Adaptive Partitioning for Large-Scale Graph Analytics in Geo-Distributed Data Centers

Amelie Chi Zhou*, Juanyun Luo*, Ruibo Qiu*, Haobin Tan*, Bingsheng He† and Rui Mao*
*College of Computer Science and Software Engineering, Shenzhen University
†Department of Computer Science, National University of Singapore

***Abstract*—Graph partitioning is an important problem to the performance and cost optimization of graph analytics in geo-distributed environments. Modern hybrid-cut model is expected to obtain better performance and cost optimizations than traditional partitioning models, but can further complicate geo-distributed graph partitioning which is already a challenging problem due to large graph sizes and network heterogeneities of geo-distributed DCs. Existing studies usually adopt heuristic-based methods to achieve fast partitioning for large graphs, which unfortunately sacrifices optimization effectiveness. Further, graph structures of many applications can change at various frequencies. Dynamic partitioning methods usually focus on achieving low latency to quickly adapt to changes, which may again sacrifice partitioning effectiveness. Also, such methods are not aware of the dynamicity of graphs and can over sacrifice effectiveness for unnecessarily low latency.**

In this paper, we propose RLCut, which uses Reinforcement Learning (RL) to help taming the complexity of the problem. Specifically, RLCut uses multi-agent learning which is more efficient than single agent RL and incorporates a sampling based optimization to adaptively control the training process to satisfy required trade-off between partitioning effectiveness and efficiency according to graph dynamicity. Experiments using real cloud DCs and real-world graphs show that, compared to state-of-the-art static partitioning methods, RLCut improves the performance of geo-distributed graph analytics by 10%-100% with comparable overhead. When users tolerate longer partitioning overhead, we can further improve the performance by up to 43%. With varying graph changing frequencies, RLCut can improve the performance by up to 60% compared to state-of-the-art dynamic partitioning.

I. Introduction

Recently, many graph analytics applications involve analyzing large sizes of data spread in multiple geographically distributed (geo-distributed) data centers (DCs). For example, social networks usually contain user data generated and stored geo-distributedly, according to where the users are located. Such applications are intrinsically geo-distributed, which makes them extremely expensive to run in a centralized manner due to the high inter-DC data movement cost [1], [2]. On the other hand, when performing graph analytics in the geo-distributed manner, it is less efficient to run an application as it is (i.e., graph data are only located at where they are generated), due to the heterogeneities in network bandwidths and graph data transfer sizes between different pairs of DCs. By moving graph data at offline time, we can possibly reduce the runtime inter-DC data communication latency and cost. That is, we should consider the geo-distributed graph (re)partitioning problem to improve application performance and cost efficiency.

Most existing graph partitioning methods take load balancing and communication minimization as optimization goals [3], [4], [5], [6], [7], [8], which can lead to good graph analytics performance in traditional systems with homogeneous network bandwidths. However, this is not the case in geo-distributed environments. As shown in Section II, the uplink/downlink bandwidths of a single DC can be highly heterogeneous and the bandwidths of different DCs also differ a lot. Thus, a load-balanced graph partitioning solution does not necessarily lead to good graph analytics performance in geo-distributed DCs. Due to problem complexities in graph sizes and network heterogeneities, it is non-trivial to obtain good partitioning results for large graphs in a short time. Recently, a few graph partitioning methods have been proposed to optimize graph analytics performance in heterogeneous environments [9], [1], [10]. However, these methods are not suitable for our graph (re)partitioning problem in geo-distributed DCs due to the following reasons.

First, existing studies usually adopt heuristic-based methods such as streaming-based partitioning to achieve fast partitioning for large graphs, which in turn sacrifices optimization effectiveness. For example, assume a graph has N vertices that need to be assigned to M DCs in the partitioning problem, the solution space of the problem is $O(M^N)$ while the search space of the one pass streaming method is only $O(M \times N)$.

Moreover, most existing algorithms adopt vertex-cut, namely partitioning graphs through vertices, which is efficient for real-world graphs [3]. However, as vertex-cut generates replicas for vertices which require inter-replica traffic to exchange vertex information, it can lead to high inter-DC data communication if partitioned inappropriately. The hybrid-cut model uses differentiated partitioning for vertices with high/low-degrees and thus can avoid introducing inter-replica communication for low-degree vertices. Our experiments show that, hybrid-cut can obtain up to 87% less inter-DC data communication than vertex-cut (see Figure 2 for details) and thus is more suitable for graph partitioning in geo-distributed environments. However, the differentiated computation and partitioning of the hybrid-cut model further complicate the problem and make it even harder for heuristic-based methods to find a good solution in a short time.

What makes the situation worse is that, many real-world graphs are dynamic with updates on vertices, edges or both

at various frequencies. For example, the Twitter graph can receive thousands of updates per second at peak when there's a hot topic and receive much less updates at other times [11]. Existing graph partitioning methods designed for dynamic graphs are mostly best-efforts methods that focus on achieving low latency to maintain good graph partitioning on the fly along with graph changes. Partitioning effectiveness is usually sacrificed for low latency in such methods, while the effectiveness has already been compromised to achieve fast partitioning for large graphs under the complicated hybrid-cut model as discussed above. The update frequency of dynamic graphs are seldomly taken into consideration by existing dynamic partitioning methods and partitioning effectiveness could be over sacrificed for the low latency that's not even necessary. A more preferred way is to have an adaptive partitioning method that can automatically adjust the trade-off between partitioning overhead and quality according to the dynamicity of graphs.

In this paper, we propose to use machine learning to help taming the complexity of efficient graph partitioning based on hybrid-cut in geo-distributed DCs. Specifically, we propose an adaptive method named *RLCut* based on Reinforcement Learning (RL). RL has been widely adopted by recent studies for complicated decision-making problems [12], [13], [14], [15], [16] and has the ability of optimizing long-term reward in dynamic environments. Thus, it can satisfy our wish to be *adaptive* on the trade-off between optimization effectiveness and overhead. Considering the dynamicity of graphs, RLCut takes the optimization overhead as a constraint to adaptively tune its optimization process, with the objective of optimizing the performance of geo-distributed graph analytics while satisfying the budget constraint on inter-DC data transfer cost. The overhead constraint could be set by an expert user or learned during graph changes. We mainly address the following challenges in RLCut.

First, to apply RL for our geo-distributed graph partitioning problem, we adapt the hybrid-cut model and formulate the differentiated partitioning as a unified state of the RL training environment. Second, RL has the problem of large optimization overhead, which is more severe when partitioning large graphs. To address this problem, we utilize multi-agent learning [17] which is much cheaper and more efficient than single agent, and propose two optimization techniques, namely batching and straggler mitigation, to reduce the overhead of RLCut while preserving good performance optimization result. Finally, to adaptively trade off between optimization effectiveness and efficiency for dynamic graphs, we propose a sampling-based technique to adaptively decide the number of agents participating in the training of RLCut considering the required optimization overhead.

We compare RLCut with six state-of-the-art graph partitioning methods based on different partitioning models for both static and dynamic graphs [1], [3], [6]. Experiments using real geo-distributed cloud DCs and real-world graph datasets show that, RLCut can improve the performance optimization result over state-of-the-art comparisons by 10%-100% with comparable optimization overhead. For example, for large

TABLE I
UPLINK/DOWNLINK BANDWIDTHS OF CC2.8XLARGE INSTANCES FROM AMAZON EC2 REGIONS TO THE INTERNET. PRICES ARE FOR UPLINKS.

	US East	AP Singapore	AP Sydney
Uplink Band. (GB/s)	0.52	0.55	0.48
Downlink Band. (GB/s)	2.8	3.5	2.5
Price ($/GB)	0.09	0.12	0.14

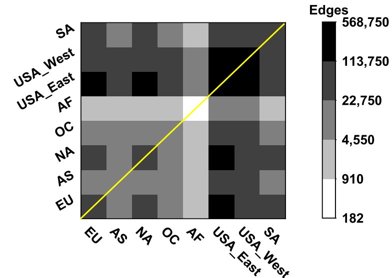


Fig. 1. Number of edges between DCs in the Twitter graph, where vertices are located in eight different DCs.

graphs such as it-2004 [18] with over one billion edges, we can obtain a good partitioning result in six minutes. When users tolerate longer partitioning overhead, RLCut is able to further improve the performance of geo-distributed graph analytics by up to 43%. When varying the graph changes at different frequencies, RLCut can improve the performance optimization results by up to 60% compared to state-of-the-art dynamic partitioning method [7].

The remainder of this paper is organized as follows. Section II introduces the background and related work. Section III formulates the problem studied in this paper. Section IV and Section V present details of our RL-based adaptive graph partitioning approach. Section VI presents the experimental results and Section VII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. Geo-Distributed Graph Analytics

Many graph applications, such as social networks, involve analyzing large sizes of data generated and stored in multiple geo-distributed DCs. Although one way of processing such applications is to move all data into the same DC, it is not always viable to do so. On one hand, moving large amount of data across DCs is very pricy and slow, which could violate the SLA requirements of latency-sensitive applications. On another hand, due to data privacy concerns, it is sometimes prohibited to move graph data out of their original DCs. In such cases, graph analytics has to be performed in a geo-distributed manner. Performance optimization of geo-distributed graph analytics is challenging, mainly due to the following reasons.

First, the wide area network (WAN) across DCs is costly and highly heterogeneous [19]. We have evaluated the WAN bandwidths of multiple geo-distributed service regions of Amazon EC2 cloud. Table I shows the uplink/downlink bandwidths

from/to three Amazon EC2 regions to/from the WAN using cc2.8xlarge instances. We have the following observations.

- Heterogeneous bandwidths within and across DCs: Within a single DC, the downlink bandwidths of the three regions are several times higher than their uplink bandwidths. Across different regions, the uplink and downlink bandwidths of the Singapore region are 17% and 40% higher than those of the Sydney region, respectively. Network heterogeneities make the performance optimization for geo-distributed graph analytics more challenging.
- High cost of WAN usage: Data transfer within the same region is usually free of charge on most public clouds, while sending data from cloud DCs to the Internet can be pricy. This motivates us to reduce the WAN usage as much as possible for geo-distributed graph analytics.

Second, inter-DC data transfer plays an important role in geo-distributed graph analytics. We study the distributions of high degree vertices ($> 10,000$ followers) in the Twitter graph using real geographic locations of users [20]. Specifically, we cluster user locations into eight DCs, including South America, USA West, USA East, Africa, Oceania, North America, Asia and Europe. Figure 1 shows the number of edges between different pairs of DCs, where the diagonal cells represent intra-DC edges and the rest represent inter-DC edges. The number of edges can reflect the amount of data transfer between DCs. We find that over 75% of all edges are inter-DC edges, which means that the optimization of inter-DC data communication plays an important role in the performance and cost optimizations of geo-distributed data analytics.

B. Graph Partitioning Methods

Large graph analytics applications are usually executed in a distributed manner, where graphs are partitioned onto multiple machines for parallel computation. In the geo-distributed environment, a graph is *naturally partitioned* across DCs. However, this initial partitioning only reflects data distribution nature and does not guarantee good performance of geo-distributed graph analytics. To optimize the performance, it is crucial to re-partition the graph considering network heterogeneities and application requirements. Different graph applications favor different graph partitioning methods. We discuss from the following two aspects.

Different partitioning models. There are three commonly used graph partitioning models, including edge-cut [21], [4], vertex-cut [3], [22] and hybrid-cut [6]. *Edge-cut* distributes vertices to multiple machines and creates replicated edges and vertices (e.g., mirrors) to form a locally consistent graph state in each machine. *Vertex-cut* distributes edges to multiple machines and only needs to replicate vertices in each machine. Thus, vertex-cut is more efficient than edge-cut for natural graphs, where a small number of vertices have very high degrees and can cause high communication cost and load imbalance if cut through edges. However, as vertex-cut treats high-degree and low-degree vertices equally, it can lead to large replication factors if partitioned inappropriately [6].

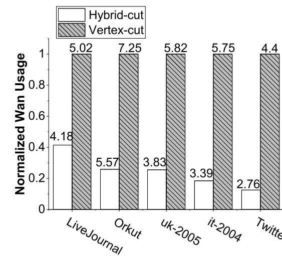


Fig. 2. Normalized WAN usage of PageRank on geo-distributed Amazon EC2 regions using different partitioning methods. Numbers on the bars represent replication factors.

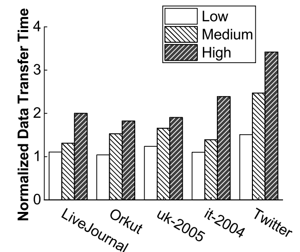


Fig. 3. Inter-DC data transfer time of PageRank on geo-distributed DCs with low/medium/high heterogeneities. Results are obtained by Ginger normalized to that of RLCut.

Hybrid-cut introduces differentiated partitioning for high/low-degree vertices. It adopts low-cut to distribute low-degree vertices along with their in-edges to multiple machines, and uses high-cut to distribute all in-edges of high-degree vertices.

Compared to vertex-cut, hybrid-cut can achieve much lower replication factors (λ) for vertices, which is important in geo-distributed DCs as the inter-DC data communication between replicas can cause poor performance and high cost to geo-distributed graph analytics. We compare the balanced p-way vertex-cut [3] and hybrid-cut [6] using five real-world graphs and the PageRank algorithm on geo-distributed Amazon EC2 regions (see Section VI for detailed experimental setup). Figure 2 presents the normalized WAN usage and vertex replication factors of the compared methods, which show the superiority of hybrid-cut over vertex-cut on reducing replication factor and communication cost.

Static vs. Dynamic graphs. Many graph partitioning methods have been proposed for static graphs [21], [3], where graph structure does not change much and the main focus is to improve the optimization effectiveness on performance, load balance, etc. Dynamic graphs, on the other hand, have frequent changes on the graph structure. For example, the Twitter graph can receive thousands of updates per second which may impact the trending topic analytics result of recommendation systems [11]. Graph partitioning methods have to quickly adapt to the changes in graph structures to guarantee good performance of the analytics jobs. Existing partitioning methods for dynamic graphs usually have low latency or good adaptivity to maintain good partitioning when facing graph changes. For example, many lightweight graph repartitioning methods [7], [23], [24], [25] have been proposed to efficiently adapt graph partitioning in the face of changes by incrementally migrating vertices/edges among partitions using certain heuristics. Streaming-based graph partitioning methods such as Fennel [5] can assign newly added vertices/edges "on-the-fly" without adapting existing partitions and thus may lead to poor application performance. Leopard[26] and GR-DEP [27] proposed to use streaming-based methods to obtain fast initial solutions and improve upon those solutions by reassigning vertices/edges among partitions. Fan et. al. [28] proposed to incrementalize batch partitioners for dynamic

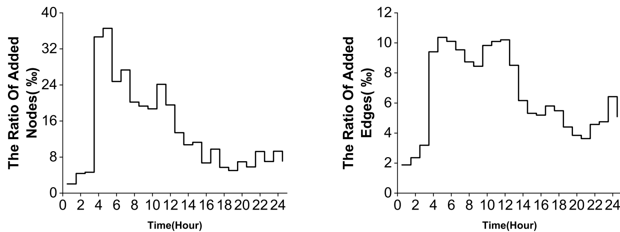


Fig. 4. The ratio of added nodes and edges to the Stack Overflow website data during September 15th, 2008.

graphs with bounded partition quality.

Clearly, partitioning methods proposed for static and dynamic graphs have very different design goals. To the best of our knowledge, no existing graph partitioning method can be adaptively applied to both static and dynamic graphs while achieving good partitioning effectiveness/efficiency as needed.

C. Motivation

Based on the above discussions, we have summarized the following needs that motivate us to design a new partitioning method for geo-distributed graph analytics.

Optimality. Inter-DC data communication plays an important role to the performance optimization of geo-distributed graph analytics. However, due to the heterogeneity of geo-distributed network environment and the large sizes of real graphs, it is hard to obtain a good solution using traditional heuristic-based methods.

To demonstrate this challenge, we simulate three geo-distributed environments with Low, Medium and High network heterogeneities, where Medium simulates the real Amazon EC2 environment described in Section VI. In Low, all DCs have the same uplink/downlink bandwidths. In High, we set the bandwidths of half the DCs to 50% of their original bandwidths to increase the heterogeneity. We compare the effectiveness of state-of-the-art heuristic-based partitioning method Ginger [6] with our proposed solution RLCut. Figure 3 shows the optimized inter-DC data transfer time of PageRank algorithm using five real-world graphs obtained by Ginger normalized to that of RLCut. Clearly, Ginger obtained higher inter-DC data transfer time compared to RLCut when the network is more heterogeneous and when the graph size gets larger. Thus, it is necessary to design a new method that can handle the high complexity of large-scale graph partitioning in geo-distributed DCs.

Adaptivity. Effectiveness and efficiency are two contradictory optimization goals of graph partitioning algorithms. Usually we have to sacrifice one in exchange of the other. For example, partitioning algorithms for static graphs often aim at obtaining good optimization effectiveness with a relatively high overhead while partitioning for dynamic graphs has to sacrifice certain effectiveness to achieve low overhead. This is especially true in the geo-distributed DCs, where graphs are extremely large and it is hard to achieve low-latency partitioning while preserving good effectiveness.

On another hand, the changes of graph structures are not stable. For example, we study the graph structure change of the Stack Overflow temporal network [29] using one day data as shown in Figure 4. We observe that the maximum ratio of added vertices and edges per hour can be five to ten times higher than the minimum value, which shows that the dynamicity of a graph is actually changing along the time. Thus, it is necessary to adaptively balance between partitioning effectiveness and efficiency according to graph changes. Existing solutions are developed towards either one of the two ends, thus do not satisfy the need.

III. Problem Formulation

A. Assumptions

We study how to efficiently and adaptively partition a large graph onto multiple geo-distributed DCs. The graph data are generated and stored geo-distributedly. We have the following assumptions. First, graph data are not replicated initially. When the location of a vertex optimized by the partitioning is different from its initial location, we replicate vertex data to the optimized location. Second, there are sufficient computation resources in each single DC, and *inter-DC data communication is the performance bottleneck of geo-distributed graph analytics*. This assumption is valid in the geo-distributed environment since the WAN bandwidth is much more scarce than the computation resources such as CPU and memory. Third, DCs are connected with a congestion-free network and the bottlenecks of the network are only from the uplinks/downlinks of DCs [30]. This assumption is based on the observation that many datacenter owners are expanding their services world-widely and are very likely to build their own private WAN infrastructure [31]. Finally, grounded on the pricing scheme of most public clouds such as Amazon EC2 and Windows Azure, we assume that only uploading data from a DC to WAN is charged.

B. Problem Definition

Consider a graph $G(V, E)$ with input data stored in M geo-distributed DCs, where V is the set of vertices and E is the set of edges. Each vertex v ($v = 0, 1, \dots, |V| - 1$) has an initial location $L_v \in [0, 1, \dots, M-1]$, which is where the input data of v is stored. Denote the input data size as d_v for vertex v. We consider a dynamic graph as regularly inserting new vertices V' and edges E' into the original G. Given a time window, if a dynamic graph has a larger number of vertices (edges) in V' (E'), we say this graph has more frequent updates.

We adopt the hybrid-cut model for graph partitioning, which categorizes vertices in V into low-degree and high-degree with a pre-defined threshold θ. Each vertex v with in-degree no less than θ is classified as high-degree and has $H_v = 1$. Otherwise, v is classified as low-degree and has $H_v = 0$. Coping with hybrid-cut, we adopt the differentiated vertex computation model of PowerLyra [6] for graph processing. Specifically, low-degree vertices are computed locally to avoid communication and the computed vertex data are transferred

to all mirrors for synchronization. The computation of high-degree vertices follow the traditional GAS model [3] to achieve good parallelism. GAS model has three stages including *Gather*, *Apply* and *Scatter*. In the gather stage, mirrors gather data from local neighbors and send the gathered data to the master. In the apply stage, the master updates vertex data locally and sends the updated data to all mirrors. In the scatter stage, all mirrors activate their local neighbors.

Performance formulation. According to our second assumption, the performance of geo-distributed graph analytics can be modeled using the inter-DC data transfer time during graph execution. There are mainly two types of inter-DC network traffic, namely the input data movement traffic before graph execution and the runtime traffic during graph execution. We first study the second type of inter-DC network traffic to model the performance of geo-distributed graph analytics. Specifically, the runtime inter-DC network traffic mainly comes from the data synchronization of low-degree vertices and the gather and apply stages of high-degree vertices. For a unified representation of vertex computation, we consider the data synchronization of low-degree vertices happen at the apply stage. Then, for a given iteration i of graph analytics and a vertex v, each mirror in DC r sends aggregated data of size $g_v^r(i)$ to the master of v in the gather stage and the master sends the combined data of size $a_v(i)$ to each mirror to update the vertex data in the apply stage. To simplify the calculation of data transfer time, we assume there is a global barrier between the gather stage and the apply stage. Thus, the inter-DC data transfer time in iteration i can be formulated as the sum of the data transfer times in gather and apply stages. In each DC, the data transfer finishes when it is finished on both uplink and downlink. We formulate the inter-DC data transfer time as below.

$$T(i) = T_G(i) + T_A(i) = \max_r T_G^r(i) + \max_r T_A^r(i) \quad (1)$$

$$T_G^r(i) = max(\frac{\sum_v I_v^r H_v \sum_{k \in R_v} g_v^k(i)}{D_r}, \frac{\sum_v (1-I_v^r) H_v g_v^r(i)}{U_r}) \quad (2)$$

$$T_A^r(i) = max(\frac{\sum_v I_v^r a_v(i)|R_v|}{U_r}, \frac{\sum_v (1-I_v^r) a_v(i)}{D_r}) \quad (3)$$

where H_v indicates whether vertex v is high-degree ($H_v = 1$) or not ($H_v = 0$). I_v^r indicates whether the replica of vertex v in DC r is the master ($I_v^r = 1$) or not ($I_v^r = 0$). R_v is the set of DCs containing replicas of v and initially is empty. U_r and D_r are the uplink and downlink bandwidths of DC r, respectively.

Cost formulation. The inter-DC data communication cost can be calculated as the sum of input data movement cost and runtime data transfer cost. Denote the data movement cost as C_{mv} and use M_v to represent whether the master location of vertex v equals to L_v ($M_v = 0$) or not ($M_v = 1$), we have:

$$C_{mv} = \sum_v M_v d_v P_{L_v} \quad (4)$$

The runtime data transfer cost is calculated as the sum of the cost spent on data uploading during the gather and apply stages. Denote the unit price of uploading data from DC r to the Internet as P_r, we have:

$$C_{rt}(i) = \sum_v \sum_{r \in R_v} P_r[I_v^r a_v(i)|R(v)| + (1 - I_v^r) H_v g_v^r(i)] \quad (5)$$

Overall formulation. Based on the above analysis, we formulate our geo-distributed graph partitioning problem as the following constrained optimization problem, where B is the budget on inter-DC data communication cost.

$$\min T(i) \quad (6)$$

s.t.
$$C_{mv} + \sum_i C_{rt}(i) \leq B \quad (7)$$

Note that the cost and performance objectives can be contradictory with each other. Thus, it is more complicated to find a good graph partitioning solution for our problem than existing studies which considers only one of the two objectives [2], [1]. To obtain a good solution in reasonable time, we propose an adaptive algorithm based on RL to trade-off optimization effectiveness and efficiency. We introduce the design details of our algorithm in the next section.

IV. RL-based Graph Partitioning for Geo-Distributed DCs

We use Learning Automata (LA), a reinforcement based learning method, to solve the geo-distributed graph partitioning problem based on hybrid-cut. LA provides an ideal basis for building multi-agent learning, which is more efficient than single-agent for our large-scale and complicated optimization problem. To apply LA, we have to answer the following questions:

1) How do we model the geo-distributed graph partitioning problem as a Markov decision process (MDP) and solve it using LA?
2) How do we design key components of LA, such as action selection and score function, to train each agent effectively and to find a good partitioning solution?
3) The training overhead of reinforcement-based online learning methods are usually high. How do we adaptively optimize the overhead to make the RL-based partitioning method feasible for large geo-distributed graphs with various update frequencies?

These questions are key to the effectiveness and efficiency of RLCut. In the following, we introduce the details of our solutions to the three questions.

A. Reinforcement Learning

Reinforcement Learning (RL) is a class of learning approaches in which an agent interacts with an *environment*. At each time step k, the agent observes some *state* s_k and takes an *action* a_k. The action changes the state of the environment to s_{k+1} and the agent receives a *reward* r_k as feedback. The state transitions and rewards are stochastic and assumed to be a Markov process. RL training proceeds in time steps. Each step k consists of a sequence of (state, action, reward) observations,

i.e., (s_k, a_k, r_k). The goal of RL training is to find an action-selection *policy* that maximizes the total reward $\mathbb{E}[\sum_{k=0}^{T} r_k]$, where T is the training length.

Geo-distributed graph partitioning using hybrid-cut is a complicated problem, mainly due to the network heterogeneities of the geo-distributed environment and the differentiated computation and partitioning of the hybrid-cut model. It is too expensive to use single agent RL to handle the problem. Multi-agent Reinforcement Learning (MARL) [17], which is cheaper and more efficient than single agent RL, is a better option for our problem. Learning automata [32], [16] belongs to the family of RL and has been viewed as an ideal basis to build multi-agent learning algorithms.

A learning automaton is a probabilistic decision-making unit situated in a unknown environment [32] that learns the optimal action through repeated interactions with its environment. The actions are chosen according to a specific probability distribution which is updated based on the environment response the automaton obtains by performing a particular action. Generally, a learning automaton can be defined using the quadruple $[A(n), P(n), R(n), T]$, where n is the current training step. $A(n) = \{a_1, a_2, ..., a_m\}$ is the action space. $P(n) = \{p_1(n), p_2(n), ..., p_m(n)\}$ is the probability distribution of all actions. $R(n) = \{r_1(n), r_2(n), ..., r_m(n)\}$ and $r_i(n) \in \{0, 1\}$ is the reinforcement signal where 0 means reward and 1 means punishment. $P(n + 1) = T(A(n), P(n), R(n))$ is the probability update function.

In the nth training step, if action $a_i(n)$ receives reward signal, probability distribution is updated as follows:

$$p_j(n+1) = \begin{cases} p_j(n) + \alpha(1 - p_j(n)), & j = i \\ p_j(n)(1 - \alpha), & j \neq i \end{cases} \tag{8}$$

Otherwise, if action $a_i(n)$ receives penalty signal, probability distribution is updated as follows:

$$p_j(n+1) = \begin{cases} p_j(n)(1 - \beta), & j = i \\ p_j(n)(1 - \beta) + \dfrac{\beta}{m-1}, & j \neq i \end{cases} \tag{9}$$

where in Equation 8 and 9, α and β are reward and penalty parameters, respectively.

B. Applying RL Model

To adopt RL for our graph partitioning problem, we first have to model our problem using MDP and define its important concepts for our problem, including *state*, *action* and *policy*.

On analyzing the partitioning rules of hybrid-cut, we find that in-edges of a low-degree vertex v are assigned to the same DC as the master of v. Each in-edge e of a high-degree vertex is assigned to the same DC as the master of the vertex on the other end of e. Thus, once the master locations of all vertices are determined, we have a fixed partitioning plan. Thus, the environment can be described purely using master vertex locations and we define $state = \{L_1, L_2, \cdots, L_{|V|}\}$, where L_i represents the master location of vertex i in the current environment. With this state definition, we can uniquely construct a graph partitioning plan in three steps: 1) assigning

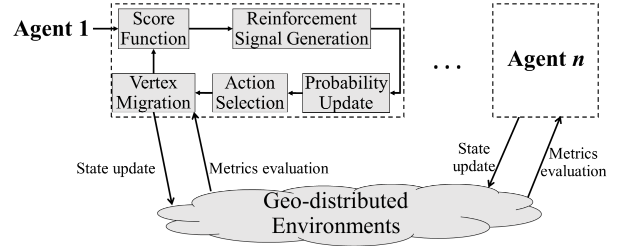


Fig. 5. Design overview of our multi-agent reinforcement learning algorithm.

all vertices according to the current state; 2) assigning all edges according to the rules of hybrid-cut and 3) creating mirrors for vertices as needed.

We assign a learning agent for each vertex in the graph and the agents take actions to assign vertices onto different DCs. Thus, we define $action = \{DC_1, DC_2, \cdots, DC_M\}$, so that each agent can choose one from the M DCs in each training iteration to assign its vertex. Although we can also assign one agent for multiple vertices, the solution space of each agent increases exponentially with the number of vertices per agent and each agent needs more time on exploration. Given a limited partitioning overhead, this could harm the effectiveness of RL training. We define $policy_s = \{p_{a_1}, p_{a_2}, \cdots, p_{a_M}\}$, which gives the probability distribution of choosing different actions under a given state s. Initially, $p_{a_i} = \frac{1}{M}, \forall i \in \{1, \cdots, M\}$.

Based on these definitions, we propose a multi-agent reinforcement learning algorithm to solve our hybrid-cut based geo-distributed graph partitioning problem. Figure 5 shows the overall structure of our algorithm. In each training iteration, each agent goes through the five steps in the dashed box, including score function computation, reinforcement signal generation, probability update, action select and vertex migration. We introduce details of these steps in the next subsection.

C. Design Details of LA

In each training iteration, each agent performs the following five steps to find a good DC for each vertex.

1) Score function: In this step, each agent computes a score for its vertex in each DC, in order to guide the action selection in later steps. The score for vertex v in DC i is calculated as:

$$Score_v^i = tw * \frac{T_l - T_a^i}{T_l} + cw * \frac{C_l - C_a^i}{C_l} * \delta(C_l - B) \tag{10}$$

where T_l represents the optimized inter-DC data transfer time in the last iteration, calculated using Equation 1. T_a^i represents the optimized inter-DC data transfer time when moving vertex v to DC i in the current iteration. Similarly, C_l and C_a^i represent the optimized cost of inter-DC data transfer in the last iteration and in the current iteration when moving vertex v to DC i, respectively, calculated using Equation 4 and 5. $\delta(x)$ is a boolean function which equals to 1 if $x > 0$ and 0 if otherwise.

We use two parameters tw and cw to trade-off the importance of performance and cost objectives in score function computation. During different stages of the training, we have different wishes in exploring the solution space. At early

stages, we prefer to explore as much as possible to find better opportunities in optimizing the performance and cost objectives. At latter stages, we have to take the budget constraint into consideration to guarantee finding feasible solutions at the end of the training. Thus, we define the tw and cw weights adaptively according to the current iteration $iter$. Specifically, we have $cw = \frac{iter}{Max_Iter}$ and $tw = 1 - cw * \delta(C_l - B)$. That is, we try to achieve better optimization goal by focusing on optimizing the performance of geo-distributed graph analytics when the training is in early iterations and when the cost is lower than the budget.

2) Reinforcement signal generation: After calculating the score for different DCs, each agent generates reinforcement signal for the last selected action accordingly. We define ρ_v as the DC that has the highest score for vertex v and S_v^i as the reinforcement signal of vertex v for DC i. The value of S_v^i is 0 or 1, which corresponds to reward and penalty signals, respectively. Specifically, as shown in the equation below, we give reward to ρ_v and penalty to all the other DCs for each vertex v.

$$S_v^i = \begin{cases} 0, & \text{if } i = \rho_v \\ 1, & \text{if } i \neq \rho_v \end{cases} \tag{11}$$

3) Probability update: In this step, each agent updates the probability distribution of choosing different actions for its vertex, in order to guide the action selection in the next step. As we know that DC ρ_v receives the reward signal for vertex v, the learning agent of v uses the following function to update the probability distribution.

$$P_v^j(n+1) = \begin{cases} P_v^j(n) + \alpha * (1 - P_v^j(n)), & \text{if } j = \rho_v \\ P_v^j(n) * (1 - \alpha), & \text{if } j \neq \rho_v \end{cases} \tag{12}$$

where $P_v^j(n)$ is the probability value defined for the action of choosing DC_j for vertex v at step n and α is the reward parameter. This function increases the probabilities of actions receiving reward signals and decreases the probabilities of other actions.

In standard learning automata, the probability distribution is also updated with the penalty signals (i.e., decrease the probabilities of actions receiving penalty signals and increase the probabilities of other actions). This method although can better explore the solution space, leads to slow convergence. Figure 6 shows the optimized inter-DC data transfer time of our method with penalty update normalized to that of without penalty update using 10 steps of training. The figure indicates that the inter-DC data transfer time result of with penalty update converges to the same as that of without penalty update at around 300 iterations. This means that without penalty update, we can have much faster convergence and almost the same performance optimization result. Thus, in our method, we adopt Equation 12 only for probability update.

4) Action selection: Each agent selects an action, namely a DC for its vertex to migrate to, according to the updated probability distribution. To compensate the sacrifice we have made in the probability update step and achieve a good balance between "exploration" and "exploitation", we adopt the Upper

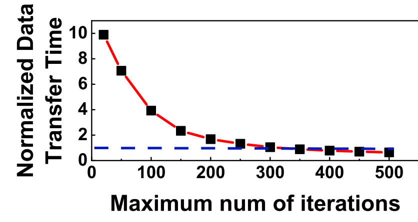


Fig. 6. Red line represents the performance optimization results of using penalty signals for probability update under different numbers of training iterations. Blue dashed line represents the results of without penalty update trained with 10 iterations.

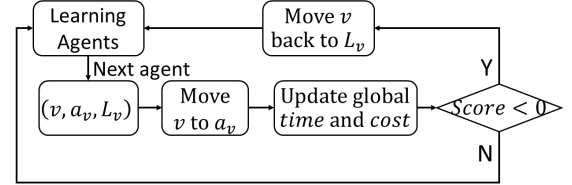


Fig. 7. Detailed flow of the vertex migration step.

Confidence Bound (UCB) strategy [33] to choose the best action. UCB trades off "exploration" and "exploitation" based on how uncertain we are about a selection. Specifically, at each iteration n, an agent update the upper confidence bound value $UCB_n(a)$ of action a according to the below equation:

$$UCB_n(a) = \begin{cases} \inf, & \text{if } N_n(a) = 0 \\ Q_n(a) + c\sqrt{\frac{log(n)}{N_n(a)}}, & \text{if } N_n(a) \neq 0 \end{cases} \tag{13}$$

where $Q_n(a)$ is the mean reward of action a when it was selected prior to iteration n and $N_n(a)$ is the number of times that action a was selected. $\sqrt{\frac{log(n)}{N_n(a)}}$ actually represents how uncertain we are about our selection and c is the confidence value that controls the level of exploration according to our uncertainty. Each agent selects the action with the largest UCB value at each iteration.

5) Vertex migration: In this step, each agent performs vertex migration according to the selected action. As agents compute scores independently, the vertex migration action of one agent changes the state of the entire environment and may affect the optimality of actions taken by other agents. Thus, to achieve good overall performance of graph analytics, we cannot simply perform vertex migration according to the selected action of each agent. Instead, we propose a global optimization method for vertex migration.

To achieve global optimization, learning agents have to cooperate and make vertex migration decisions in a sequential manner. Specifically, each agent is represented using (v, a_v, L_v), namely vertex v, selected action a_v and the current vertex location L_v. Given the set of agents, we iteratively apply the selected action for each agent, and calculate the overall inter-DC data transfer time and cost after applying the action. If the optimization result gets worse, we roll back to the state of before applying the action (i.e., move the vertex back to its original location). We use Equation 10 to determine if the optimization result is getting worse (i.e., $score < 0$, where

score is calculated using the time and cost results before and after applying the action). As the actions of different agents are applied sequentially, changes made to the environment by one agent are visible to all the others. Thus, the vertex migration achieves global optimization. Figure 7 shows the detailed flow.

After the five steps, if we have reached the maximum number of steps or the training is converged, the algorithm is finished. Otherwise, we continue the next training step.

V. Adapting optimization overhead

For large graphs such as Twitter which has over 40 million vertices, the overhead of online RL training becomes very high and hinders it from being applied to large graph partitioning problems. Using multi-agents is our first step toward reducing the overhead of RL. Further, we propose two optimization techniques including batching and straggler mitigation to effectively reduce the overhead of RLCut without sacrificing too much of the cost and performance optimization goals. In order to adapt to graph changes at different frequencies, we propose a sampling based optimization technique which can automatically trade off between partitioning quality and overhead for RLCut.

A. Batching technique

Taking a close look at the training overhead, we find that the bottleneck mainly comes from the sequential vertex migration step. The sequential requirement is set to guarantee global optimization. However, in fact it is not necessary to require strict sequential application of all actions, as the changes made to the environment by a small number of vertex migrations may not be significant enough to impact the decisions of following agents. For example, one agent may decide not to move the vertex in the vertex migration step and thus does not change the environment at all. With this in mind, we propose to batch multiple agents together, where the vertex migration operations execute in parallel for agents in the same batch and execute sequentially for agents in different batches. Although we can design various greedy rules to decide which agents to be batched together, running those greedy rules costs additional time. Instead, we find that randomly select the agents that should be batched together can already generate good optimization effectiveness with low overhead. Batch size is an important parameter to balance the optimization effectiveness and overhead. We study its impact in the evaluations.

B. Straggler mitigation

In our implementation, we use multiple threads to perform the five training steps for different agents in parallel. To achieve good performance of the parallel implementation, one important task is to balance the workload of different threads. A straightforward way of balancing the workload is to equally distribute agents to different threads. However, this does not guarantee load balancing as the computation overhead of different agents differ a lot.

As agents need to cooperate in the vertex migration step, there is actually a barrier before this step where all threads

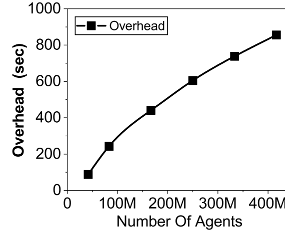


Fig. 8. The training overhead under different number of agents participating in training using Twitter graph and PageRank algorithm.

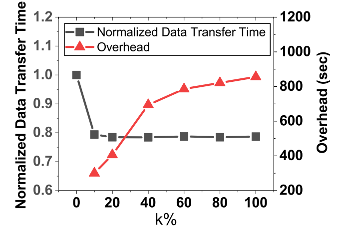


Fig. 9. Normalized data transfer time and training overhead results using agents with the lowest k% degrees participating in the training.

synchronize. Thus, the load balancing task mainly needs to mitigate stragglers in the first four steps. Our evaluations show that, the score function calculation is the most time consuming step of the four. Thus, we propose an agent assignment method based on the score function computation complexity to achieve good performance of our parallel implementation.

According to Equation 10, the score function computation overhead of a vertex v mainly comes from updating T_a^i and C_a^i after moving v from its original location L_v to DC i. Further according to Equation 1 and 7, calculating these two values requires updating the upload and download data sizes of DC L_v and DC i, the complexity of which is related to the degree of vertex v. Thus, we propose to assign agents to threads with the goal of minimizing $Var(\sum_{v \in t} degree_v)$, where t is the thread ID and v is the vertex assigned to thread t. We solve the problem in a greedy fashion. Specifically, we schedule the next unassigned agent to the thread with the least current load. The overhead of this simple greedy method is low and thus is suitable to reduce the overhead of RLCut.

C. Sampling technique

Although the above two techniques can reduce the overhead of RLCut, they do not meet the adaptivity goal which requires the optimization overhead of RLCut to adapt to the graph dynamicity. By studying the overhead of RLCut, we found that the overhead is almost linearly related to the number of agents participating in the training. For example, Figure 8 shows the change of the RLCut overhead when optimizing Twitter graph and PageRank algorithm using different numbers of agents in the RL training. This motivates us to design a sampling-based technique which dynamically tunes the number of agents in the training according to graph dynamicity to achieve adaptivity.

We further found that, the training overhead of different agents also vary, where agents for vertices with higher degrees usually consume more time than agents with lower degrees. Also, different agents contribute differently to the optimization effectiveness of RLCut. We need to sample those agents which can make greater contribution to the optimization goal at a given training time. We denote such agents as important agents. Through detailed study, we found that agents for vertices with lower degrees are more important due to the fact that high-degree vertices usually have more replicas spanning in different DCs. Thus, there are less potential for agents

TABLE II
EXPERIMENTED REAL-WORLD GRAPHS.

Notation	#Vertices	#Edges
LiveJournal (LJ)	4,847,571	68,993,773
Orkut (OT)	3,072,441	117,185,083
uk-2005 (UK)	39,454,746	936,364,282
it-2004 (IT)	41,290,682	1,150,725,436
Twitter (TW)	41,652,230	1,468,365,182

to improve the optimization effectiveness by moving high-degree vertices among DCs. This can also be verified with our experiment as shown in Figure 9. We sample the top $k\%$ agents with the lowest vertex degrees in RLCut and report the resulted data transfer time of geo-distributed graph analytics and the optimization overhead of RLCut. The data transfer time drops sharply when k increases from 0 to 10 and remains almost stable after that. This verifies that agents with high-degree vertices contribute little to the performance optimization goal.

The next question is how to decide the sampling rate. We propose an adaptive way to vary the sampling rate along training steps. Specifically, we start the training with a small sampling rate $SR_0 = 1\%$, and record the training time t_0. Given a required optimization overhead T_{opt}, we calculate SR_i for the i-th training step as follows.

$$SR_i = \frac{T_{opt} - \sum_{k=0}^{i-1} t_k}{Iter_{max} - i} * \frac{1}{i} \sum_{j=0}^{i-1} \frac{SR_j}{t_j} \qquad (14)$$

where $Iter_{max}$ is the maximum number of training iterations. That is, we use the sampling rate and training overhead data of past iterations to estimate the optimal sampling rate in future iterations, assuming that the training overhead of one iteration is proportional to its sampling rate.

With the above three techniques, we are able to adaptively reduce the optimization overhead of RLCut and make it applicable to large dynamic graph partitioning problems.

VI. EVALUATION

We evaluate the effectiveness and efficiency of RLCut using both real Amazon EC2 cloud and a cloud simulator. To emulate the congestion-free network model, we limit the uplink and downlink bandwidths of the instances to be smaller than the WAN bandwidth. The limited bandwidths are proportional to their original bandwidths. We integrate RLCut into Powerlyra. We adopt a multi-threaded implementation to parallelize the multi-agent training of RLCut. Vertex-cut based graph partitioning methods are experimented using PowerGraph [3].

A. Experimental Setup

1) Dataset: We experimented with five large real-world graphs selected from open source datasets [29], [18], which are representative graphs in social networks and web graphs. Table II gives details of the datasets.

2) Graph algorithms: We adopt three graph algorithms which are widely used in different areas, including Pagerank (PR) [34], Single Source Shortest Path (SSSP) [35] and Subgraph Isomorphism (SI) [36]. PR is widely used in web information retrieval to evaluate the relative importance of webpages. SSSP finds the shortest paths starting from a single source to all other vertices in the graph. SI is used to find the subgraphs matching certain graph pattern in a large graph.

3) Comparisons: We compare RLCut with the following state-of-the-art graph partitioning methods, where the first five algorithms are designed for static graph partitioning and Spinner can be used for dynamic graph partitioning.

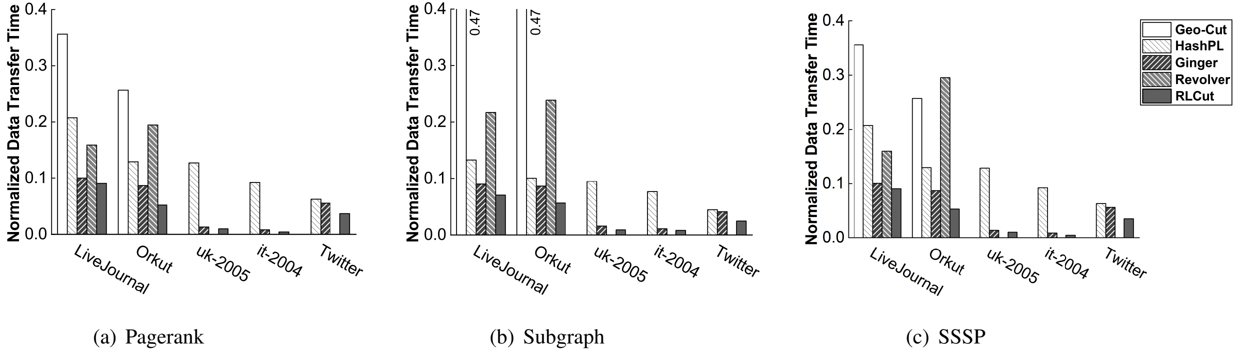
- *RandPG* [3] achieves balanced p-way vertex-cut by randomly assigning all edges to p partitions.
- *Geo-Cut* [1] is a heuristic-based vertex-cut algorithm aiming at optimizing the performance of geo-distributed graph analytics while satisfying the WAN usage budget.
- *HashPL* [6] and *Ginger* [6] are both balanced p-way hybrid-cut algorithms which are using hash-based and greedy-based vertex assignments, respectively.
- *Revolver* [37] is an edge-cut algorithm which also uses LA algorithm to assign vertices to partitions.
- *Spinner* [7] is an edge-cut algorithm based on label propagation algorithm to achieve scalable and adaptive partitioning.

4) Configurations: We use both real cloud and cloud simulator to evaluate the effectiveness of RLCut.

Real cloud experiments (Exp#1). We select eight regions of Amazon EC2 as the geo-distributed DCs, namely US East (USE), US West Oregon (OR), US West North California (NC), EU Ireland (EU), Asia Pacific Singapore (SIN), Asia Pacific Tokyo (TKY), Asia Pacific Sydney (SYD) and South America (SA). In each region, we construct a cluster of five cc2.8xlarge EC2 instances. In all experiments, we compare the performance and monetary cost of graph algorithms optimized by RLCut and the comparisons. To set the budget parameter, we first calculate the lowest cost of moving all graph data into a single DC (i.e., the cost of centralized graph analytics) and set the default budget to 40% of the calculated cost. The required optimization overhead T_{opt} is set to the overhead of Ginger by default. The batch size is set to 48 and maximum number of training steps is 10 by default. All performance results are normalized to those of RandPG and cost results are normalized to the budget, if not otherwise specified.

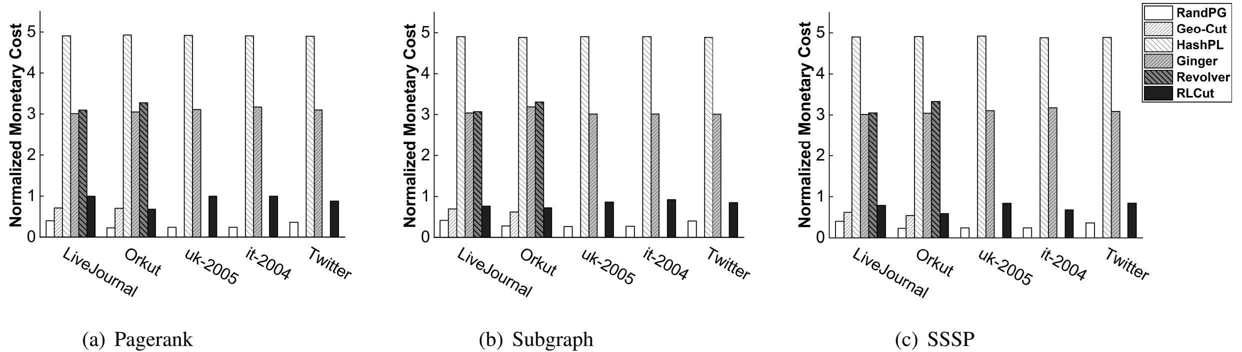
Simulations. To have more detailed and controlled experiments, we perform four sets of simulations to evaluate the sensitivity of RLCut to the budget (Exp#2), batch size parameters (Exp#3), required optimization overhead T_{opt} (Exp#4) and graph dynamicity (Exp#5). For all sensitivity studies, we construct eight geo-distributed DCs to simulate the real Amazon EC2 environment using the measured network bandwidths and prices. All cost results are normalized to the budget if not otherwise specified.

First, we study the impact of the budget constraint to the effectiveness of RLCut on reducing inter-DC data transfer time of geo-distributed graph analytics. Specifically, we vary the budget from 1%, 10%, 40% to 50% of the data movement cost of centralized execution. We compare RLCut with Ginger and Geo-Cut using the Orkut graph and PageRank algorithm. All performance results are normalized to those of Ginger.



(a) Pagerank (b) Subgraph (c) SSSP

Fig. 10. (Exp#1) Normalized data transfer time optimization results obtained on Amazon EC2.



(a) Pagerank (b) Subgraph (c) SSSP

Fig. 11. (Exp#1) Normalized overall monetary cost optimization results obtained on Amazon EC2.

Second, we study the impact of the batch size parameter to the trade-off between optimization effectiveness and overhead of RLCut. Again, we use the large size Twitter graph and increase the batch size from 1, 2, 4, 8, 16, 32 to 48, where 48 is the number of cores on our testbed. To clearly show the impact of batch size, we fix the sampling rate to 10% in all training iterations. We repeat each set of experiments for ten times and report the average results. All performance results are normalized to those when the batch size is 1.

Third, we study the impact of the required optimization overhead T_{opt} to the effectiveness of RLCut using Twitter graph. Specifically, we vary T_{opt} from 1x, 10x, 20x to 50x of the optimization overhead of Ginger. All performance results are normalized to those when T_{opt} is set to 1x.

Finally, we compare RLCut with Spinner to study the adaptivity of RLCut on partitioning dynamic graphs. We construct dynamic graphs by inserting edges to an initial graph. Specifically, we load 70% of LiveJournal edges as the initial graph and insert 1%, 5%, 10%, 15%, 20%, 25%, and 30% of the rest edges to synthesize graphs with different dynamicity. We assume all the new edges are coming in a time window with the same length, and are expected to be inserted quickly into the partitions of the initial graph. We set the time window to 60 seconds. We adopt RLCut and Spinner to perform the initial partitioning and the adaptive partitioning separately, and normalize all performance results to those of Spinner when the number of inserted edges is 1%.

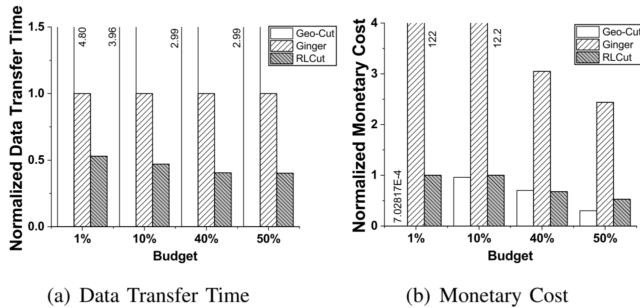
B. Overall Evaluation Results (Exp#1)

Figures 10 and 11 present the normalized inter-DC data transfer time and monetary cost optimization results obtained by the compared algorithms on Amazon EC2. Note that, as the overhead of Geo-Cut and Revolver are much larger than the other comparisons, we only present their results for the relatively small LiveJournal and Orkut graphs. We have the following observations.

First, RLCut obtains the lowest inter-DC data transfer time results among all compared algorithms for all settings. Specifically, RLCut reduces the data transfer time by 90%-100%, 74%-88%, 41%-95%, 10%-48% and 43%-82% over RandPG, Geo-Cut, HashPL, Ginger and Revolver, respectively. RLCut is able to satisfy the budget constraint under all settings while both HashPL and Ginger obtain very high inter-DC data transfer cost (almost as high as the cost of moving all graph data into a single DC). This shows that existing partitioning algorithms designed for a single DC could easily lead to large cost in geo-distributed environment. Although Geo-Cut can satisfy the budget constraint, it obtains much worse performance results compared to RLCut with a much larger optimization overhead. The reason that RandPG obtains low monetary cost is due to its large replication factor. For example, the average vertex replication factor optimized by RandPG for Twitter graph and PageRank is 4.4 while that of HashPL, Ginger and RLCut are 2.8, 2.2 and 2.4, respectively. A larger replication factor can lead to higher runtime inter-

TABLE III
(EXP#1) OPTIMIZATION OVERHEAD (SEC) OF DIFFERENT PARTITIONING METHODS USING PAGERANK ALGORITHM ON AMAZON EC2.

	RandPG	Geo-Cut	HashPL	Ginger	Revolver	RLCut
LJ	6	338	7	15	1760	15
OT	9	525	9	20	1770	20
UK	143	-	141	294	-	294
IT	195	-	204	397	-	395
TW	305	-	312	613	-	618

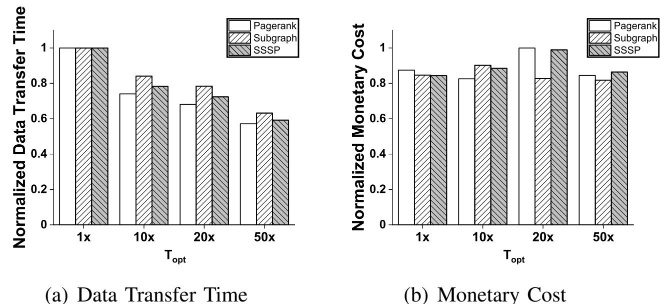


(a) Data Transfer Time (b) Monetary Cost

Fig. 12. (Exp#2) Sensitivity study on the budget constraint using Orkut and PageRank.

TABLE IV
(EXP#3) OPTIMIZATION OVERHEAD (SEC) OF RLCUT WITH DIFFERENT BATCH SIZES FOR TWITTER GRAPH AND PAGERANK ALGORITHM.

Batch size	1	2	4	8	16	32	48
Overhead	735	646	505	421	381	333	300



(a) Data Transfer Time (b) Monetary Cost

Fig. 13. (Exp#4) Sensitivity study on the T_{opt} parameter using Twitter graph.

DC data traffic but less input data movement, which is more important to the overall cost results.

Second, hybrid-cut based partitioning methods obtain much lower inter-DC data transfer time compared to the vertex-cut based partitioning methods. For example, RandPG and HashPL are both hash-based balanced graph partitioning methods. However, the hybrid-cut based HashPL obtains much lower inter-DC data transfer time than the vertex-cut based RandPG under all settings. This is mainly due to the lower WAN usage of the hybrid-cut model compared to vertex-cut. We looked into the runtime inter-DC data transfer size optimized by RandPG and HashPL for different graphs using PageRank algorithm, where HashPL reduces the data transfer size by up to 87% compared to RandPG.

Third, RLCut can well satisfy the user-defined requirement on optimization overhead. Table III shows the optimization overhead of different partitioning algorithms on the five graphs using Pagerank on Amazon EC2. The overhead of RLCut is almost the same as that of Ginger, which is used as the T_{opt} constraint. This demonstrates that our adaptive sampling technique is effective on achieving trade-off between optimization effectiveness and efficiency of RLCut. In Section VI-C, we further perform sensitivity studies on the T_{opt} parameter to show its impact to RLCut.

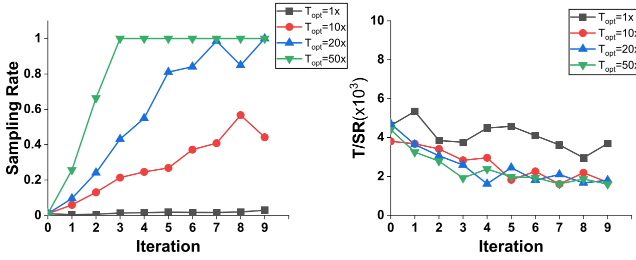
C. Sensitivity Study

1) Budget constraint (Exp#2): As the overhead of Geo-Cut is much larger than the other compared algorithms, we use the Orkut graph for this experiment. Figure 12 shows the normalized inter-DC data transfer time and monetary cost results optimized by three comparisons using Orkut graph and PageRank algorithm. Overall, RLCut obtains the best performance and cost optimization results among all. It reduces the data transfer time by 85%-89% and 47%-60% over Geo-Cut and Ginger, respectively. We have the following observations.

First, when the budget increases, RLCut is able to obtain better performance optimization results and hence higher inter-DC data transfer time reduction over Ginger. This is because when the budget is loose, RLCut has a larger space to search for a good solution. It also demonstrates the ability of RLCut to well explore the solution space. Although Geo-Cut is also able to obtain better performance optimization when the budget is loose, RLCut outperforms Geo-Cut under all settings. The same observation is also found on the monetary cost optimization results. Second, when the budget is larger than 40%, the optimization results of RLCut does not improve much. This is because the required optimization overhead is low, which hinders RLCut from searching a larger space to find a better solution. Finally, RLCut can satisfy the budget constraint under all settings. Even with a tight budget of 1%, it obtains 47% performance improvement over Ginger, the best-performing comparison, with much lower monetary cost.

2) Batch size (Exp#3): With different batch sizes, the variances of inter-DC data transfer time and monetary cost results optimized by RLCut for Twitter graph are lower than 1%, meaning that the batch size parameter does not have much impact on the performance and cost optimization effectiveness of RLCut. On the other hand, a large batch size greatly reduces the optimization overhead of RLCut as shown in Table IV. However, if we continue to increase the batch size to be larger than 48 (the number of cores), the overhead of RLCut will dramatically increase due to the large thread management overhead such as context switch. In our experiments, we set the batch size to 48 by default.

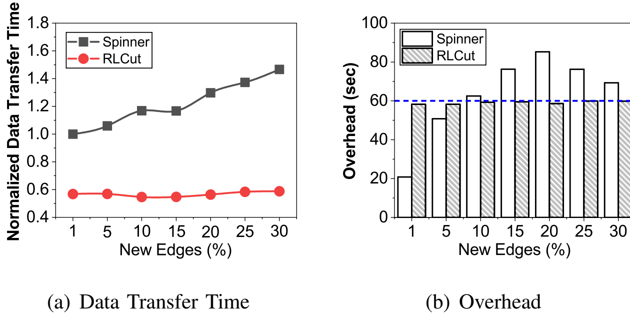
3) Required optimization overhead (Exp#4): Figure 13 shows the normalized inter-DC data transfer time and monetary cost optimization results of RLCut when the required optimization overhead T_{opt} increases from 1x to 50x of the overhead of Ginger. Specifically, the inter-DC data transfer



(a) Sampling rate adaptively chosen per iteration

(b) Proportion between training overhead and sampling rate per iteration

Fig. 14. (Exp#4) Detailed study on sampling under different T_{opt}.



(a) Data Transfer Time

(b) Overhead

Fig. 15. (Exp#5) Sensitivity study on graph dynamicity using LiveJournal and PageRank. Blue line in (b) represents time window size.

time optimized by RLCut decreases by up to 26%, 32% and 43% when T_{opt} increases from 1x to 10x, 20x and 50x, respectively. This is because when the required overhead is high, RLCut can allow more agents exploring the solution space to find a good solution. We look into the sampling rates adaptively chosen at each training iteration for different T_{opt} as shown in Figure 14 (a). Clearly, the average sampling rate per iteration is high when T_{opt} is large, which verifies our analysis. Another observation is that, the sampling rate is increasing along training iterations. We dig deeper into the results and find that the proportion between training overhead and sampling rate is not stable in different iterations. As shown in Figure 14 (b), the proportion is smaller at later iterations. This is because at later training iterations, the optimization is getting close to convergence. As a result, fewer vertices will be migrated and the computation time needed for each agent becomes low. We plan to use such observations to improve our adaptive sampling technique in future work.

4) Graph dynamicity (Exp#5): In this experiment, we assume that 1%-30% of new edges arrive in a 60-second time window which requires the graph partitioning algorithm to complete re-partitioning within 60 seconds (i.e., $T_{opt} = 60s$). Figure 15 shows the normalized data transfer time and overhead optimized by RLCut and Spinner with different graph update frequencies. We have two observations.

First, according to Figure 15(a), RLCut can achieve much better performance optimization results than Spinner. Specifically, RLCut reduces the data transfer time by 43%-60% compared to Spinner. Further, with more new edges coming, RLCut is able to maintain stable optimization quality while Spinner leads to poorer performance optimization results when there are more edges to insert. Second, according to Figure 15(b), RLCut can satisfy the required optimization overhead at all times. In contrary, Spinner wastes the optimization time for further improving optimization effectiveness when the graph update frequency is low and violate the required optimization overhead when the graph update frequency is high. In a real dynamic environment, this could lead to more graph updates queuing in the system and eventually lead to even worse optimization results. Experiments on edge deletions have shown similar observations.

VII. Conclusion

We study the graph partitioning problem in geo-distributed environments, which is an important problem to the performance and cost optimization of graph analytics. However, no existing partitioning can generate good partitioning solutions for our problem, due to the high problem complexity from geo-distributed environments, large graph sizes and the differentiated graph partitioning model. Further, for large-size dynamic graphs which can have different update frequencies, existing dynamic partitioning methods which are based on best-efforts no longer work.

Encouraged by the success of Reinforcement Learning (RL) in many scheduling problems, we propose to use RL to help taming the complexity of the problem. To obtain good performance and cost optimizations with comparable overhead, we propose an adaptive multi-agent learning algorithm named RLCut based on Learning Automata (LA). RLCut incorporates several techniques to adaptively trade-off the optimization effectiveness and user-defined partitioning overhead. Our experiments using real cloud DCs and real-world graphs show that, compared to state-of-the-art partitioning methods using different partitioning models, RLCut improves the performance optimization results by 10%-100% with comparable overhead. When users tolerate longer partitioning overhead, RLCut is able to further improve the performance of geo-distributed graph analytics by up to 43%. When varying the graph changes at different frequencies, RLCut can improve the performance optimization results by up to 60% compared to state-of-the-art dynamic partitioning method [7]. As future work, we plan to improve our adaptive optimization techniques to better explore the solution space.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (No. 62172282, 62072311 and U2001212), Guangdong Basic and Applied Basic Research Foundation 2020B1515120028, Shenzhen Project JCYJ20210324094402008 and JCYJ20210324093212034, and the Tencent "Rhinoceros Birds" - Scientific Research Foundation for Young Teachers of Shenzhen University. Bingsheng's work is in part supported by a research grant (ECT-RP1) in Advanced Research and Technology Innovation Centre (ARTIC) in NUS. Rui Mao is the corresponding author.

REFERENCES

[1] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *ICDCS '17*, 2017, pp. 1397–1407.

[2] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, "GrapH: Heterogeneity-aware graph computation with adaptive partitioning," in *ICDCS '16*, 2016, pp. 118–128.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI'12*, 2012, pp. 17–30.

[4] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *UAI '10*, 2010, p. 340–349.

[5] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *WSDM '14*, 2014, p. 333–342.

[6] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *EuroSys '15*, 2015, pp. 1–15.

[7] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Spinner: Scalable graph partitioning in the cloud," in *ICDE '17*, 2017, pp. 1083–1094.

[8] S. Liu, L. Chen, B. Li, and A. Carnegie, "A hierarchical synchronous parallel model for wide-area graph analytics," in *INFOCOM '18*, 2018, pp. 531–539.

[9] A. C. Zhou, B. Shen, Y. Xiao, S. Ibrahim, and B. He, "Cost-aware partitioning for efficient large graph processing in geo-distributed datacenters," *IEEE TPDS*, vol. 31, no. 7, pp. 1707–1723, 2020.

[10] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," *IEEE TKDE*, vol. 27, no. 6, pp. 1560–1572, 2015.

[11] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *ICDCS '14*, 2014, pp. 144–153.

[12] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *SIGCOMM '19*, 2019, pp. 270–288.

[13] C. Wu, C. Ji, Q. Li, C. Gao, R. Pan, C. Fu, L. Shi, and C. J. Xue, "Maximizing i/o throughput and minimizing performance variation via reinforcement learning based i/o merging for ssds," *IEEE TOC*, vol. 69, no. 1, pp. 72–86, 2019.

[14] J. Zhang, Y. Liu, K. Zhou, G. Li *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *SIGMOD '19*, 2019, pp. 415–432.

[15] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *SC '17*, 2017.

[16] S. Sahoo, B. Sahoo, and A. K. Turuk, "A learning automata-based scheduling for deadline sensitive task in the cloud," *IEEE TSC*, pp. 1–1, 2019.

[17] M. Tan, *Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents*. Morgan Kaufmann Publishers Inc., 1997, p. 487–494.

[18] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[19] A. C. Zhou, Y. Xiao, Y. Gong, B. He, J. Zhai, and R. Mao, "Privacy regulation aware process mapping in geo-distributed cloud data centers," *IEEE TPDS*, vol. 30, no. 8, pp. 1872–1888, 2019.

[20] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10*, 2010, pp. 591–600.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD '10*, 2010, pp. 135–146.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI '14*, 2014, pp. 599–613.

[23] N. Xu, L. Chen, and B. Cui, "LogGP: A log-based dynamic graph partitioning method," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.

[24] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic partitioning for distributed social network graph databases." in *EDBT*, 2015, pp. 25–36.

[25] A. Zheng, A. Labrinidis, and P. K. Chrysanthis, "Planar: Parallel lightweight architecture-aware adaptive graph repartitioning," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 121–132.

[26] J. Huang and D. J. Abadi, "Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, p. 540–551, Mar. 2016.

[27] H. Li, H. Yuan, J. Huang, J. Cui, X. Ma, S. Wang, J. Yoo, and P. S. Yu, "Group reassignment for dynamic edge partitioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2477–2490, 2021.

[28] W. Fan, M. Liu, C. Tian, R. Xu, and J. Zhou, "Incrementalization of graph partitioning algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1261–1274, 2020.

[29] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[30] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *SIGCOMM '15*, 2015, pp. 421–434.

[31] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM '13*, 2013, pp. 3–14.

[32] K. S. Narendra and M. A. Thathachar, *Learning automata: an introduction*. Courier corporation, 2012.

[33] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," in *ICML'10*, 2010, p. 1015–1022.

[34] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW '98*, 1998, pp. 107–117.

[35] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *Journal of Optimization Theory and Applications*, vol. 88, no. 2, pp. 297–320, 1996.

[36] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," in *VLDB '11*, vol. 5, no. 4, 2011, pp. 310–321.

[37] M. H. Mofrad, R. Melhem, and M. Hammoud, "Revolver: vertex-centric graph partitioning using reinforcement learning," in *IEEE CLOUD '18*, 2018, pp. 818–821.