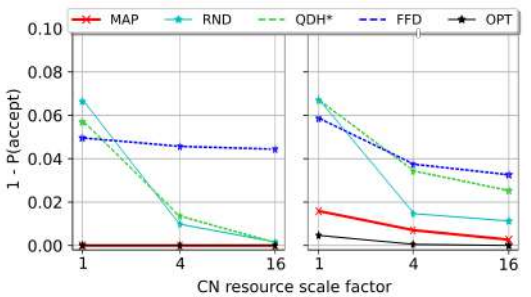

FIGURE 9. Probability of dropping under scenario 2b for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

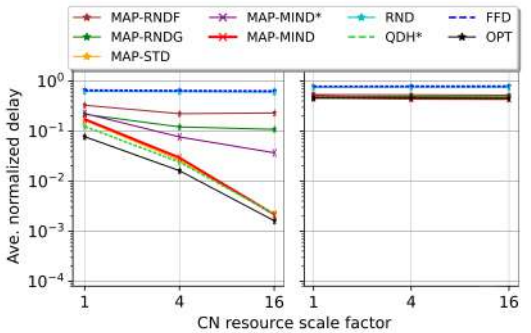

FIGURE 10. Average normalized delay under scenario 2b for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

Scenario 2b: Scaling resources of CNs

Under scenario 2b, we present only the results for the UDC case. The reason is that, similarly to scenario 2a in the NDC case, all algorithms exhibit a null dropping probability, and the delay results align closely with those observed in the UDC scenario. The dropping probability and the delay experienced by the GEMs for UDC are shown in Fig. 9 and Fig. 10 respectively.

As anticipated, our proposed algorithms maintain a minimum dropping probability of zero (similar to optimal) across all network scales for 1-player GEMs. With increased resources, QDH* and RND converge rapidly to our algorithms due to the widened disparity between GEM and resource sizes, mitigating competition for available resources. In terms of delay, MAP-MIND exhibits behavior close to optimal, almost matching QDH*, while MAP-MIND* trades off some delay for execution speed, outperforming competitors except QDH*. For multiplayer GEMs, all MAP-based algorithms outperform RND, QDH*, and FFD in dropping probability, approaching results closest to optimal. Regarding the delay, all algorithms demonstrate similar behavior, with RND and FDD showing slightly worse delays.

Scenario 3: Varying average graph degree

An interesting, yet expected, observation is that by increasing the average degree of the graph, the dropping probability of the less sophisticated algorithms slightly decreases for UDC with 2-player GEMs. This is because as the graph

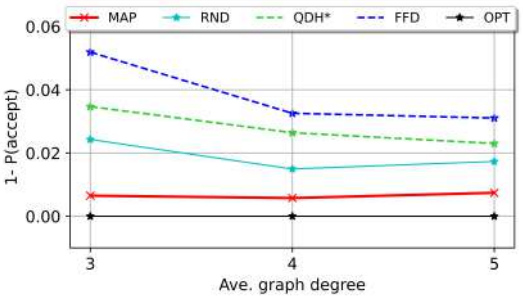

FIGURE 11. Probability of dropping under scenario 3 for UDC with 2-player GEMs.

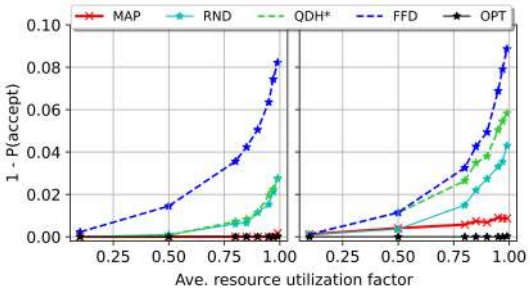

FIGURE 12. Probability of dropping under scenario 4 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

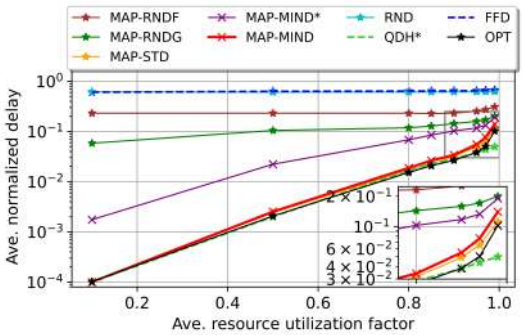

FIGURE 13. Ave. normalized delay under scenario 4 for UDC with 1-player GEMs.

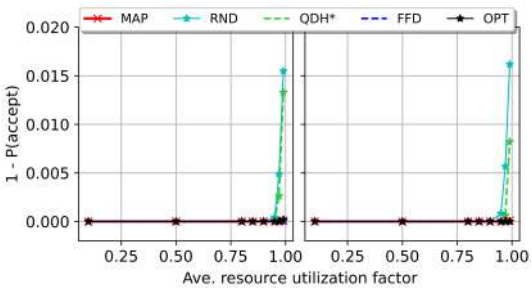

FIGURE 14. Probability of dropping under scenario 4 for NDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

becomes more connected, the length of the shortest paths decreases, which is equivalent to reducing network delays. This leads to more delay-eligible CNs for less sophisticated algorithms at each GEM placement, thereby reducing their dropping probability. For NDC, no dropping occurs for any of the algorithms. However, we observed that altering the average degree of the graph does not affect the average normalized delay results, and all previous observations regarding the behavior quality of the algorithms remain consistent. The dropping probability for UDC with 2-player GEMs is illustrated in Fig. 11.

Scenario 4: Varying the utilization factor

The dropping probability for UDC with both 1-player and 2-player GEMs is illustrated in Fig. 12. From the results of NDC, we report the dropping probability of both 1-player and 2-player GEMs as shown in Fig. 14. This is because it presents the only interesting behavior that differs from the aforementioned scenario. The delay results for UDC with 1-player GEMs are illustrated in Fig. 13. We do not report the delay results for UDC with 2-player GEMs, as it does not provide any interesting observations, being similar to the 2-player case in Fig. 8.

For 1-player GEMs, considering Fig. 13 and Fig. 12 together, as expected, RND and FFD exhibit the worst delay since they are indifferent to the achieved delay. RND shows some minor dropping, especially at high loads. However, interestingly and contrary to what is expected, FFD displays a considerable dropping probability (i.e., the worst among all algorithms for UF $\geq$ 0.5). This is due to the algorithm ignoring the important role of maximum tolerable delay for GEMs while trying to maximize placement.

On the other hand, QDH* achieves a good delay, almost approximating the OPT, which is gained by dropping some of the GEMs. All the MAP-based algorithms show almost zero dropping, even at the highest loads, while MAP-STD and MAP-MIND display the best delays, completely approximating the OPT. Notably, as discussed in Sec. V-E, the execution time of MAP-MIND, is much less than that of MAP-STD, making MAP-MIND the winner. Also, MAP-MIND* demonstrates a completely acceptable delay compared to other algorithms by considering its brilliant mix of execution time and dropping probability.

In the case of 2-player GEMs, other than RND and FFD, which achieve the worst delay, the remaining algorithms show almost identical delay results. The difference lies in the dropping probability, where we observe an increase for all algorithms, except for the OPT, compared to 1-player GEMs. This behavior for 2-player GEMs is expected, as in most cases, only the central CNs are eligible due to the potential positions of the two players of each GEM.

For NDC with 1-player GEMs, when the offered load exceeds 0.95 of the network resources, RND and QDH* start to drop the GEMs, even for NDC. This highlights the weakness of the algorithms that do not consider placement maximization in their logic. Now, compared to UDC cases, FFD can demonstrate its placement power by having no dropping, even at the highest load, as it does not consider the maximum tolerable delay as a constraint for GEM placement.

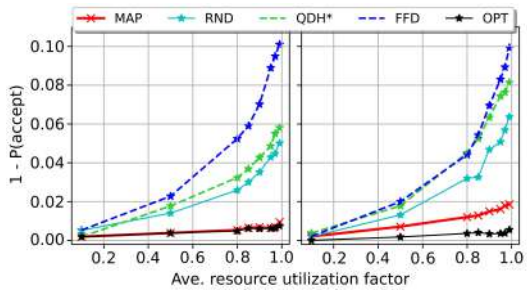

FIGURE 15. Probability of dropping under scenario 5 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

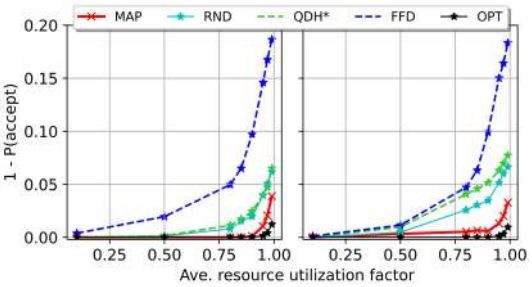

FIGURE 17. Probability of dropping under scenario 6 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

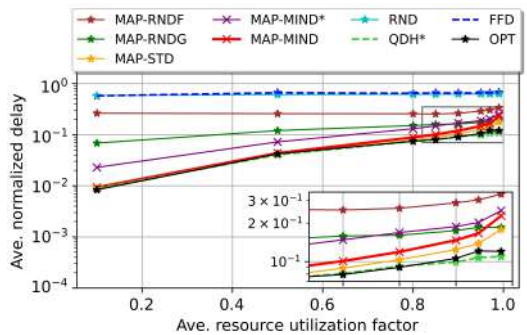

FIGURE 16. Ave. normalized delay under scenario 5 for UDC with 1-player GEMs.

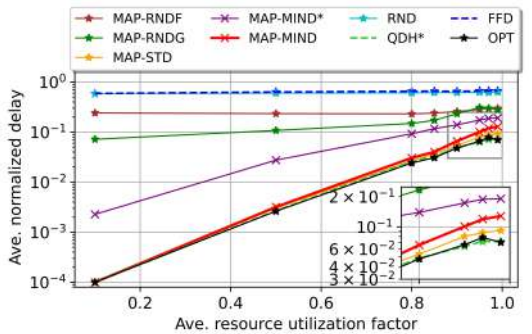

FIGURE 18. Ave. normalized delay under scenario 6 for UDC with 1-player GEMs.

MAP-based algorithms only show negligible dropping at the highest load.

Scenario 5: Varying UF with heterogeneous resources

The dropping probability for both 1-player and 2-player GEMs and the average normalized delay for 1-player GEMs are illustrated in Fig. 15 and Fig. 16, respectively.

When contrasted with a network featuring homogeneous resources, all algorithms with heterogeneous resources manifest qualitatively similar behavior regarding average normalized delay and dropping probability. As expected, in the presence of heterogeneous resources, we observe a small increase in both delays at low loads and dropping probabilities. The former observation can be attributed to the restricted resources on certain CNs, resulting in the placement of some GEMs on more distant CNs. The latter observation is a direct consequence of the former since distant nodes are more likely to exceed the maximum tolerable delay for the mentioned GEM, leading to its dropping.

Scenario 6: Multiple-resource bottleneck

The dropping probability for both 1-player and 2-player GEMs and the average normalized delay for 1-player GEMs are illustrated in Fig. 17 and Fig. 18, respectively. When comparing the results with scenario 4 (i.e., single-resource bottleneck), no significant differences are observed except at very high loads (i.e., UF $\geq$ 0.95). At these high loads, all algorithms begin to drop the GEMs more rapidly. By concentrating solely on one bottleneck during placement, another bottleneck can emerge, leading to drops. In such cases, even the OPT struggles to accommodate all the GEMs within the network.

Scenario 7: Batch-placement in an online scenario

The dropping probability and the average normalized delay for an online scenario are illustrated in Fig. 19.

In general, the decision to place the newly arrived GEMs at the end of each batch-window is based on a snapshot of GEMs, lacking foresight into upcoming GEMs in future windows. This myopic view can lead to sub-optimal global decisions over the entire simulation period. The effect should be explained by simultaneously considering two factors: (i) batch-window size Δt, and (ii) the efficiency of the algorithm. Here Δt has a paramount effect, as a larger size of collected GEMs provides more information, enabling the proposed algorithms to make more informed decisions about GEM placement and lowering the probability of dropping.

From Fig. 19 (left graph), it is evident that increasing the value of Δt reduces the dropping probability for all algorithms. On the other hand, when algorithms make locally optimized decisions based on the current window's GEMs, they may allocate resources in a way that efficiently satisfies immediate GEMs but could potentially block critical resources needed for special or delay-critical GEMs in subsequent windows. Thus, more locally optimized algorithms are likely to achieve worse results globally for the smallest windows and better results for the largest windows. This is coherent with the OPT results in Fig. 19 (left graph).

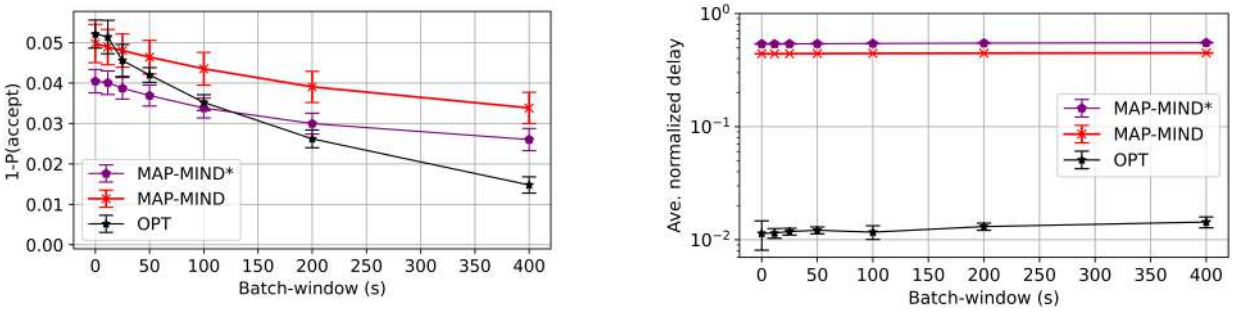

FIGURE 19. Probability of dropping (left) and ave. normalized delay (right) in UDC and under scenario 7.

Also, MAP-MIND demonstrates the same behavior, having worse results than MAP-MIND* for small windows. Considering the small structural difference between the two algorithms, investigating larger windows than shown in Fig. 19 is needed to show that the two algorithms eventually converge together and possibly change their positions as forecasted above. Regarding the delay, for all window sizes, OPT outperforms the algorithms as it aims to minimize placement delay, while MAP-MIND performs slightly better than MAP-MIND* due to slightly more dropped GEMs. It is worth reminding that, due to the nature of MAP as the first phase in our algorithms, to maximize placement, it trades delay for acceptance, leading to high acceptance at the cost of large but even acceptable delay (see Fig. 19 right graph).

On the other hand, introducing a collection window alters the incoming pattern, creating a burst of GEMs to place, which leads to raising the dropping probability. For instance, consider two distinct GEMs arriving at different times where only one specific server can serve them. If the computation of the first GEM concludes before the arrival of the second, one of them would need to be dropped if they coexist within the same window under batch placement. This scenario highlights the limitations of both window-based processing of collected GEMs and locally optimized decision-making. In the NDC case, where servers are interchangeable for GEMs, the effect of the window is limited to merely extending the average waiting time for GEMs to be served, which may result in dropping excess GEMs if Δt exceeds the maximum computation time. Conversely, in the UDC scenario, where servers are not interchangeable for GEMs (i.e., some servers are not suitable in terms of delay for some GEMs), a combination of the two aforementioned effects is observed.

Interestingly, in the current landscape of online gaming, a wait time averaging more than 60 s to start a game is likely to be considered inconvenient by players. This suggests a window size not greater than 60 s, which, according to Fig. 19, showcases MAP-MIND* as a viable choice for online scenarios given its simplicity and efficiency.

E. COMPUTATIONAL COMPLEXITY OF DIFFERENT ALGORITHMS

As a theoretical performance bound, we investigate and compare the computational complexity of our algorithms.

TABLE 4. Computational complexity and average calculated iterations for different algorithms under the scenario 1.

Algorithm	Worst-case computational complexity $\mathcal{O}()$	Ave. iterations for k-player GEM with k equal to			
		1	2	10	50
RND	$\|S\|\|N\|$	908	928	$1.58k$	$2.37k$
QDH*	$\|S\|\|N\|^2 \log \|N\|$	350	780	$2.63k$	$3.57k$
FFD	$\|S\|(\log \|S\| + \|N\|)$	$4.32k$	$4.27k$	$4.37k$	$4.72k$
MAP-RNDF	$\|S\|^2$	$23.3k$	$14.5k$	$11.1k$	$10.5k$
MAP-RNDG	$\|S\|^2$	$74.2k$	$72.1k$	$68.5k$	$64.1k$
MAP-MIND*	$\|S\|^2$	$23.2k$	$29.8k$	$43.8k$	$48.2k$
MAP-MIND	$\|S\|^3$	$201k$	$247k$	$368k$	$388k$
MAP-STD	$\|S\|^3$	$15M$	$15M$	$16M$	$17M$

The computational complexity of the process is influenced by the number of GEMs ($|S|$) and CNs ($|N|$). As the worst-case scenario, we consider the NDC scenario to maximize the search space. In such a scenario, all nodes are eligible, resulting in the highest number of potential moves and swaps.

The first phase of all MAP-based algorithms, in the worst case, results in $|S||N|$ iterations. The moving phase of MAP-MIND iterates over all GEMs and CNs, resulting in $\mathcal{O}(|S||N|)$ iterations. In the worst case, this process is repeated $|S|$ times, leading to $\mathcal{O}(|S|^2|N|)$ total steps. The swapping phase iterates over all pairs of GEMs, resulting in $\mathcal{O}(|S|^2)$ iterations that, in the worst case, can be repeated $|S|$ times. This leads to $\mathcal{O}(|S||N| + |S|^2|N| + |S|^3)$ as the complexity of MAP-MIND. In contrast, in the delay-enhancing part, MAP-MIND* iterates over all GEMs while merging the move and swap in one phase, resulting in $\mathcal{O}|S|(|N| + |S|)$ steps in the worst case. This leads to $\mathcal{O}(|S||N| + |S|^2)$ as the complexity of MAP-MIND*. It can be simply shown that the complexity of MAP-RNDF and MAP-RNDG is equal to that of MAP-MIND*. On the other hand, the MAP-STD moving phase iterates over all GEMs and CNs, resulting in $\mathcal{O}(|S||N|)$ iterations, and the swapping phase iterates over all pairs of GEMs, resulting in $\mathcal{O}(|S|^2)$ iterations. In the main iteration, the whole process can be repeated $|S|$ times in the worst case, as it can loop over all pairs of GEMs, which leads to a complexity of $\mathcal{O}(|S||N| + |S|^2|N| + |S|^3)$. Given that $|S| \gg |N|$, Table 4 reports the worst-case computational complexity for all algorithms.

We also reported the observed average number of iterations, moves, and swaps for all two-phased algorithms in Table 4. Upon comparing the two columns of Table 4,

it is interesting to note that MAP-MIND*, despite having the same worst-case complexity as MAP-RNDF and MAP-RNDG, exhibits the minimum number of iterations and the best delay results among the three. However, while it slightly underperforms MAP-MIND in terms of delay, it is significantly more efficient and faster. Conversely, while MAP-MIND and MAP-STD have similar worst-case complexities and yield comparable delay performance, MAP-MIND proves to be substantially faster in practice compared to MAP-STD. In conclusion, we anticipate that our algorithms, implemented in fast, general-purpose languages like C++, require minimal computational resources and thus incur negligible overhead when deployed in real-time cloud gaming environments.

VI. RELATED WORK

The GEM placement problem is similar to the VNF (Virtual Network Function) placement in cloud systems. The Co-Location Placement (CLP) strategy, utilized in various studies, aligns with our approach where we treat all GEMs of a multi-GEM game as a single large abstract GEM [24]–[26]. Authors in [24] justify CLP by addressing seamless handovers for mobile users moving between providers using a fog-based authentication mechanism. The work in [25] uses CLP by treating each task as an independent inseparable request, each served by a single data center. In [26], CLP is justified assuming each task in the vehicular networks context can only be offloaded to one resource.

The problem of optimal VNF placement is typically NP-hard, making optimal solutions computationally challenging. For instance, in [27], a heuristic based on Integer Linear Programming (ILP) for VNF placement was introduced, akin to our two-phase approach. While effective for larger instances, its high execution time, due to the iterative call to the ILP solver, may curtail its practical application. As a result, machine learning techniques [28]–[30] or heuristic/meta-heuristic algorithms [31]–[35] are employed to efficiently address the VNF placement challenge.

In [28], the authors presented an online VNF placement strategy for Edge Computing networks using Reinforcement Learning. This strategy diminished user delay by positioning VNFs closer to users, but it exhibited a high rejection rate. Another study in [29] integrated Deep Reinforcement Learning with Graph Neural Networks to enhance VNF placement, thereby reducing service request rejections and adapting to broader network types. Meanwhile, [30] utilized deep reinforcement learning for VNF placement to reduce the average end-to-end delay by allocating more VNFs in MEC. However, this approach did not account for maximizing VNF acceptance. The authors of [31] introduced a heuristic solution, MaxSR, which dynamically orchestrates services in 5G networks using a backtracking approach. In [32], a genetic algorithm-based heuristic was developed to minimize resource costs. Another study [33] proposed the MINI heuristic algorithm to optimize resource utilization, showing marginal improvements over another heuristic based on the Genetic Algorithm (GA) for VNF acceptance. Some works focused on offline batch request placements. For example, [34] proposed an offline simulated annealing-based heuristic for placing VNFs for delay-sensitive requests. The heuristic introduced in [35], termed Previous Window Deployment (PWD), is based on Learn and Deploy (LAD) and aims to maximize user service. Notably, none of these studies contemplated shared services among users, which corresponds to the multi-player case in our gaming scenario.

Several studies have explored VNF sharing, also better described as reusability, within the context of the VNF placement problem [36]–[44], mimicking the multi-player gaming scenario. For instance, [36] explored the VNF placement within a single physical node, allowing VNF sharing across multiple Network Services (NSs). They proposed a heuristic algorithm that prioritizes NSs on shared VNF instances, ensuring processing delays meet the required thresholds. [39] tackled the dynamic VNF placement challenge, deciding whether to migrate existing VNFs or deploy new ones to optimize VNF reuse. The objective was to enhance the network operator's profitability, and a column generation-based algorithm was suggested for this purpose. In [41], a dynamic heuristic algorithm was introduced to minimize overall network OPEX and physical resource fragmentation for the VNF placement issue, where multiple NSs can share VNF instances. Although these studies focus on VNF placement for NSs that incorporate VNF re-usability, none align with the multiplayer shared GEM challenge.

Several works have proposed heuristic-based algorithms emphasizing maximizing the number of accepted VNFs [35]–[37], [44]–[52]. For example, [45] introduced a potential game approach for VNF placement in satellite edge networks. Like our approach, they support VNF co-location, but they applied a distributed placement decision and did not consider shared VNFs. In [48], two heuristics, one greedy-based and the other Tabu search-based, were proposed to maximize profit through placement. They successfully increased the admission rate for offline placement but did not consider shared VNFs. Authors of [50] tackled the delay-aware VNF dynamic placement and routing problem by constructing a heuristic optimization data structure called VNF-splitted multi-stage edge-weight graph. Their algorithm demonstrated superior performance in average traffic acceptance rate compared to others. Like our approach, they considered a maximum tolerable delay but did not focus on minimizing the delay or considering shared VNFs. The study in [52] presented an online heuristic framework named Holu, which aimed to solve the VNF placement problem by considering the centrality of the compute nodes and the power consumption of a VNF. They showed improvement in VNF acceptance but did not consider minimizing delay. Also, the shared VNF considered in their model is distinct in nature as users share resources but do not interact.

Some works have proposed heuristic algorithms for the VNF placement problem with a focus on delay minimization [49], [53]–[56]. For instance, in [49], an

evolutionary algorithm was proposed to enhance various metrics for IoT devices' service placement. Their approach maximizes the use of the fog nodes for the placement, which in turn improves service delay and response times. However, they did not consider the shared VNF scenario. The work in [54] addressed the VNF placement problem in non-terrestrial networks by formulating it as a weighted graph-matching problem using a Linear Programming algorithm and the Hungarian-based algorithm. Their goal was to minimize delay and maximize resource utilization. Lastly, [56] introduced a genetic-based heuristic algorithm for service placement aimed at minimizing application delay. However, neither shared VNFs nor the maximization of VNF placement was targeted in these last two papers.

In summary, while the aforementioned related works have attempted to address various facets of our problem, to our knowledge, no existing research has holistically addressed all the requirements of our problem, especially concerning multiplayer GEMs. This distinction is significant, as the definitions of shared VNFs in current literature differ markedly from our approach.

VII. CONCLUSION

The research presented in this paper addresses the optimal placement of game engine modules (GEM) in cloud gaming scenarios. The proposed algorithms, MAP-MIND and MAP-MIND*, have been developed to effectively balance the maximization of the number of placed GEMs and the minimization of delay experienced by the players. We rigorously evaluated the performance of these algorithms through extensive simulations. MAP-MIND was shown to approximate the optimal solution closely, with a very small dropping probability in worst-case scenarios. Conversely, while the MAP-MIND* algorithm slightly under-perform in terms of delay compared to MAP-MIND, it offers significant advantages in terms of computation time, making it a practical alternative for real-world applications where large instances of the placement problem must be considered. We defer the extension of our proposed approach to include other cost functions, such as energy costs and computation costs from various cloud providers, to future work. Additionally, we expect that our work will inspire the development of new enhanced algorithms helped by machine-learning approaches to learn from past data in online scenarios.

REFERENCES

[1] M. Claypool, D. Finkel, A. Grant, and M. Solano, "Thin to win? network performance analysis of the OnLive thin client game system," in *ACM NetGames*, 2012.
[2] A. Bujari, M. Massaro, and C. E. Palazzi, "Vegas over access point: Making room for thin client game systems in a wireless home," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, 2015.
[3] "Google Stadia," https://stadia.google.com, accessed: 2023-10.
[4] "Amazon Luna," https://www.amazon.com/luna, accessed: 2023-10.
[5] "GeForce Cloud Gaming," https://www.nvidia.com/geforce-now, accessed: 2023-10.
[6] "Xbox Cloud Gaming," https://www.xbox.com/en-US/cloud-gaming, accessed: 2023-10.
[7] D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi, "SMASH: A distributed game engine architecture," in *IEEE ISCC*, 2016.
[8] L. De Giovanni, D. Gadia, P. Giaccone, D. Maggiorini, C. E. Palazzi, L. A. Ripamonti, and G. Sviridov, "Revamping cloud gaming with distributed engines," *IEEE Internet Computing*, vol. 26, no. 6, 2022.
[9] G. Dósa and J. Sgall, "Optimal analysis of best fit bin packing," in *Automata, Languages, and Programming*. Springer, 2014.
[10] "Red Dead Redemption II," https://www.rockstargames.com/reddeadredemption2/, accessed: 2023-10.
[11] "Horizon Forbidden West," https://www.playstation.com/it-it/games/horizon-forbidden-west/, accessed: 2023-10.
[12] "Street Fighter 6," https://www.streetfighter.com/6, accessed: 2023-10.
[13] "FIFA 23," https://www.ea.com/games/fifa/fifa-23, accessed: 2023-10.
[14] "Portal 2," https://store.steampowered.com/app/620/Portal_2/, accessed: 2023-10.
[15] "Dota 2," https://www.dota2.com/, accessed: 2023-10.
[16] "Leage of Legends," https://leagueoflegends.com/, accessed: 2023-10.
[17] "Valorant," https://playvalorant.com/, accessed: 2023-10.
[18] "Counter-Strike," https://www.counter-strike.net/, accessed: 2023-10.
[19] "Fortnite," https://www.fortnite.com/, accessed: 2023-10.
[20] "Player Unknown Battle Ground (PUBG)," https://pubg.com/en-na, accessed: 2023-10.
[21] "Fall Guys," https://www.fallguys.com/en-US, accessed: 2023-10.
[22] H.-J. Hong, D.-Y. Chen, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu, "Placing virtual machines to optimize cloud gaming experience," *IEEE Transactions on Cloud Computing*, vol. 3, no. 1, 2015.
[23] J. N. Gupta and J. C. Ho, "A new heuristic algorithm for the one-dimensional bin-packing problem," *Production planning & control*, vol. 10, no. 6, 1999.
[24] A. Ali, T. Mallick, S. Sakib, M. Hossain, Y.-D. Lin *et al.*, "Provisioning fog services to 3GPP subscribers: Authentication and application mobility," *arXiv preprint arXiv:2112.02476*, 2021.
[25] T. Sato and E. Oki, "Program file placement strategies for machine-to-machine service network platform in dynamic scenario," *IEICE Transactions on Communications*, 2020.
[26] B. Kar, K.-M. Shieh, Y.-C. Lai, Y.-D. Lin, and H.-W. Ferng, "QoS violation probability minimization in federating vehicular-fogs with cloud and edge systems," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 12, 2021.
[27] B. Addis, G. Carello, and M. Gao, "ILP-based heuristics for a virtual network function placement and routing problem," *Networks*, vol. 78, no. 3, 2021.
[28] C. R. De Mendoza, B. Bakhshi, E. Zeydan, and J. Mangues-Bafalluy, "Near optimal VNF placement in edge-enabled 6G networks," in *IEEE ICIN*, 2022.
[29] P. Sun, J. Lan, J. Li, Z. Guo, and Y. Hu, "Combining deep reinforcement learning with graph neural networks for optimal VNF placement," *IEEE Communications Letters*, vol. 25, no. 1, Jan 2021.
[30] A. Dalgkitsis, P.-V. Mekikis, A. Antonopoulos, G. Kormentzas, and C. Verikoukis, "Dynamic resource aware VNF placement with deep reinforcement learning for 5G networks," in *IEEE GLOBECOM*, 2020.
[31] M. Golkarifard, C. F. Chiasserini, F. Malandrino, and A. Movaghar, "Dynamic VNF placement, resource allocation and traffic routing in 5G," *Computer Networks*, vol. 188, 2021.
[32] N. Kiran, X. Liu, S. Wang, and C. Yin, "VNF placement and resource allocation in SDN/NFV-enabled MEC networks," in *IEEE WCNCW*, 2020.
[33] C. Zhiqi, Z. Sheng, W. Can, Q. Zhuzhong, X. Mingjun, W. Jie, and J. Imad, "A novel algorithm for NFV chain placement in edge computing environments," in *IEEE GLOBECOM*, 2018.
[34] C. S. R. M. Prabhu Kaliyammal Thiruvasagam, Abhishek Chakraborty, "Latency-aware and survivable mapping of VNFs in 5G network edge cloud," in *IEEE (DRCN)*, 2021.
[35] D. Harris and D. Raz, "Dynamic VNF placement in 5G edge nodes," in *IEEE NetSoft*, 2022.
[36] F. Malandrino, C. F. Chiasserini, G. Einziger, and G. Scalosub, "Reducing service deployment cost through VNF sharing," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, 2019.
[37] T. V. Doan, G. T. Nguyen, M. Reisslein, and F. H. P. Fitzek, "SAP: Subchain-aware NFV service placement in mobile edge cloud," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, March 2023.
[38] A. Mohamad and H. S. Hassanein, "On demonstrating the gain of SFC placement with VNF sharing at the edge," in *IEEE GLOBECOM*, 2019.

[39] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu, "On dynamic service function chain deployment and readjustment," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, 2017.
[40] P. Jin, X. Fei, Q. Zhang, F. Liu, and B. Li, "Latency-aware VNF chain deployment with efficient resource reuse at network edge," in *IEEE INFOCOM*, 2020.
[41] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, 2016.
[42] A. Mohamad and H. S. Hassanein, "PSVShare: A priority-based SFC placement with VNF sharing," in *IEEE NFV-SDN*, 2020.
[43] A. Ebrahimzadeh, N. Promwongsa, S. N. Afrasiabi, C. Mouradian, W. Li, Á. Recse, R. Szabó, and R. H. Glitho, "H-horizon sequential look-ahead greedy algorithm for VNF-FG embedding," in *IEEE NFV-SDN*, 2021.
[44] Y. Yue, B. Cheng, M. Wang, B. Li, X. Liu, and J. Chen, "Throughput optimization and delay guarantee VNF placement for mapping SFC requests in NFV-enabled networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, 2021.
[45] X. Gao, R. Liu, and A. Kaushik, "Virtual network function placement in satellite edge computing with a potential game approach," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, 2022.
[46] A. Hmaity, M. Savi, L. Askari, F. Musumeci, M. Tornatore, and A. Pattavina, "Latency-and capacity-aware placement of chained virtual network functions in FMC metro networks," *Optical Switching and Networking*, vol. 35, 2020.
[47] C. Morin, G. Texier, C. Caillouet, G. Desmangles, and C.-T. Phan, "VNF placement algorithms to address the mono and multi-tenant issues in edge and core networks," in *IEEE CloudNet*, 2019.
[48] N. Promwongsa, A. Ebrahimzadeh, R. H. Glitho, and N. Crespi, "Joint VNF placement and scheduling for latency-sensitive services," *IEEE Transactions on Network Science and Engineering*, vol. 9, 2022.
[49] C. Liu, J. Wang, L. Zhou, and A. Rezaeipanah, "Solving the multi-objective problem of IoT service placement in fog computing using cuckoo search algorithm," *Neural Processing Letters*, vol. 54, no. 3, 2022.
[50] L. Liu, S. Guo, G. Liu, and Y. Yang, "Joint dynamical VNF placement and SFC routing in NFV-enabled SDNs," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, Dec 2021.
[51] A. Mohamad and H. S. Hassanein, "Prediction-based SFC placement with VNF sharing at the edge," in *IEEE LCN*, 2022.
[52] A. Varasteh, B. Madiwalar, A. Van Bemten, W. Kellerer, and C. Mas-Machuca, "Holu: Power-aware and delay-constrained VNF placement and chaining," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, 2021.
[53] C. Richard, A. Christos, and P. P. Dimitrios, "Dynamic, latency-optimal VNF placement at the network edge," in *IEEE INFOCOM*, 2018.
[54] Y. Yue, X. Tang, W. Yang, X. Zhang, Z. Zhang, C. Gao, and L. Xu, "Delay-aware and resource-efficient VNF placement in 6G non-terrestrial networks," in *IEEE WCNC*, 2023.
[55] M. Guido, S. Riccardo, Y. Jalolliddin, and K. Adlen, "Formally verified latency-aware VNF placement in industrial internet of things," in *IEEE WFCS*, 2018.
[56] N. Sarrafzade, R. Entezari-Maleki, and L. Sousa, "A genetic-based approach for service placement in fog computing," *The Journal of Supercomputing*, vol. 78, no. 8, 2022.

IMAN LOTFIMAHYARI received his BSc and MSc in Electronics Engineering from IAU University in 2003 and 2007, respectively. In March 2020, he received his second MSc in Telecommunication Engineering from Politecnico di Torino, Italy, and joined the Telecommunication Networks Group of Politecnico di Torino as a Ph.D. student. His current research interests involve programmable data planes for SDN, blockchains, and cloud computing.

LUIGI DE GIOVANNI received his M.S. and Ph.D. in Computer and Systems Engineering from Politecnico di Torino, Italy, in 1999 and 2004 respectively. He is an Associate Professor of Operations Research at Università degli Studi di Padova, Italy. His research interests focus on Combinatorial Optimization, Mathematical Programming, Meta-heuristics applications for telecommunication and logistic network design, scheduling, and airport/air traffic management.

DAVIDE GADIA received his M.Sc. and Ph.D. in Computer Science from the University of Milan, Italy, in 2003 and 2007 respectively. He is an Associate Professor at the Department of Computer Science of the University of Milan. His research interests focus on Video Game Programming, Procedural Content Generation for Computer Graphics and Video Games, Game Engine architectures, and VR. He is affiliated with PONG (Playlab fOr inNovation in Games) Lab.

PAOLO GIACCONE received the Dr.Ing. and Ph.D. degrees in telecommunications engineering from the Politecnico di Torino, Italy, in 1998 and 2001, respectively. He is a Full Professor in the Department of Electronics, Politecnico di Torino. His main area of interest is the design of optimal control and resource allocation algorithms in next-generation networks.

DARIO MAGGIORINI received his M.S. and Ph.D. in Computer Science from the University of Milan in 1997 and 2002 respectively. He is an Associate Professor at the Department of Computer Science of the University of Milan and co-founder of the PONG (Playlab fOr inNovation in Games) lab. He worked on QoS for multi-service IP-based networks, large network scalability, multimedia transmission, mobile services, and opportunistic networks. Since 2011, he focused his expertise on software architectures for entertainment applications.

CLAUDIO E. PALAZZI received his M.S. degree in Computer Science from UCLA in 2005, his Ph.D. degree in Computer Science from UniBO in 2006, and his Ph.D. degree in Computer Science from UCLA in 2007. He is an Associate Professor of Computer Science at the Università degli Studi di Padova, Italy. His research interests are the design and analysis of communication protocols for wired and wireless networks, Internet architectures, and mobile users, with an emphasis on mobile applications and multimedia entertainment.