Applying Q-Learning Agents to Distributed Graph Problems

Jeffrey McCrea and Munehiro Fukuda
Division of Computing and Software Systems
University of Washington Bothell
Bothell, WA 98011, U.S.A.
e-mail: {jeffmccr | mfukuda}@uw.edu

***Abstract*—Breadth-first search is used as a brute-force approach to parallelizing graph computations over a distributed graph structure, such as the shortest path, closeness centrality, and betweenness centrality search. As a smart alternative, we integrate Q-learning capabilities into agents, dispatch them over a distributed graph, have them populate the Q-table, and accelerate their graph computations. We developed the Q-learning agents with the Multi-Agent Spatial Simulation (MASS) library and measured their parallel performance when running over a distributed graph with 16K or more vertices. This paper identifies the graph scalability, static/dynamic graph structures, application types, and Q-learning hyperparameters that take advantage of Q-learning agents for parallel graph computing.**

Keywords-Q learning; agent-based modeling; cluster computing; graph algorithms.

I. Introduction

Graphs play a critical role in data representation across diverse domains, such as social networks, transportation systems, and biological networks. The exponential growth of graph sizes and complexity necessitates scalable solutions to handle vast datasets. Distributed frameworks, such as Google's Pregel [1] and Apache Spark GraphX [2], address this need by partitioning graph data across multiple computing nodes for parallel processing. While these solutions provide robust support for large-scale static graphs, their predefined computation models fall short when adapting to dynamic graph environments.

In contrast, reinforcement learning, particularly Q-learning, introduces a paradigm shift by enabling agents to learn graph structures and adapt their strategies dynamically. Q-learning's flexibility demonstrates a promising approach to handling static and dynamic graphs, especially in scenarios with frequent and unpredictable topology changes. To evaluate the Q-learning agents' adaptability, flexibility, and parallel performance, we implemented them within the Multi-Agent Spatial Simulation (MASS) Java library [3].

Our work applied MASS-enabled Q-learning agents to three fundamental graph problems: shortest path, closeness centrality, and betweenness centrality. Experimental results demonstrated the efficiency of this approach, including performance gains of 66% to 99% and a 190% reduction in training time when executed on eight computing nodes. These improvements highlight the potential of combining distributed Q-learning agents to effectively tackle graph analysis.

The remainder of this paper is structured as follows. Section II reviews related work on parallel graph computing, reinforcement learning, and dynamic graph processing. Section III details the design and implementation of Q-learning agents within the MASS framework. Section IV presents the experimental setup and results, comparing Q-learning-based solutions with existing MASS implementations. Finally, Section V concludes our work on distributed graph computing using Q-learning agents.

II. Related Work

This section reviews some libraries and algorithms related to our MASS-parallelized Q-learning agents from the following three viewpoints: (1) parallel graph computing, (2) Q-learning graph computing, and (3) tolerance to dynamic graphs.

A. Parallel graph computing

The exponential growth of graph sizes has necessitated distributed platforms, such as Google's Pregel [1] and Spark GraphX [2], to handle large-scale graph computations. In Pregel, the master computing node partitions a graph dataset across worker computing nodes for parallel computation. Pregel simulates message propagation or breadth-first search over the graph in iterative supersteps, where each vertex exchanges messages with its neighbors. GraphX utilizes a similar vertex-centric message propagation approach, facilitated by the Pregel API and enhanced through Spark's Resilient Distributed Datasets, enabling efficient in-memory computation and fault tolerance.

While GraphX and Pregel are robust for static graphs, they struggle with dynamic environments where changes in the graph necessitate complete recomputation. Depending on the size of the graph, this can result in significant computational costs as the frameworks have no native mechanism to respond incrementally to these changes. In this way, adaptive solutions have a significant advantage over purely static implementations.

B. Q-Learning Graph Computing

Q-learning [4] has seen wide adoption in reinforcement learning due to its ability to learn and adapt to unknown environments and derive optimal policies for navigating them, particularly in the shortest path problem. In this context, graph vertices represent states, edges represent possible actions, and rewards are based on the edge weights. As the Q-learning agent navigates the graph, it iteratively populates the Q-table, capturing the optimal policies for efficient traversal.

In literature, Q-learning has successfully addressed the shortest path problem. Sun [5] utilized Q-learning to navigate a static city graph, demonstrating its effectiveness in road network-like graph structures. Wang et al. [6], applied Q-learning to autonomous robots in a grid-like environment, prioritizing the shortest path and obstacle avoidance. The study highlighted Q-learning's ability to enable intelligent decision-making in uncertain environments driven by learned agent intelligence. Nannapaneni [7] introduced a novel path-routing algorithm based on a modified Q-learning algorithm for dynamic network packet routing, underscoring its adaptability to evolving environments. However, these studies often focus on small graphs with fixed hyperparameters on single-machine setups.

Two extensions of the shortest path search are closeness and betweenness centrality computation [8]. The former computes the shortest path from each vertex *u* to all other vertices *v*, sums all the path lengths, and takes its reciprocal with $C(u) = \frac{N-1}{sum(u)}$. The latter determines all shortest paths between each pair of vertices *u* and *v* in a graph, counts how many paths pass through a given vertex, and computes the centrality with $B(s) = \sum_{u \neq s \neq v} \frac{\sigma_{uv}(s)}{\sigma_{uv}}$, where σ_{uv} is the total number of the shortest paths from vertex *u* to vertex *v*, and $\sigma_{uv}(s)$ is the number of those paths passing through s. For dynamic graphs, recalculating these metrics with each change can be computationally expensive. Parallel approaches, such as using landmarking techniques [9] or GPU acceleration [10], have been proposed to address this complexity. However, these methods often require extensive preprocessing or specialized hardware, limiting their practicality.

C. Dynamic graphs

Dynamic graphs, where changes to the graph's vertices or edges occur over time, better emulate real-world situations like those in traffic networks. Changes to the graph can be incremental, adding vertices and edges or increasing weights, or decremental, removing vertices and edges or decreasing weights [11]. Traditional algorithms, such as Dijkstra's or Bellman-Ford, are designed for static graphs as they require complete recomputation for each change. This recomputation becomes prohibitively expensive when handling frequent changes.

To address the challenges associated with dynamic graphs, several approaches have been explored. Early efforts adapted traditional algorithms like A* [12] for incremental changes. Techniques like those employed in Dynamic Dijkstra's [13] and D* [14] utilized incremental search strategies that recompute only affected portions of the graph. Subsequent methods used landmark-based techniques [15] for real-time shortest path approximation and distributed techniques [16] to partition the graph data and subgraph computation across a distributed cluster of computing nodes. While these approaches improve performance, they often rely on knowledge of where changes occur, which is impractical for unpredictable, real-world environments. Additionally, landmarking requires extensive preprocessing to select landmark vertices accurately. Our work focuses on measuring dynamic performance without prior knowledge of graph changes or the underlying structure of the graph.

D. Challenges and Agent-based Solutions

Our project seeks to expand on these studies by exploring a diverse range of graph sizes, assessing the performance of Q-learning on dynamic graphs using the MASS framework, and improving the algorithm's efficiency through performance-based hyperparameter tuning and MASS-enabled agent features. For this purpose, we take the following four strategies:

1) MASS constructs and maintains a distributed graph over a cluster system for repetitive graph computing [17].
2) MASS agents build a Q-table over a distributed graph and dynamically tune hyperparameters during training.
3) MASS facilitates dynamic graphs by adding or deleting vertices and their edges.
4) MASS agents adapt to graph changes by reaching semi-optimal solutions when the initial graph structure is altered.

III. An Implementation of Q-Learning Agents

This section details the Q-learning algorithm for the shortest path problem and its application to closeness and betweenness centrality. It also covers the implementation details for the Q-learning shortest path, closeness centrality, and betweenness centrality in MASS, including MASS-specific performance improvements.

A. MASS Library

The MASS Java Library [3] uses two primary components: *Places* and *Agents*. *Places* represent individual data members of a larger dataset, whereas *Agents* comprise individual agent execution entities that traverse *Place* objects to perform computation. Individual *Place* objects are allocated across a specific number of computing nodes within a cluster and can be referenced using a universally recognized set of coordinates.

After mapping each *Place* within the cluster, they are assigned to threads, allowing them to communicate with other *Place* objects and the agents residing on them. *Agents* are organized into groups on each computing node and can move between nodes using serialization and TCP communication.

The MASS library has been extended to support explicit graph structures in a multi-node cluster. *GraphPlaces*, an extension of the original *Places* class, consists of *Vertex-Place* objects that store graph vertex information. Graphs can be manually created by invoking the `addVertex()` and `addEdge()` methods, or users can load graph data from a supported graph format, including Hippe, Matsim, or a few propriety formats [17]. MASS enables parallel file processing, making graph creation far more efficient than serial methods. Keeping with its tradition of distributed cluster computing, vertices are balanced across the entire cluster in a round-robin fashion to ensure an even distribution of graph data between all the cluster nodes.

B. Algorithm Design

The Q-learning shortest path algorithm models the environment as a graph $G = (V, E)$, where vertices V are states and edges E are actions connecting them. An agent learns the optimal path by iteratively updating the Q-table $Q(s, a)$, which stores the value of taking action a in state s. The Q-value update rule is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_a Q(s',a) - Q(s,a) \right]$$

Here, α is the learning rate, γ is the discount factor, and r is the reward received for the state transition. Rewards guide the agent by assigning penalties for non-terminal states as the edge cost, a positive reward for reaching the destination, and a penalty for encountering dead ends. The parameter ϵ controls the exploration-exploitation tradeoff in an epsilon-greedy action selection process (see Figures 1 and 2), where agents explore new paths with probability ϵ and exploit the highest Q-value action otherwise. Training continues until the Q-values converge or the maximum number of episodes is met.

To improve on traditional Q-learning, dynamic hyperparameter tuning is utilized to enhance learning efficiency and adaptiveness. This approach uses a distributed reward window, a linked list that stores cumulative rewards received during training episodes. If the current average reward in the window decreases relative to the previous average, α and ϵ are increased to promote exploration. In contrast, if the reward improves, these parameters are decreased to stabilize learning and encourage exploitation. This dynamic adjustment enables agents to adapt to changing performance, improving convergence and efficiency compared to fixed parameter decay.

Beyond finding the shortest path, the Q-table also enables suboptimal path enumeration. By defining a Q-value and path length threshold, agents can explore alternative paths within a certain percentage of the optimal Q-value and path length. This flexibility allows the agent to identify viable alternative routes in the event of changes in the graph structure, such as blocked paths or updated edge weights. The Q-table's comprehensive knowledge provides a robust basis for exploring these alternative paths without requiring additional training.

The Q-learning shortest path algorithm naturally extends to calculating graph centrality metrics like closeness and betweenness. In MASS, this is achieved by assigning a Q-learning agent to each source-destination pair, with Q-tables organized in a two-dimensional array of hashmaps at each vertex. For betweenness centrality, agents are further tasked with enumerating all optimal paths between vertex pairs.

C. Algorithm Implementation with MASS

The MASS library streamlines the application of the Q-learning shortest path algorithm by distributing computation and facilitating multi-agent collaboration. The algorithm is built upon the fundamental abstractions of MASS: *Places* and *Agents*, utilizing the graph-specific extension of *Places*, *GraphPlaces*, to enable easy-to-use MASS graph functionality. Furthermore, the *Node class*, representing individual graph nodes and an extension of *VertexPlace*, is created to incorporate Q-learning functionality related to distributed Q-table storage and dynamic hyperparameter tuning.

```
1: node ← getPlace()
2: neighbors ← unvisited neighbors of node
3: if neighbors is empty then
4:     return −1 {Terminal state}
5: end if
6: if random() < ϵ then
7:     {Exploration: Choose a random action}
8:     return random neighbor from neighbors
9: else
10:    {Exploitation: Best action based on Q-values}
11:    return neighbor with arg max Q(node, neighbor)
12: end if
```

Figure 1. Action Selection algorithm in the Q-learning process.

The primary agent, *QLAgent*, navigates the graph, updates Q-values, and dynamically adjusts hyperparameters based on feedback from the reward window. During training, the *QLAgent* iteratively explores the graph, storing the learned values in the distributed Q-tables. The action selection process is handled by the `chooseAction` function, detailed in Figure 1, which implements the epsilon-greedy policy to balance exploring new actions and exploiting the highest-value actions stored in the Q-table. At the end of each episode, the *QLAgent* evaluates its performance and fine-tunes its hyperparameters using the `adjustAlphaEpsilon` function, described in Figure 2. This adaptive function increases exploration and learning rates when performance declines and decreases when performance improves, ensuring efficient learning throughout the training process. The adjustment factors of 1.035 and 0.95 for α and ϵ were identified through manual testing as values that provide balanced yet gradual change without excessive oscillation. Once training is complete, the *QLAgent* uses the learned values from the Q-table to identify the optimal path.

```
1: Output: Adjusted α and ϵ
2: if rewardSize == slidingWindowSize then
3:     currentAvg ← rewardSum / rewardSize
4:     if currentEpisode ≥ slidingWindowSize then
5:         previousSum ← rewardSum - rewardWindow
6:         previousAvg ← previousSum / (rewardSize - 1)
7:         if currentAvg > previousAvg then
8:             ϵ ← min(1.0, ϵ · 1.035)
9:             α ← min(0.5, α · 1.035)
10:        else
11:            ϵ ← max(0.01, ϵ · 0.95)
12:            α ← max(0.01, α · 0.95)
13:        end if
14:    end if
15: end if
```

Figure 2. Dynamic α and ϵ tuning algorithm.

Suboptimal paths are explored using the *PathAgent*, a simplified version of the *QLAgent* designed for path enumeration. The *PathAgent* identifies routes within user-defined Q-value

and path length thresholds, enabling the discovery of both optimal and viable alternative paths.

MASS significantly enhances Q-learning performance on large-scale graphs by optimizing resource utilization and scalability. The distributed architecture efficiently shares memory and computational load across computing nodes and enables the sparse Q-table representation focused on only relevant state-action pairs. Moreover, MASS's multi-agent training support allows numerous agents to explore the graph simultaneously, accelerating the training process and ensuring comprehensive graph coverage.

IV. Evaluation

This section presents the experimental results for the Q-learning shortest path, closeness, and betweenness centrality applications, comparing their performance against current MASS implementations. Experiments were conducted on the CSSMPI cluster at the University of Washington Bothell, comprising eight virtual machines allocated four cores at 2.10GHz from an Intel Xeon Gold 6130 processor and 16GB of RAM. The MASS Q-learning applications utilized MASS Java Core version 1.4.3, enhanced by Fastutil version 8.5.8 for optimized data structures. Performance metrics focus on training time, adaptability for dynamic graph changes, and efficiency improvements through multi-agent training.

TABLE I. Graph Datasets Properties

Dataset	Vertices	Edges	Avg Deg	Max Deg
Synthetic (Shortest)	500-16K	63K-64M	126-4038	189-5833
Road Networks	1861-19K	4K-50K	2.35-5.3	5-10
Synthetic (Centrality)	8-256	16-16K	2-64	3-97

Table I summarizes the datasets used for evaluation, including synthetic graphs for shortest path testing, real-world road network graphs for dynamic performance analysis, and synthetic centrality graphs for closeness and betweenness centrality testing.

A. Shortest Path

Figure 3 demonstrates the training performance of the Q-learning shortest-path application on synthetic graphs. The application performs best on single-node executions for graphs up to 8000 vertices, as the need for communication between nodes in the cluster is minimized. For the 16,000-vertex graph, multi-node execution becomes necessary, with eight computing nodes achieving optimal performance due to distributed agent coordination and parallel execution.

After training, the Q-learning shortest path algorithm significantly outperforms the current MASS shortest path implementation in output speed, as shown in Figure 4. The trained *QLAgent* efficiently navigates the graph using the trained Q-table to produce the shortest path, demonstrating the inherent advantage of pre-trained Q-learning models.

Dynamic benchmarking evaluates the adaptability of the Q-learning application to graph changes. Figure 5 compares Q-learning performance with the current MASS shortest path

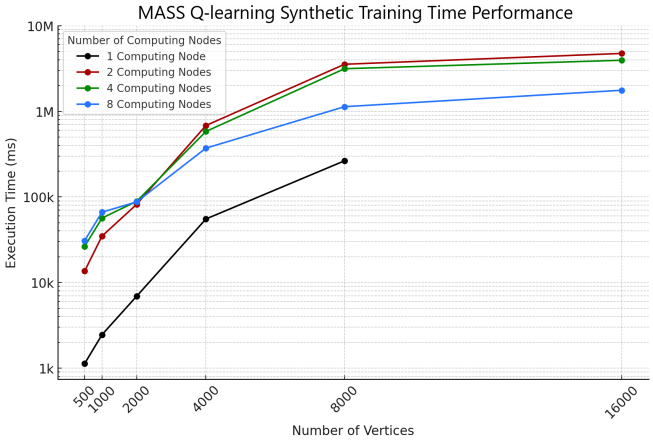

Figure 3. MASS Q-learning Shortest Path Synthetic Training Time

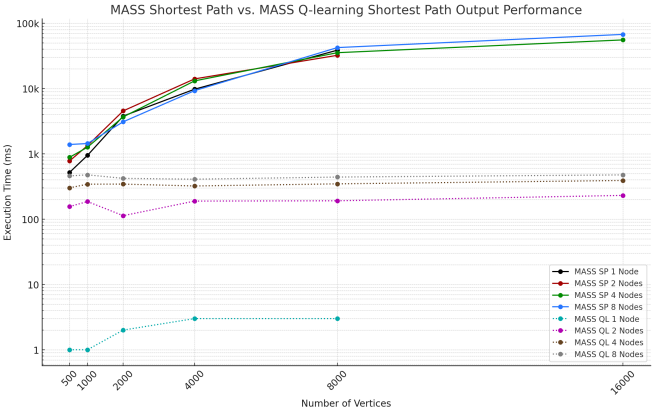

Figure 4. MASS Shortest Path vs. MASS Q-learning Shortest Path

application when faced with the removal of a single vertex. The Q-learning implementation excels in well-connected synthetic graphs, quickly adapting to minor changes due to the knowledge stored in the Q-table. More substantial disruptions, such as the removal of five vertices, resulted in increased retraining times, thereby making rerunning the current MASS implementation more efficient for significant changes.

B. Closeness and Betweenness Centrality

The Q-learning approach for centrality analysis demonstrates potential but faces scalability challenges. For closeness centrality, training individual Q-tables for all vertex pairs resulted in quadratic scaling with the number of vertices, making the approach inefficient for larger graphs as shown in Figure 6. Similarly, betweenness centrality incurred additional overhead from enumerating all shortest paths between vertex pairs, further amplifying scalability issues. The scalability issues and the need to retrain extensively for dynamic graph changes made the Q-learning approach less competitive compared to the current MASS implementation.

Overall, while the Q-learning-based centrality method showed promise for small, static graphs, its scalability and

 ISBN: 978-1-68558-241-8

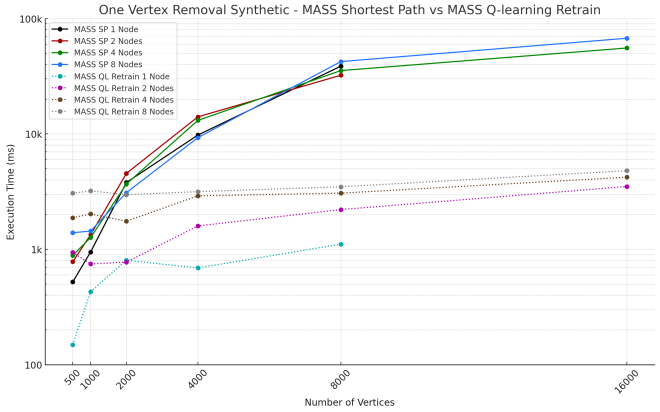

Figure 5. MASS vs. MASS Q-learning Shortest Path One Vertex Removal

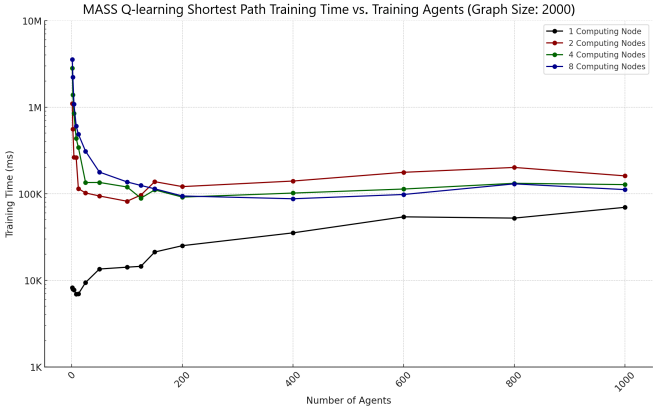

Figure 7. MASS Q-learning Shortest Path Multi-Agent Training Time

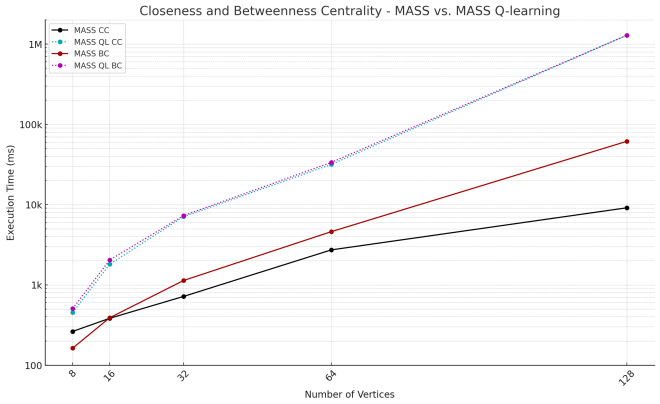

Figure 6. MASS vs. MASS Q-learning Closeness and Betweenness Centrality

adaptability fell short of the success in the shortest path application. Future work could explore hybrid approaches or optimizations, such as combining Q-learning with heuristics or leveraging pre-computed graph partitions, to address these limitations and improve centrality analysis performance.

C. MASS Enabled Q-learning Performance Improvements

The Q-learning shortest path application incorporates key innovations enabled by the MASS framework: multi-agent training, distributed reward window, and dynamic hyperparameter tuning. Multi-agent training, facilitated by MASS's distributed multi-agent architecture, significantly reduces training times by allowing agents to operate in parallel across all computing nodes. As shown in Figure 7, this approach achieved a 190% reduction in training time on an 8-node cluster with 1750 training agents compared to a single agent execution. The MASS framework facilitates this improvement by efficiently managing agent migration, synchronization, and data sharing across distributed nodes, enabling seamless multi-agent parallelism.

In multi-agent Q-learning scenarios, a considerable challenge is the need for agents to share the knowledge gained during training, often requiring costly synchronization methods. Our implementation of a distributed reward window, enabled by the MASS framework, was a key enhancement to the Q-learning applications. Figure 8 illustrates that the distributed reward window allows agents to share their cumulative episodic rewards without costly synchronization methods, enabling efficient hyperparameter tuning and reducing training times. MASS's distributed architecture provides a simple and effective mechanism for aggregating these rewards. By streamlining communication and eliminating costly synchronization overhead, the distributed reward window significantly increases the scalability and efficiency of Q-learning as applied to large-scale graph analysis.

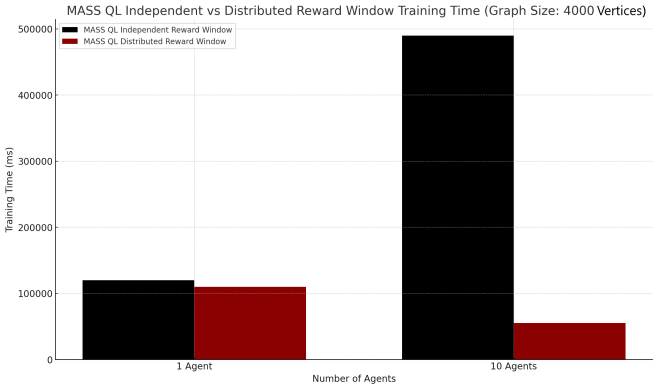

Figure 8. MASS Q-learning Independent vs. Distributed Reward Window

Dynamic hyperparameter tuning further enhances the training process by adjusting exploration and learning rates based on the aggregated rewards from the distributed reward window. Figure 9 demonstrates how this capability minimizes unnecessary computations, reducing overall training time without compromising accuracy. By leveraging MASS's robust distributed communication and state management features, agents can adapt their hyperparameters dynamically, ensuring a balanced and efficient approach between exploration and exploitation.

These improvements underscore the critical role of the MASS framework in enhancing the Q-learning implementa-

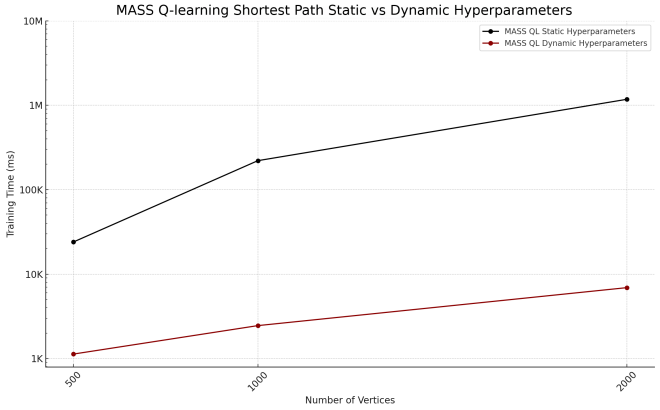

Figure 9. MASS Q-learning Static vs. Dynamic Hyperparameter Tuning

tions. By providing a robust distributed parallel environment, MASS enables efficient multi-agent operations, adaptive learning, and scalable performance for large-scale graph analysis.

V. Conclusion and Future Work

We successfully demonstrated the capability of MASS to accelerate reinforcement learning algorithms like Q-learning through its distributed and parallel multi-agent programming paradigm. In our experimentation, MASS's multi-agent faculties resulted in up to a 190% reduction in training time compared to a single agent, and the distributed architecture enabled Q-learning for large-scale graph analysis. MASS's ability to support distributed graph computing and facilitate adaptive learning presents a promising framework for future multi-agent machine learning applications. Furthermore, the Q-learning applications, particularly the shortest path implementation, exhibited promising performance, especially in their adaptability to dynamic data, and served to highlight where MASS can be improved to better align with machine learning workloads.

Our future work will focus on expanding the scope of machine learning in MASS by exploring advanced graph-based techniques, such as Graph Convolutional Networks and embedding techniques like FastRP. Additionally, research into efficient agent communication, reduced agent overhead, and more dynamic graph modifications could unlock new capabilities and improve MASS's adaptability for a broader range of applications.

Finally, the MASS library and the applications featured in this project are available at: http://depts.washington.edu/dslab/MASS/.

References

[1] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Association for Computing Machinery, 2010, pp. 135–146.

[2] R. S. Xin *et al.*, *Graphx: Unifying data-parallel and graph-parallel analytics*, 2014. arXiv: 1402.2394 [cs.DB].

[3] M. Fukuda, C. Gordon, U. Mert, and M. Sell, "Agent-based computational framework for distributed analysis," *IEEE Computer*, vol. 53, no. 4, pp. 16–25, 2020.

[4] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[5] Z. Sun, "Applying reinforcement learning for shortest path problem," in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, IEEE Computer Society, 2022, pp. 514–518. DOI: 10.1109/BDICN55575.2022.00100.

[6] X. Wang, L. Jin, and H. Wei, "The shortest path planning based on reinforcement learning," *Journal of Physics: Conference Series*, vol. 1584, no. 1, 012006: 1–6, 2020. DOI: 10.1088/1742-6596/1584/1/012006.

[7] R. Nannapaneni, "Optimal path routing using reinforcement learning," Dell Inc., Tech. Rep., 2020, Accessed: May 03, 2024.

[8] J. Golbeck, "Chapter 3 - network structure and measures," in *Analyzing the Social Web*, Morgan Kaufmann, 2013, pp. 25–44. DOI: 10.1016/B978-0-12-405531-5.00003-1.

[9] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck, "Computing classic closeness centrality, at scale," in *Proceedings of the Second ACM Conference on Online Social Networks*, Association for Computing Machinery, 2014, pp. 37–50. DOI: 10.1145/2660460.2660465.

[10] K. Shukla, S. C. Regunta, S. H. Tondomker, and K. Kothapalli, "Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs," in *Proceedings of the 34th ACM International Conference on Supercomputing*, Association for Computing Machinery, 2020, pp. 10–21. DOI: 10.1145/3392717.3392743.

[11] D. Ferone, P. Festa, A. Napoletano, and T. Pastore, "Shortest paths on dynamic graphs: A survey," *Pesquisa Operacional*, vol. 37, no. 3, pp. 487–508, 2017. DOI: 10.1590/0101-7438.2017.037.03.0487.

[12] J. Doran and D. Michie, "Experiments with the graph traverser program," *Proceedings of the Royal Society A*, vol. 294, no. 1437, pp. 235–259, 1966. DOI: 10.1098/rspa.1966.0205.

[13] Y. Yang and E. P. Chan, "Shortest path tree computation in dynamic graphs," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 541–557, 2009. DOI: 10.1109/TC.2008.198.

[14] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, vol. 4, 1994, pp. 3310–3317. DOI: 10.1109/ROBOT.1994.351061.

[15] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *23rd International World Wide Web Conference*, Association for Computing Machinery, 2014, pp. 237–248. DOI: 10.1145/2566486.2568007.

[16] H. Cui, R. Liu, S. Xu, and C. Zhou, "Dmga: A distributed shortest path algorithm for multistage graph," *Scientific Programming*, vol. 2021, no. 1, p. 6 639 008, 2021. DOI: 10.1155/2021/6639008.

[17] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, "Agent-navigable dynamic graph construction and visualization over distributed memory," in *7th Int'l Workshop on BigGraphs'20*, IEEE, 2020, pp. 2957–2966.