Enhancement of Agent Performance with Q-Learning

Jeffrey McCrea

A whitepaper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2024

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

Enhancement of Agent Performance with Q-Learning

Jeffrey McCrea

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
Computer Science and Software Engineering

Graphs store data from various domains, including social networks, biological networks, transportation systems, and computer networks. As these graphs grow in size and complexity, single-machine solutions become impractical due to limitations in computational resources. Distributed graph computing addresses these challenges by leveraging multiple machines to process and analyze large-scale graphs collaboratively.

This capstone project investigates the enhancement of distributed graph computing performance in the Multi-Agent Spatial Simulation (MASS) library by integrating Q-learning for computing shortest path, closeness centrality, and betweenness centrality on distributed large-scale dynamic graphs. This approach is compared to traditional and agent-based graph computing algorithms. Previous approaches in the MASS framework relied on large populations of unintelligent agents to exhaustively traverse graphs to compute solutions, making them inefficient when faced with dynamic graph data. By leveraging Q-learning and the distributed agent-based graph capabilities of MASS, we aim to optimize the decision-making processes of distributed agents, thus improving computational efficiency and accuracy.

Experimental results demonstrate that the adaptive learning mechanism of Q-learning, coupled with the MASS library, allows agents to dynamically adjust to changing graph structures, leading to a more robust and scalable distributed graph computing solution. This research contributes to the field of distributed systems and artificial intelligence by providing an innovative approach to enhancing multi-agent intelligence for graph computing tasks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my professor and chair of my committee, Dr. Munehiro Fukuda, for his patience and invaluable assistance on my project. Without his guidance and support, I would not have accomplished a fraction of what I did.

I also want to extend my heartfelt thanks to my colleagues at the Distributed System Laboratory for their assistance throughout this journey. In particular, I would like to thank Lilan Cao for the significant time she spent helping debug and troubleshoot issues. Her support and collaboration were instrumental in the completion of this project.

Chapter 1. INTRODUCTION

Graphs store data from various fields, including social networks, biological networks, transportation systems, and computer networks. As these graphs grow in size and complexity, single-machine solutions become impractical due to limitations in computational resources. Distributed graph computing addresses these challenges by leveraging multiple machines to process and analyze large-scale graphs collaboratively. Apache Spark GraphX [1] and Google's Pregel [2] are two prominent solutions for distributed graph computing.

GraphX integrates graph computation within a data-parallel framework, utilizing Resilient Distributed Datasets (RDDs) for representing and processing graph data. Vertices are partitioned across a cluster of computing nodes, and the Pregel API enables iterative algorithms for parallel computing. Pregel follows a vertex-centric programming model, where computation proceeds in parallel supersteps, and individual vertices perform local computations and message passing to their neighbors. GraphX and Pregel offer large-scale graph data computation; however, while these are robust solutions, intelligent agent-based approaches can offer superior flexibility and adaptability for dynamic and complex graph structures, as they inherently can better handle real-time changes and interactions within the graph.

Recently, the Distributed Systems Laboratory (DSL) at the University of Washington Bothell has been developing agent-based approaches to large-scale distributed data analysis built on the Multi-Agent Spatial Simulations (MASS) library [3]. Agent-based solutions have been successfully applied to tasks such as biological network motif search [4], transportation simulations [5], climate change analysis [6], and many traditional algorithms like breadth-first search (BFS). In contrast to the previously mentioned SparkX and Pregul, agent-based data analysis in MASS focuses on distributed graphs or arrays over a cluster system and dispatching a large number of reactive agents to ramble over the data and compute efficient and accurate solutions.

While the approach with agent-based solutions in MASS has been demonstrated to be effective in reducing execution time compared to traditional algorithms, agent-based solutions like BFS for distributed graph computing require exhaustive graph traversals by large unintelligent agent populations, making them inefficient when faced with dynamic graphs. Machine learning solutions, like Q-learning, a popular reinforcement learning algorithm, allow agents to learn the structure of a graph through interactions with the environment and dynamically respond to changes in the graph structure. This adaptive learning capability enables agents to dynamically adjust their behaviors based on real-time feedback, leading to more efficient and resilient graph processing. Unlike the more rigid, predefined computation models in GraphX and Pregel, Q-learning agents can continuously improve their strategies, making the system more robust to changes in graph structure or workload. This approach can be particularly advantageous in scenarios with highly dynamic or unpredictable graph topologies, where traditional methods might struggle to maintain performance and efficiency. By leveraging Q-learning, we can enhance the performance of agent-based solutions and offer a more intelligent and flexible alternative to conventional and agent-based approaches for distributed graph computing.

The enhancement of agent performance with Q-learning project has had four main goals:

1. **Design Agent-based Q-learning Algorithm Applications:** The primary goal at the inception of this project was to identify and implement intelligent agent-based solutions in the form of Q-learning for graph computing problems previously implemented in the MASS library.

2. **Assess Agent Performance:** To complement the primary goal, we sought to assess the agent performance against the currently implemented MASS Java shortest path, closeness centrality, and betweenness centrality applications against static and dynamic graph data.
3. **Evaluate MASS Agent Machine Learning Capabilities:** Examine the suitability of MASS's features for enabling performance improvements in Machine Learning algorithms like Q-learning.
4. **Update and Evaluate Previous MASS Application:** The previous MASS Applications were built on MASS Java 1.0. A secondary goal for this project was to update the previous applications to the latest MASS Java version and evaluate the performance impact.

The end result of this project saw the goals met. The final version of the project designed and implemented three Q-learning-based algorithms to address the shortest path, closeness centrality, and betweenness centrality. The performance of the Q-learning implementations was assessed against the current MASS implementations, comparing both the training and output performance on static and dynamic graphs. The results demonstrated that a trained Q-learning agent presented significant performance improvements on applications like the shortest path, with between 66% and 99% performance increase on eight-node executions. Additionally, unique features in MASS were utilized to enhance the performance of the Q-learning implementations, with features like multi-agent training resulting in training time decreases of 190% on large-scale graphs. The results of this project provide a unique perspective and improvement on past works and serve to motivate future progress on machine learning in the MASS Java framework.

Chapter 2. BACKGROUND

This chapter provides a comprehensive overview of the graph problems addressed in this project and the conventional methods used to solve them. It also reviews the MASS Java library and agent-based approaches and explores the application of reinforcement learning to graph computing.

2.1 Graph Problems

In graph theory, the shortest path problem, closeness centrality, and betweenness centrality constitute essential graph problems with significant practical applications. These problems are crucial to domains such as network routing, social network analysis, and infrastructure optimization. Understanding and optimizing solutions to these problems can lead to more efficient and robust network designs, better resource allocations, and enhanced communication strategies.

The shortest path problem involves finding the most efficient route between vertices in a graph. In this context, vertices represent points or locations, and edges present connections weighted by distance, time, or cost. Formally, given a graph $G = (V,E)$ with vertices V and edges E, the goal is to find a path from a source vertex to a destination vertex such that the sum of the edges is minimized. Traditional algorithms for solving the shortest path problem include Dijkstra's algorithm for non-negative weights, the Bellman-Ford algorithm for negative weights, and the Floyd-Warshall algorithm for all-pairs shortest path problems.

Closeness centrality measures a vertex's importance based on its proximity to all other vertices in the network. It quantifies how close a vertex is to all other vertices in the network and helps identify vertices that can spread information efficiently or quickly connect with different parts of the network. The measure is particularly useful in social network analysis [7], communication networks [8], and other scenarios where understanding transfer efficiency across the network is important. Traditional methods for calculating closeness centrality include BFS for unweighted graphs, Dijkstra’s for weighted graphs with non-negative weights, and Floyd-Warshall for all-pairs shortest path.

Betweenness centrality quantifies a vertex’s importance based on its position as an intermediary in the paths connecting other vertex pairs, highlighting central connectors or ‘bridges’ within the network [9]. The measure is crucial to applications in social networks [9], protein interaction networks[10], and optimization of computer network infrastructure [11]. Traditional approaches to calculating betweenness centrality include Brandes’ algorithm, which computes exact centrality scores using BFS for unweighted graphs, and Dijkstra's algorithms for weighted graphs.

Exploring these fundamental graph problems and their traditional solutions lays the groundwork for applying advanced techniques to improve performance and robustness in different scenarios. By implementing adaptive methods like Q-learning, we can address the complexity and dynamism of real-world networks, potentially enhancing efficiency and adaptability in solving these complex problems.

2.2 Conventional Approach

As graphs grow in size, distributed platforms like Spark’s GraphX and Google’s Pregel become necessary for handling large-scale graph computing. In Pregel, the shortest path problem is solved using a vertex-centric approach, as shown in Figure 2.1. The master computing node partitions the graph data across worker computing nodes for parallel computation. Beginning at the source vertex, Pregel operates in iterative supersteps, where each vertex exchanges messages with its neighbors containing potentially updated minimum distances from the source vertex [2]. GraphX utilizes a similar vertex-centric message propagation approach, facilitated by the Pregel API and enhanced through Spark's Resilient Distributed Datasets (RDDs), enabling efficient in-memory computation and fault tolerance [12].

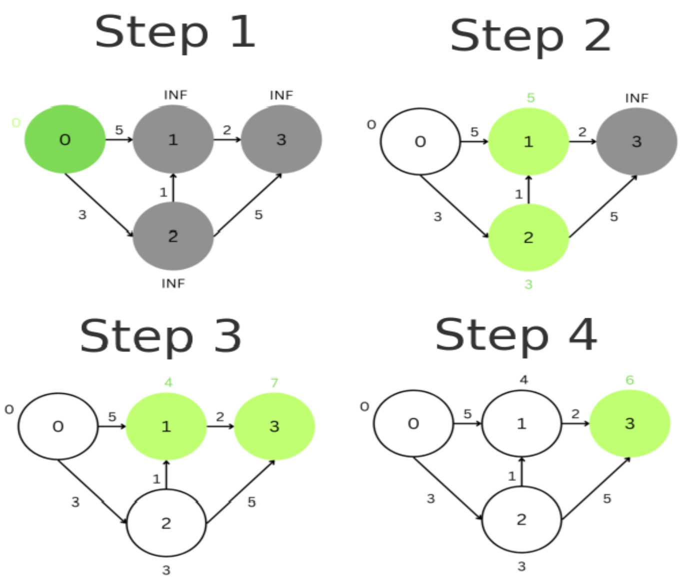


Figure 2.1 Pregel Shortest Path Process

While GraphX and Pregel are robust for static graphs, they struggle with dynamic environments where graph changes necessitate complete recomputation to respond to any graph changes. 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.

2.3 AGENT-BASED APPROACH

The Multi-Agent Spatial Simulations Library, developed by the Distributed System Laboratory at the University of Washington Bothell, facilitates spatial simulations and big data analysis in a parallel environment [13]. MASS Java is built on two primary components: Places and Agents. Places represent individual data members of a larger dataset, while Agents comprise individual agent entities that traverse the Place objects to perform computation. Individual Place objects are distributed across a specified number of computing nodes within a cluster and can be referenced using a globally known set of coordinates. An example of this is shown in Figure 2.2, where each place is mapped using x and y coordinates.

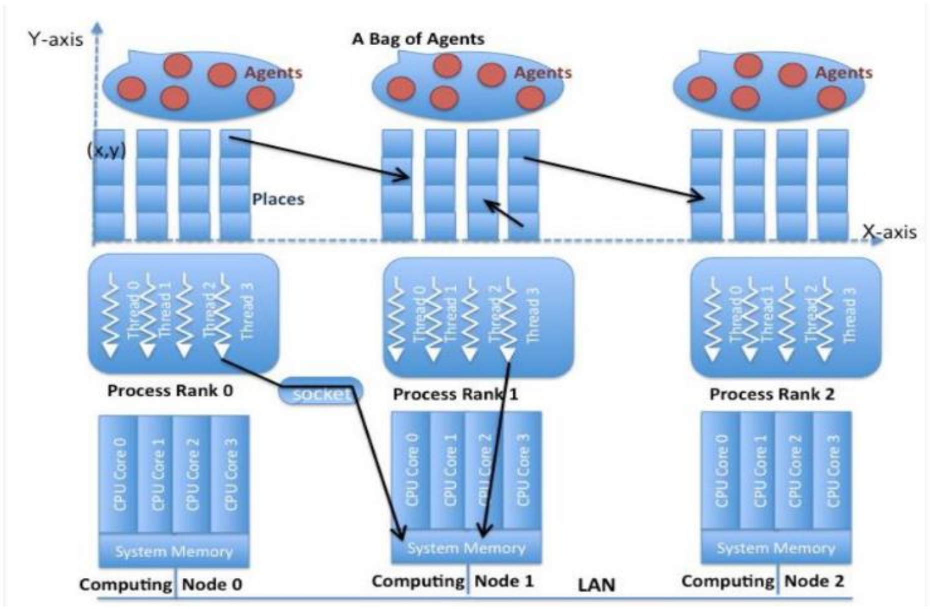


Figure 2.2 MASS Model[14]

After mapping each Place across the cluster, they are assigned to threads and can communicate with other Place objects and the agents residing on them. Agents are stored in groupings on each computing node and can traverse between computing nodes through serialization and TCP communication. In this way, MASS Java facilitates large-scale simulations and big data analysis from the agent-based perspective.

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 VertexPlace 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 the DSL propriety formats DSL and SAR [15]. Loading data from one of the supported file formats has been demonstrated to be incredibly efficient, as the data can be processed in parallel, making graph creation 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.

Despite its effectiveness, the agent-based programming approach in MASS has some drawbacks. In applications like BFS and triangle counting, the approach relies on brute-force methods, distributing a large population of simple reactive agents that lack a structural understanding of the underlying data. While this method is more efficient than serial and other big-data computing frameworks, allowing agents to learn from their interactions with the graph can lead to a more informed and efficient solution for agent-based data analysis.

This project focuses on blending reinforcement learning, specifically Q-learning, to develop agents that can understand the graph's structural composition and, through training, learn to navigate the underlying graph more intelligently.

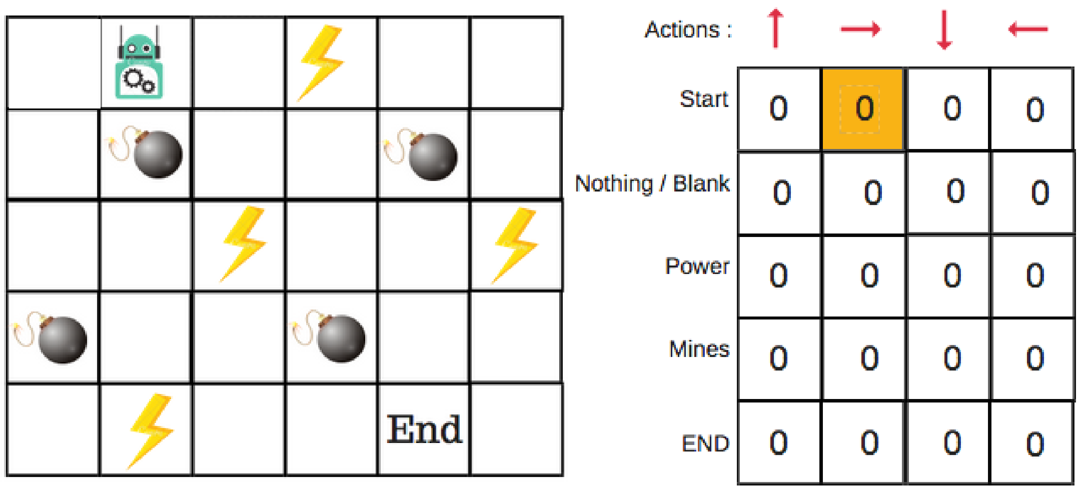
2.4 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where agents learn through interactions with their environment by receiving feedback based on their actions. Unlike supervised learning, which uses labeled examples, or unsupervised learning, which discovers patterns in unlabeled data, RL focuses on learning how agents should behave in a given environment. The ability to train intelligent agents to act autonomously in dynamic environments without complete visibility presents a powerful tool for solving problems in new and efficient ways.

Reinforcement Learning can be formalized as a Markov Decision Process (MDP) [16], which is characterized by several components pertinent to RL:

1. States (S): A set of unique situations that the given environment or system can take. In finite environments, the states encompass all states an environment can be in at any given timestep.
2. Actions (A): A set of possible actions or decisions that can be taken in the given environment and can vary based on the state of the environment or system.
3. Transition Probabilities (P): This represents the probability of transitioning from one state to another when a given action is selected.
4. Rewards (R): The numerical value of a given state-action pair. Rewards provide the agent with feedback when taking a specific action in a given state.
5. Policy (π): Represents the agent's strategy to determine each state's optimal action.

Much like human learning, MDPs occur over a set number of time steps. At each time step, the agent assesses its current state, determines the action to take based on the defined policy, gains experience from the action by transitioning to the associated state, and receives a reward or punishment. For example, in the grid navigation task shown in Figure 2.3, the goal is for the RL agent to navigate to the endpoint while avoiding potential obstacles. At any given state, the agent has four actions to choose from, which can result in their own unique reward. In the agent's current state, if it were to move down, it would encounter a bomb, and the training episode would end. While that individual training episode may end, the reward, or in this case, punishment, the agent receives for encountering that negative state will inform that agent to avoid taking the action that resulted in it encountering the bomb. Over a number of episodes, the agent will learn the optimal policy for navigating the grid and finding the path to the end square.



Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

Figure 2.3 RL Agent Grid Movement [16]

The process illustrated in the previous example can be easily adapted to a number of different environments, including graphs. In this context, the graph's vertices represent states, and the edges represent actions an agent can take to move from one state to another. Each state transition comes with a cost represented by the edge weight. As the agent explores different paths through the graphs, it receives positive rewards for finding the end vertex or negative rewards for encountering dead ends. By repeatedly interacting with the environment and using RL algorithms like Q-learning, the agent can store and update its knowledge about the value of each state-action pair, leading to a gradual learning process where objectives like finding the shortest path can be effectively solved without an explicit model of the environment.

A notable aspect of reinforcement learning algorithms, like Q-learning, is their ability to tackle dynamic environments in graph computing. Traditional algorithms, such as Dijkstra's or A*, are optimized for static structures and will require complete recomputation when faced with graph changes. In contrast, Q-learning continually updates its policy as the agent explores the environment, making it inherently adaptable to any changes in the graph structure. In this way, a Q-learning agent can incrementally adapt to changes in the environment without starting from scratch. The adaptability of Q-learning agents makes them uniquely suitable to environments where changes occur, such as transportation or social networks. For dynamic environments, the agent can leverage previously gained knowledge to train only a fraction of its initial training, thereby improving efficiency and offering a robust solution for dynamic graph computing.

Chapter 3. RELATED WORK

3.1 Q-LEARNING SHORTEST PATH

Q-learning 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. This is particularly important for the shortest path problem, which is utilized in various fields, including robotics [17], navigation [18], and computer network optimization [19].

In the Q-learning shortest path algorithm, graph vertices represent states, and edges represent possible actions. Rewards are constructed around the edge weights, with positive rewards for reaching the destination and negative rewards for arriving at dead ends. As the Q-learning agent navigates the graph, it populates the Q-table, forming a learned understanding of the environment and the optimal navigation strategy. Q-learning's model-free approach makes it a compelling method for dynamic or unknown environments, as it can adapt to the environment as it continues to navigate the graph.

In literature, Q-learning has successfully addressed the shortest path problem. Sun [18] utilized Q-learning to navigate a static city graph city graph, demonstrating its effectiveness in road network-like graph structures. Wang et al. [17], applied Q-learn 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 [19] introduced a novel path-routing algorithm based on a modified Q-learning algorithm for dynamic network packet routing, highlighting its adaptability to changing environments.

While these previously mentioned studies demonstrate Q-learning as an effective tool for enabling intelligent agent-based navigation of unknown environments, they often focus on small, static graphs with fixed hyperparameters on single computing nodes. This 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.

3.2 Closeness & Betweenness Centrality

Closeness centrality is an important measure used in the field of network analysis to determine the importance of a vertex within a network based on its proximity to all other vertices [20]. Closeness centrality is particularly useful in social network analysis [7], communication networks[8], and other scenarios where transfer efficiency across the network is important.

Traditional approaches to calculating closeness centrality involve three steps [9]: computing the shortest path from each vertex *u* to all to all other vertices *v* in the network typically with algorithms such as Dijkstra, summing all the path lengths from vertex *u* to all other reachable vertices in the network, and calculating closeness centrality of a vertex *u* by taking the reciprocal of the sum of the shortest path distances with Equation (1):

$$C(u) = \frac{N-1}{SP\_Sum(u)} \qquad (1)$$

Where N is the total number of vertices in the network

Similar to closeness centrality, betweenness centrality is another measure popular in network analytics. It quantifies the importance of a vertex based on its position as an intermediary in the paths connecting other pairs of vertices. Simplistically, it measures how often a vertex appears on the shortest path between pairs of other vertices in the network [9]. Betweenness centrality is particularly useful for identifying vertices positioned in such a way that they are central connectors or 'bridges' within a network. Betweenness centrality is widely used in various applications such as social networks [9], protein interaction networks [10], and optimization of computer network infrastructure [11].

Calculating betweenness centrality involves determining all shortest paths between each pair of vertices u and v in the network using algorithms like Dijkstra's, Bellman-Ford, Floyd-Warshall, and A*, counting how many paths pass through a given vertex, and determining betweenness centrality using the Equation (2):

$$BC(s) = \sum_{u \neq s \neq v} \frac{\sigma_{uv}(s)}{\sigma_{uv}} \qquad (2)$$

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*.

While these metrics are vital, computation can be time-consuming, especially for dynamic graphs, where metrics must be recalculated with each change. Parallel approaches have been presented to tackle the computational complexity of closeness and betweenness centrality, particularly related to dynamic graphs. Cohen et al. [20] utilize a hybrid estimator to randomly sample vertices to approximate the closeness centrality for all vertices. Estimator-based methods, like the one presented in Cohen, typically use landmark-type techniques, where pivotal vertices are identified and used to base distance calculations to help mitigate skewed distance distributions. Shukla et al. [21] developed a parallel method for determining closeness and betweenness centrality, focusing on bi-connected components for efficient centrality calculation.

The methods presented in the previous studies were effective in reducing the computational burden associated with producing closeness and betweenness centrality, but they come with significant caveats. The landmarking technique employed by Cohen demonstrated that the random selection of landmark vertices was ineffective and that an intensive search for accurate landmark vertices required significant preprocessing. The biconnected component technique in Shukla required the use of high-end GPUs to accelerate the batch processing of graph vertices and a preexisting understanding of the structural properties of the graph to enhance performance. This work focuses on centrality calculations without preprocessing the graph data or specialized hardware.

3.3 Dynamic Graphs

Dynamic graphs, where changes to the graph's vertices or edges occur over time, better emulate real-world situations, like traffic networks where changes occur often and randomly. Changes to the graph can be incremental, adding vertices and edges or increasing weights, or decremental, removing vertices and edges or decreasing weights [22]. Traditional algorithms like Dijkstra's or Bellman-Ford are designed for static graphs as they require complete recomputation for each change that occurs. Recomputation for repeated changes can make traditional algorithms computationally expensive for repeated updates.

Several approaches have been studied to address the challenges of dynamic graphs. Early efforts adapted traditional algorithms like A* for incremental changes. Algorithms like dynamic Dijkstra's [23] and D* [24] were proposed, which utilized incremental search strategies focusing on recomputing only the changed parts of the graph. Subsequent methods used landmark-based techniques [25] for real-time shortest path approximation and distributed techniques [26] to partition the graph data and subgraph computation across a distributed cluster of computing nodes. While these previous methods improved performance, they often rely on knowing where changes occur, which is impractical in real-world applications where changes can occur suddenly and randomly. Additionally, landmarking requires extensive preprocessing to select landmark vertices accurately. This work focuses on measuring dynamic performance without prior knowledge of graph changes or the underlying structure of the graph.

3.4 DIFFERENTIATING MASS Q-LEARNING

1. Adaptive Q-Learning

This paper examines numerous graph sizes, distributed dynamic graphs, and adaptive hyperparameters to optimize the learning process and execution time. It aims to provide a holistic and realistic study of Q-learning applied to agent-based programming, improving on the previously mentioned implementations.

2. Multi-Agent Training

The aforementioned studies on Q-learning for the shortest path problem primarily focus on a single Q-learning agent for the training process. While a single agent simplifies the implementation process, employing multiple agents in the training process can reduce training time and enhance learning processes. This project aims to quantify the effect multiple agents, implemented through the MASS multi-agent framework, can have on the Q-learning process.

3. Agent Adaptability

While works on agent-based shortest path calculations focus on leveraging agents to learn how to properly navigate and determine optimal policies for applications like shortest path, these works fail to quantify how adaptable agents are to changes in their environment. The Q-learning algorithm is inherently adaptable, and we seek to quantify how the Q-learning agents adapt to changes in the environment, what level of retraining is required to adapt to specific changes in the environment, and how the agent responds to these changes in distributed dynamic graphs.

Chapter 4. Q-LEARNING IMPLEMENTATION IN MASS

This chapter describes the Q-learning algorithm design 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.

4.1 ALGORITHM DESIGN

In the Q-learning shortest path algorithm, the environment is represented as a graph $G = (V, E)$ where V represents the graph vertices and E represents the set of edges connecting the vertices. Formulated as a reinforcement learning problem, each vertex corresponds to a state, and edges represent an action leading to a neighboring state.

An agent learns the optimal path through the graph through trial and error, storing knowledge in a Q-table $Q(s, a)$. At each state transition, the Q-value for taking action a in state s is updated. Rewards serve to guide the agent towards the optimal policy, with rewards set to the negative edge cost for non-terminal states, 100 for reaching the destination, and -100 for reaching a dead end.

In Q-learning, hyperparameters are crucial for learning efficiency and convergence. Q-learning has three main hyperparameters: learning rate (α), discount factor (γ), and exploration rate (ε) [18]. The learning rate controls how new information is favored over old information. The discount factor balances immediate and future rewards. Epsilon controls the tendency to explore new paths versus exploiting the knowledge in the Q-table.

4.1.1 *Optimal Path Training & Enumeration*

The training process involves the agent iteratively interacting with the graph environment, receiving rewards, and updating the values stored in the Q-table until the agent has completed the specified number of training iterations or the change in the Q-values reaches convergence. The process includes:

1. Initialization: The agent is initialized at the source vertex with a Q-table consisting of all zero values
2. Action Selection: Using an epsilon-greedy action selection policy, the agent selects an action based on a random value compared to ε. The agents explore if the value is less than ε; otherwise, it exploits the highest Q-value action.
3. Transition and Reward: The agent moves to the next state *s'* dictated by the action and receives the rewards *r*
4. Q-Table Update: Using the reward it received for the state transition, the agent updates the Q-table with a new Q-value using the Q-learning update rule shown in Equation (3):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)] \quad (3)$$

where α is the learning rate, γ is the discount factor, and r is the reward received for transitioning from state *s* to *s'* with action *a*.

5. Iteration: The agent updates its state to s’, and the process repeats until the agent reaches a terminal state in the form of a dead end or destination vertex. At that point, training terminates if the maximum change in Q-values for the episode reaches convergence or the maximum number of training episodes is completed. If those conditions are not met, the agent returns to the source vertex and repeats steps two to four.

After training, the Q-table reflects the agent’s knowledge of the relationship between vertices in the graph. The agent exploits this knowledge to find the shortest path by setting its ε value to 0.0 and navigating from the source to the destination.

4.1.2 *Dynamic Hyperparameter Tuning*

In our Q-learning implementation, we enhance the learning process by incorporating dynamic hyperparameter tuning facilitated by a distributed reward window. The distributed reward is a LinkedList that stores the cumulative reward received by an agent over the course of a training episode. This mechanism allows multiple Q-learning agents to communicate and adjust their hyperparameters based on the collective training progress.

When the reward window reaches its capacity, the agents use this cumulative data to adjust their hyperparameters alpha and epsilon. If the current average reward of any episode is lower than the previous average stored in the reward window, it indicates that the agent’s performance is degrading. Consequently, alpha and epsilon are increased to encourage more exploration and learning from new experiences. Conversely, if the average reward is higher, indicating improved performance, alpha and epsilon are decreased to stabilize learning and promote the exploitation of the knowledge stored in the Q-table.

By utilizing dynamic hyperparameter tuning, agents can adapt their learning strategies based on the progress of the entire agent population, promoting more effective and efficient learning. Unlike methods that employ the fixed decay of hyperparameters, the collective experiences stored in the reward window create more adaptable agents that balance exploration and exploitation to improve convergence and the overall performance of the Q-learning algorithm.

4.1.3 *Suboptimal Path Enumeration*

The Q-table captures comprehensive knowledge of the graph, including multiple optimal and suboptimal paths. This comprehensive knowledge allows for solving problems like closeness and betweenness centrality. By deploying multiple agents and controlling their navigation through control parameters for Q-value and path threshold, the agents can explore paths within a certain percentage of the optimal Q-value and path length, enabling them to find viable routes beyond the optimal path.

The knowledge stored in the Q-table can be further extended to enumerate suboptimal paths beyond just the shortest path. As the agent iteratively explores the graph, it updates the Q-table not only in optimal state transitions but also in suboptimal state transitions. While many of the paths the agents will explore will be well outside of the optimal path, there are many suboptimal paths that can present promising routes if the graph's structure or conditions were to change. When sufficiently trained, the Q-table's values can be considered a holistic view of the graph's navigational options, providing many potentially viable routes beyond the optimal path. To enable the enumeration of the suboptimal path, we define two control parameters: Q-value threshold and path threshold. In any given state, the Q-value threshold controls the agent's tendency to explore Q-values within a percentage of the optimal Q-value in that state. A larger threshold allows for more potential paths to be explored. The path threshold controls the percentage length away from the already determined optimal path. While the Q-value for a given state transition may be within a percentage of the optimal Q-value, the path produced in subsequent states may not be worth exploring. The path threshold allows us to control how deep an agent will explore before determining the path to be not viable.

The comprehensive knowledge provided in the Q-table provides a significant benefit in dynamic graph environments. If the shortest path becomes unsuitable, the agent can adapt to the change by selecting an alternative route from the Q-table. The holistic knowledge of the graph ensures the agent can reach its goal even when it must depart from the optimal path in response to changes in the environment.

4.1.4 *Closeness & Betweenness Centrality Extension*

Utilizing the Q-learning approach to find the shortest path outlined in the previous section, a similar approach can be implemented to find closeness centrality and betweenness centrality.

Closeness centrality requires finding the shortest path between each vertex in the graph. To do this with Q-learning, we deploy a Q-learning agent for each vertex-to-vertex pair to find the shortest path. This would present an issue in a typical Q-learning scenario as it would require the storage and maintenance of multiple different Q-tables for each vertex-to-vertex pairing. By leveraging the MASS library's distributed capacity, we can store multiple distributed Q-tables at each vertex. This is accomplished by storing a two-dimensional array of HashMaps at each vertex, which can be accessed by the source and destination pairing for each agent. After all agents have completed training, they enumerate the final path and return it to the main program for individual vertex closeness centrality calculation.

Building off the approach of closeness centrality, betweenness centrality operates in a similar way but requires all the shortest paths between each vertex-to-vertex pairing in the graph. After the agents have completed training and populated their respective Q-tables for their source and destination vertex pairing, we are left with the information necessary to enumerate the optimal path and all optimal paths in the graph. By leveraging the threshold pathing extension mentioned in Section 4.1.2, we can enumerate all the optimal paths by deploying a path agent for each source-to-destination vertex pair with a Q-value threshold of 25% and a path threshold of 0%, indicating we are only looking for optimal paths. After all PathAgents complete enumeration, we will be left with all the optimal paths for each source-to-destination vertex pairing. With this information, we can then calculate the betweenness centrality for each vertex in the graph.

4.2 IMPLEMENTATION IN MASS

4.2.1 *Environment*

The fundamental building blocks of programs in MASS are Places and Agents, with GraphPlaces being particularly relevant for graph environments. As mentioned in Section 2.3, GraphPlaces extends the traditional Placess object in MASS, consisting of VertexPlace objects for graph-based computing. For the Q-learning shortest path in MASS, the VertexPlace class is further extended by the Node class to enable specific Q-learning functionalities. Beyond the traditional elements provided by VertexPlace, such as neighbors and their associated edge weights, the Node class includes a sparse Q-table representation for the neighbors at each graph vertex and hosts the distributed reward window at the destination vertex. Node, and by extension MASS, plays a crucial role in facilitating communication between agents in multi-agent training.

4.2.2 *Q-Learning Agent*

The Q-learning agent, dubbed QLAgent in MASS, is the primary agent class used in the Q-learning shortest path process. The goal of the agent is to iteratively navigate the graph and population the Q-table based on the previously mentioned Q-value update function. The QLAgent trains until the Q-values have reached convergence or the specified number of training iterations has been completed. Once training is complete, its job is to derive the optimal path based on the knowledge it gained during training. The QLAgent is designed to operate autonomously from the main program and learn which sequence of vertices constitutes the optimal path.

On inception, the QLAgent is provided with seven key pieces of information needed to conduct the Q-learning process: source vertex, destination vertex, alpha, gamma, epsilon, number of training episodes, and the sliding window size. The source vertex and destination vertex provide the agent with its start and end goal as it navigates the graph; alpha, gamma, and epsilon are the hyperparameters used in the Q-learning update process and for controlling navigation; training episodes are used to define the maximum length of training; and the sliding window size is used to control the size of the reward window used to tune the hyperparameters alpha and epsilon.

The QLAgent operates on two main functions typical of MASS agent implementations: *departure* and *onArrival*. The departure is used for the action selection process. Here, the agent pulls its current vertex from MASS using the *getPlace()* function, which gives the agent access to the information stored at the current vertex, including the neighboring vertices, their edge weights, and associated Q-values. The QLAgents then adds the current vertex to its list of visited vertices and invokes the *chooseAction()* function shown in Algorithm 4.1. The *chooseAction()* function utilizes the epsilon hyperparameter to choose between random exploration or Q-table exploitation. After the action is selected, the agent moves to the next vertex with the MASS *migrate()* function.

```
Algorithm 1: chooseAction
  Result: Action to take (neighbor node or -1)
  node ← getPlace();
  neighbors ← filter() and map() node's neighbors to a list of integers;
  remove() visitedNodes from neighbors;
  if neighbors is empty then
      return -1 // No unvisited neighbors, need to backtrack
  if random() value < epsilon then
      // Exploration:  Choose a random action
      return a random neighbor from neighbors
  else
      // Exploitation:  Choose the best action based on
         Q-values
      return neighbor with the maximum Q-value
  end
```

Algorithm 4.1 chooseAction() Function in QLAgent

After migration, the QLAgent moves to the *onArrival()* function, which updates the Q-value for the state transition and evaluates current agent performance. To update the Q-value, the QLAgent calls the *updateQValue()*, passing the reward, its neighbors, and the previous Q-value. The *updateQValue()* function utilizes the Q-value update function to generate a new Q-value. Since the Q-table is distributed across the Node Place objects, the agent then spawns a child agent to traverse back to the previous vertex and update the Q-value stored for the vertex the QLAgent currently resides on. After spawning the update agent, the agent then evaluates its current position. If the agent is in a terminal state, either at the destination vertex or a dead end, the episode is over, and the agent evaluates its current performance. To do this, it adds the total of all the rewards for the episode to the reward window and invokes the *adjustAlphaEpsilon()* function to tune the hyperparameters. The *adjustAlphaEpsilon()* function shown below in Algorithm 4.2 adjusts the alpha and epsilon parameters based on the average rewards the agent received over the episode compared to the previous episodes. If the current average reward is less than the previous average reward, agent performance is degrading, and alpha and epsilon are increased by 3.5% to ensure more exploration and long-term planning. If it is less, alpha and epsilon are decreased by 5% to encourage the agent to exploit its knowledge and focus on immediate rewards.

```
Algorithm 2: Adjust Alpha and Epsilon
Input: rewardSize, slidingWindowSize, rewardSum, rewardWindow,
       rewardIndex, currentEpisode, epsilon, alpha
Output: Adjusted epsilon and alpha
if rewardSize == slidingWindowSize then
    currentAverageReward ← rewardSum / rewardSize;
    if currentEpisode ≥ slidingWindowSize then
        previousSum ← rewardSum - rewardWindow[rewardIndex];
        previousAverageReward ← previousSum / (rewardSize - 1);
        if currentAverageReward ¡ previousAverageReward then
            epsilon ← min (1.0, epsilon * 1.035);
            alpha ← min (0.5, alpha * 1.035);
        end
        else
            epsilon ← max (0.01, epsilon * 0.95);
            alpha ← max (0.01, alpha * 0.95);
        end
    end
end
```

Algorithm 4.2. adjustAlphaEpsilon() Function in QLAgent

4.2.3 *Path Agent*

In MASS, we utilize the PathAgent to enumerate optimal and suboptimal paths. The PathAgent is a simplified version of the QLAgent, whose main goal is multiagent path enumeration. On initialization, the PathAgent takes in the source vertex, destination, optimal path length, Q-value threshold, and path threshold.

Path enumeration begins with an initial PathAgent at the source vertex. The PathAgent first invokes the departure function to begin navigation. In the departure method, the PathAgent first checks if its current path length is within the defined path threshold. If the path is within the threshold, the agent gets its current vertex and its neighbors. The PathAgent then invokes the *largestQValueThresold()* function, which returns a list of neighbors within the Q-value threshold in descending order. The parent agent sets its migration destination to the first value, which is the destination with the largest Q-value. If there is more than one neighbor in the list, the PathAgent spawns additional child PathAgents with their next vertex set to the potential other paths.

In the *onArrival* function, the PathAgent gets its current vertex and checks if its next vertex ID is equal to the vertex ID it currently resides on. This check is necessary, as the child agent will still reside at the previous vertex and will need to migrate to the next vertex. If the PathAgent is on the correct vertex, it adds the vertex to its path and checks if it is within the path threshold. If the agent is outside the path threshold, it immediately terminates with the *kill()* function. If it is within the threshold, it checks if it is at the destination, where it will return the path to the main program, or it will wait for the next time step.

In the end, only PathAgents with paths that are within the path threshold will return their path to the main program. The return from the agents constitutes the enumeration of all optimal paths and suboptimal paths within the defined threshold stored in the Q-table. The multi-agent approach, coupled with the knowledge from the Q-table, allows us to enumerate available paths in the graph intelligently.

4.2.4 *MASS Performance Improvements*

1. Distributed & Sparse Q-table:

A notable limitation of Q-learning on large graphs is the creation and maintenance of a large Q-table. In Q-learning, the Q-table needs to store the value for every possible state-action pair, which is typically stored in a 2D double array sized for two times the number of vertices. When the graph has a large number of vertices and edges, this results in an exponentially growing Q-table, which requires a considerable amount of memory and computational resources to store.

To address this challenge, we leverage the distributed memory capabilities of MASS to distribute the Q-table across the Node Place objects, effectively sharing the capacity load across all computing nodes in the cluster. The distributed approach also enables multiple agents to access and make parallel updates, significantly speeding up the training process. In addition, by utilizing a HashMap linked to the neighbors at the given vertex and their associated Q-values, we create a sparse representation of the Q-table to enhance efficiency further. The combination of distributed computing and the sparsity of the Q-table ensures that the Q-learning algorithms remain practical and efficient for large-scale graphs.

2. Multi-Agent Training:

As the name implies, MASS is designed and particularly adept at facilitating multi-agent applications. Given this feature, we leveraged MASS's easy-to-use multi-agent functionality to add multiple agents to the training process. By utilizing multiple agents, we can populate the Q-table more efficiently, reducing training time and exploring the graph more thoroughly than with a single agent.

3. Optimized Agent Data Structures:

An important aspect of developing agent applications in MASS is ensuring that the agents are as memory-efficient as possible. When agents move between computing nodes in the cluster, they must be serialized and sent over the network to access data stored on those remote computing nodes. As the Q-learning process is an iterative and in-depth exploration process, agents must frequently move between remote computing nodes. The initial version of the Q-learning shortest path in MASS demonstrated that native Java data structures proved inefficient when multiple serialization and network transportation operations were required. To improve the efficiency of agent movement, all native Java data structures were converted to FastUtil structures, which are significantly more efficient in terms of their memory footprint.

Chapter 5. EVALUATION

This chapter presents the experimental results of the updated MASS and Q-learning-based shortest path, closeness centrality, and betweenness centrality applications. All implementations were tested on the CSSMPI cluster detailed in Section 5.1. This chapter follows the naming convention where "MASS Original" refers to the previous MASS applications utilizing an emulated graph distributed evenly over a distributed array, and "MASS Updated" refers to the updated previous MASS applications that utilize the latest version of MASS with GraphPlaces and GraphAgent, and MASS QL refers to the Q-learning-based applications introduced in Section 4.

The results for the MASS QL applications are divided into individual sections for each of the applications and include two subsections for static and dynamic benchmarking. Static benchmarking assesses performance with an unchanged graph, while dynamic benchmarking involves two different vertex removal operations. For the shortest path, we assess the performance of the application when faced with a single vertex and five-vertex removal from the previous shortest path. Static benchmarking is performed on the synthetic dataset, while dynamic benchmarking is performed on the synthetic with one-vertex removal and on the road network datasets with one and five-vertex removals.

5.1 Execution Environment

Experiments were conducted on the CSSMPI cluster of 8 computing nodes provided by the University of Washington Bothell. Each computing node is a virtual machine allocated four cores at 2.10GHz from the 16 cores available on an Intel Xeon Gold 6130, with 16GB of system memory. The "MASS Original" applications utilize MASS Java core version 1.0, the "MASS Updated" applications utilize MASS Java core version 1.4.3, and the MASS Q-learning applications utilize MASS Java core version 1.4.3 and Fastutil version 8.5.8.

5.2 INPUT DATASETS

This section outlines the unweighted graph utilized in testing all applications included in this paper and how it is created. The graphs are divided into three distinct sets: synthetic shortest path, road network, and synthetic centrality graphs.

5.2.1 *Synthetic Shortest Path Graphs*

Table 5.1 lists six unweighted and undirected graphs used to conduct experiments on the shortest path applications. The synthetic graphs are generated using the GraphGen and Map class present in the "MASS Original" shortest path application, which is detailed in Appendix A. The random graph generator creates vertices and edges based on the specified number of vertices, with each vertex having up to 30% of the total vertices as neighbors. The generator includes functionality to ensure that self-loops and duplicated edges are avoided. This results in densely populated graphs with linear growth of edges compared to vertices and is useful for evaluating the performance of the applications when faced with large and dense networks.

Table 5.1 Synthetic Graph Dataset

Vertices	Edges	Avg Degree	Max Degree	Minimum Degree
500	63300	126.6	189	57
1000	250638	250.6	383	110
2000	1020496	510.25	747	248
4000	4018172	1004.5	1470	500
8000	16442720	2055.3	2951	1044
16000	64599688	4037.5	5833	2072

5.2.2 *Road Network Graphs*

To evaluate the shortest path applications in realistic settings, we consider five road network graphs. Table 5.2 outlines the five road network graphs, which include generated graphs of the four Washington cities and the Rome99 graph modeled after the Roman road network. Compared to the synthetic graphs presented in section 5.2.1, these graphs are far less dense and significantly less connected.

With the exception of the Rome road network graph, the city graphs are created using the Python ‘osmnx’ library[27] to download and process real network information for the desired cities. The data is sourced from OpenStreetMap[28], which provides geographic data and mapping. ‘osmnx’ queries the specific city data from OpenStreetMap API and retires drivable road network information. While the data originally comes with labeled vertex IDs from OpenStreetMap, the downloaded graph is remapped to sequential IDs starting from 0 to ease in use and importation into MASS. Edges are assigned a uniform weight of one and made bidirectional, and graph data is written to file in the DSL format. The full script is provided in Appendix B**.**

The Rome99 graph originates from the 9th DIMACS Implementation Challenge and serves as a benchmark for evaluating network analysis algorithms, specifically the shortest path [29]. The graph represents a large portion of the road network in Rome, Italy, and helps diversify the road network graphs featured in this paper.

Table 5.2 Road Network Graph

City	Vertices	Edges	Avg Degree	Max Degree	Min Degree
Bothell	1861	4293	2.35	5	1
Rome	3352	17740	5.3	10	2
Tacoma	6801	19786	3.01	6	1
Spokane	8899	25991	3.15	6	1
Seattle	19096	50331	3	7	1

5.2.3 *Synthetic Centrality Graphs*

Table 5.3 lists the six unweighted and undirected synthetic graphs used for benchmarking the closeness and betweenness centrality applications. The centrality graphs are generated with the same graph generator outlined in Section 5.2.1.

Table 5.3 Synthetic Centrality Graphs

Vertices	Edges	Avg Degree	Max Degree	Min Degree
8	16	2	3	1
16	52	3.25	5	2
32	208	6.5	13	2
64	960	15	24	6
128	3912	30.56	50	10
256	16270	63.55	97	29

5.3 SHORTEST PATH

The Q-Learning shortest path performance was measured using the synthetic graph dataset consisting of 500 to 16,000 vertices for the static benchmarking and the road network graph dataset consisting of 1861 to 19,096 vertices for the dynamic benchmarking. The performance was gauged over eight computing nodes, and the hyperparameters outlined in Table 5.4 were used. For the static benchmarking, the evaluation is divided into training time performance and the optimal output performance compared to the MASS Updated shortest path performance.

Table 5.4 Static MASS QL Hyperparameters

Graph Size	Alpha	Gamma	Epsilon	Window Size	Max Iterations	# Agents - 1 Computing Node	# Agents - 2 Computing Nodes	# Agents - 4 Computing Nodes	# Agents - 8 Computing Nodes
500	0.5	0.9	1	12	10000	2	50	150	400
1000	0.5	0.9	1	20	10000	12	25	100	400
2000	0.5	0.9	1	40	10000	8	100	125	400
4000	0.5	0.9	1	80	10000	8	150	600	800
8000	0.5	0.9	1	120	10000	25	150	200	1750

16000	0.5	0.9	1	200	10000	50	150	250	1750

5.3.1 *Static Graphs*

Table 5.5 MASS QL Shortest Path Training

Vertices	1 Computing Node	2 Computing Nodes	4 Computing Nodes	8 Computing Nodes
500	1130	13595	26391	30829
1000	2456	34816	56556	66320
2000	6916	81840	88763	87398
4000	55230	681969	581240	371308
8000	264130	3542149	3141897	1131391
16000	OOM	4739915	3945661	1760854

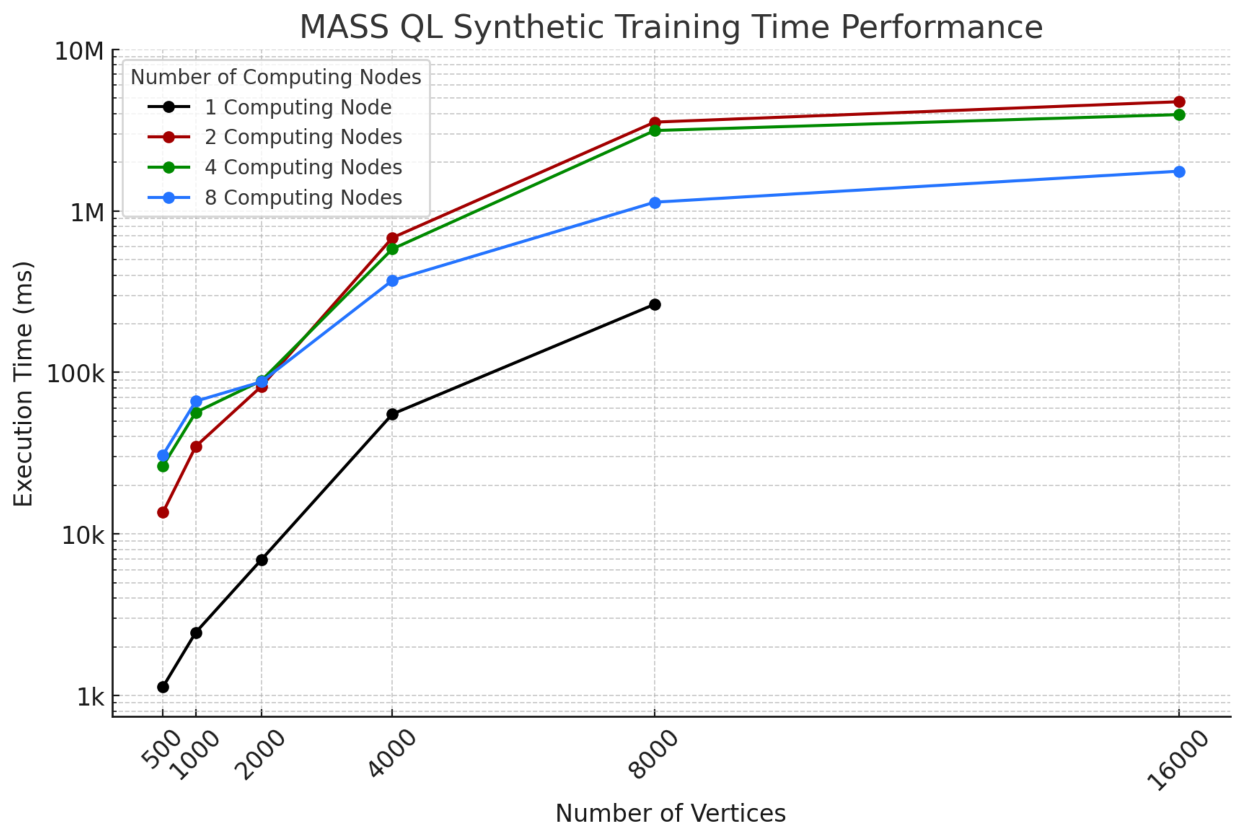


Figure 5.1 MASS QL Shortest Path Synthetic Training

The MASS QL shortest path application consists of two phases: training and path output. Figure 5.1 and Table 5.5 show the application's training performance over eight computing nodes. The application performs best on a single computing node up to the 8000-vertex synthetic graph, beyond which system memory is exhausted. The single computing node performs best because all MASS Node objects reside on a single computing node, and agents do not have to serialize and transmit over the network to migrate to remote computing nodes. For the 16,000-vertex graph where only multi-node executions are feasible, eight computing node executions are optimal. With more computing nodes in the cluster, we can add more training agents before seeing a performance decrease. On eight computing nodes, we were able to scale up to 1750 training agents compared to 150 on two computing nodes and 200 on four computing nodes. The multi-agent training on eight computing nodes represents a performance increase of ~63% compared to two-node performance and ~55% compared to four-node execution.

Table 5.6 MASS QL Shortest Path Output

Vertices	1 Computing Node	2 Computing Nodes	4 Computing Nodes	8 Computing Nodes
500	1	156	302	459
1000	1	186	343	477
2000	2	113	344	421
4000	3	189	322	409
8000	3	191	347	441
16000	OOM	231	391	476

Table 5.7 MASS Updated Shortest Path

Vertices	1 Computing Node	2 Computing Nodes	4 Computing Nodes	8 Computing Nodes
500	521	780	886	1391
1000	948	1337	1264	1438
2000	3791	4547	3680	3095
4000	9776	14061	13103	9286
8000	38703	32293	35407	42397

16000	OOM	OOM	55584	67515

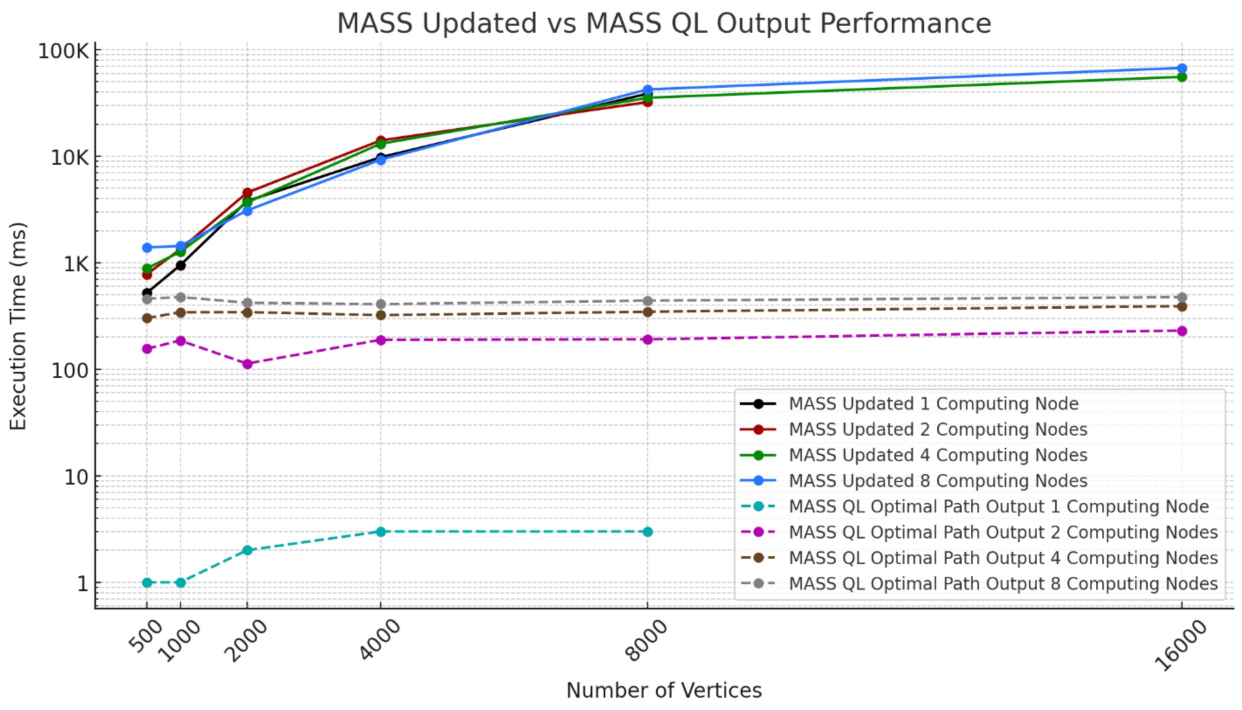


Figure 5.2 MASS Updated vs MASS QL Synthetic Shortest Path Output

After the Q-learning algorithm completes the training phase and populates the Q-table, its performance is extremely fast across all computing node executions. As shown in Figure 5.2 and Table 5.6, when a trained Q-learning shortest path algorithm is compared to the MASS Updated shortest path execution, shown in Table 5.7, the trained Q-learning agent can quickly migrate to the graph vertices that constitute the shortest path between the trained source and destination graph vertex.

5.3.2 *Dynamic Graphs*

For dynamic performance, we measure the Q-learning shortest path's performance on single-vertex removal and five-vertex removal. The single-vertex removal is designed to simulate a minor traffic accident, and the five-vertex removal is designed to simulate a large traffic disruption.

The MASS Updated performance is gauged over two runs of the application: one to determine the original optimal path and the second to respond to the change in the graph. The Q-learning shortest path application utilizes a previously trained Q-table for the source vertex and destination vertex and retrains to respond to the change in the graph.

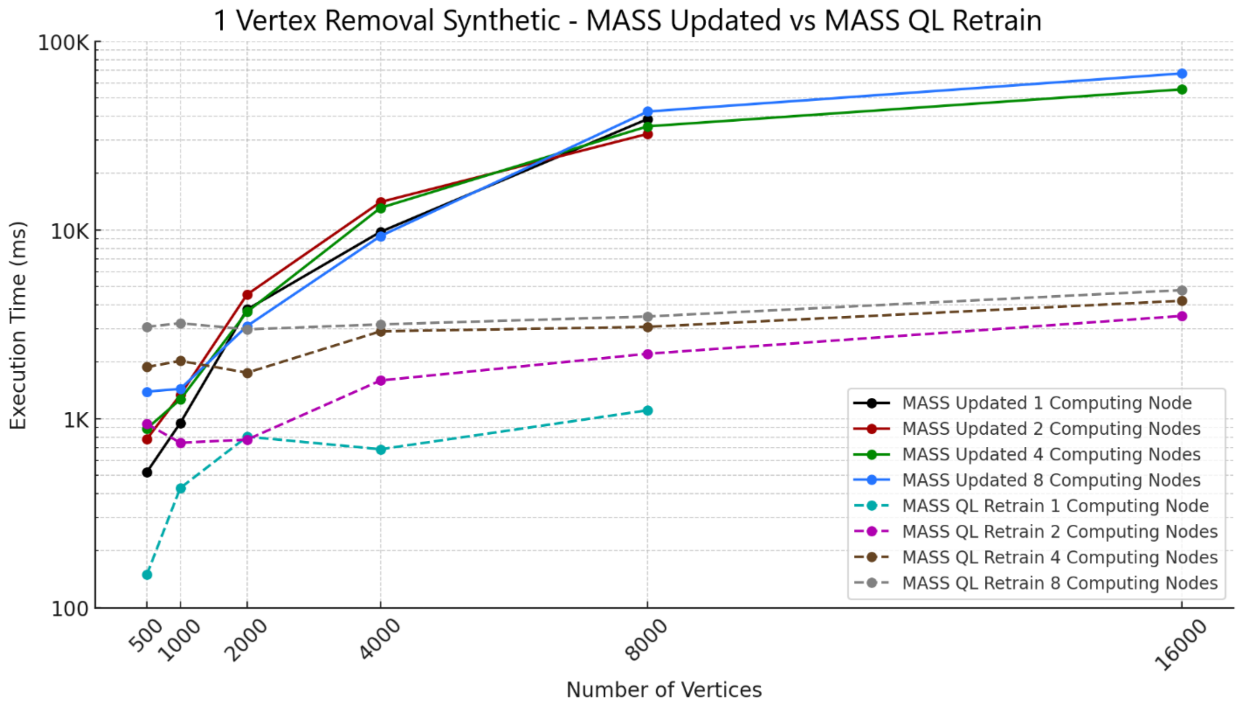


Figure 5.3 1 Vertex Removal MASS Updated vs MASS QL Retrain Synthetic

Figure 5.3 details the performance of the Q-learning shortest path algorithm compared to the MASS Updated shortest path algorithm with one vertex removal on the synthetic dataset. The results here demonstrate that the MASS QL implementation can adapt quickly to a single vertex removal in the synthetic graphs significantly quicker than a second run of the MASS Updated algorithm. A primary reason for this lies in the nature of the graph. The synthetic graphs are incredibly well connected. The Q-table has several different paths already trained, and the QLAgent quickly adapts to the removal of a single vertex. As we will see in the road network dataset dynamic testing, when the structure of the graph is different, the ability to adapt changes.

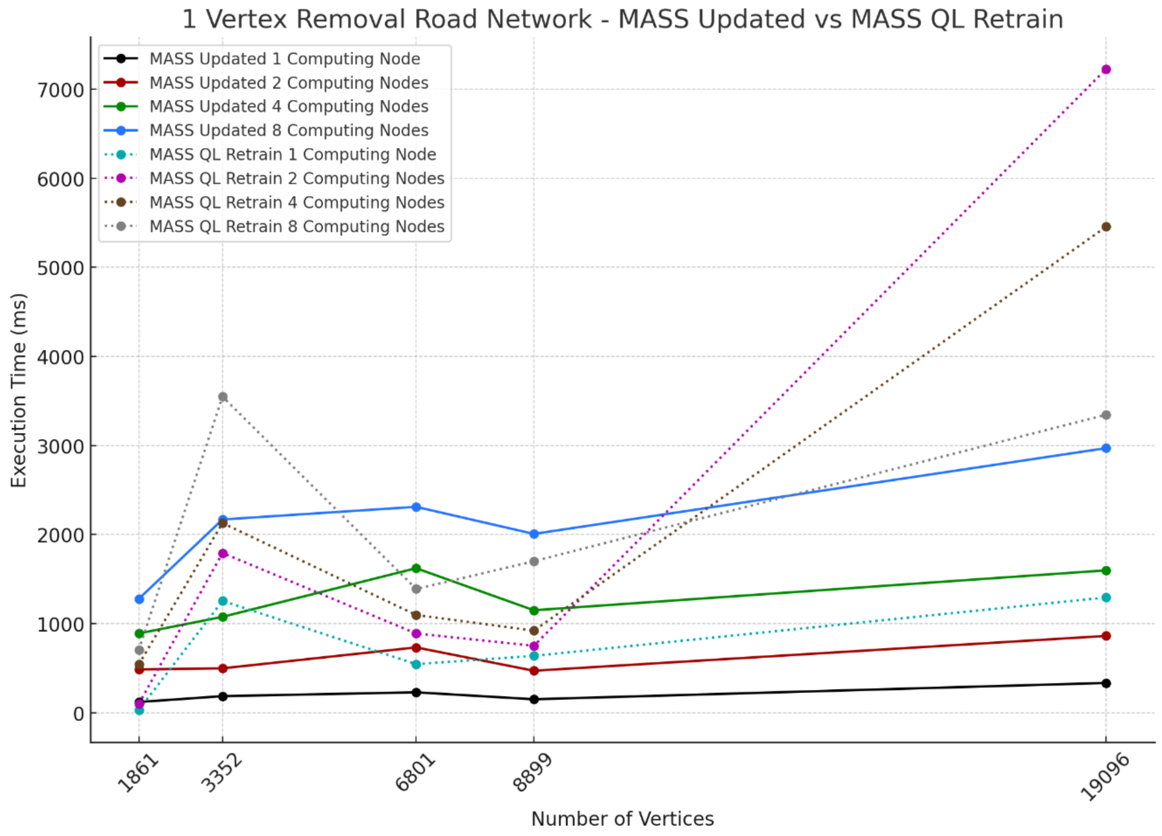


Figure 5.4 1 Vertex Removal MASS Updated vs MASS QL Retrain Road Network

Figure 5.4 details the performance of the Q-learning shortest path algorithm compared to the MASS Updated shortest path algorithm for a single vertex removal on the road network dataset. As the figure demonstrates on smaller graph sizes, the Q-learning shortest path algorithm can leverage multi-agent training to quickly adapt to the change and produce a new optimal path exceeding the performance of the MASS Updated shortest path application. As the graph grows in size, the need for repeated training and more training agents increases the execution time, leading to the MASS Updated shortest path application performing better than the Q-learning implementation.

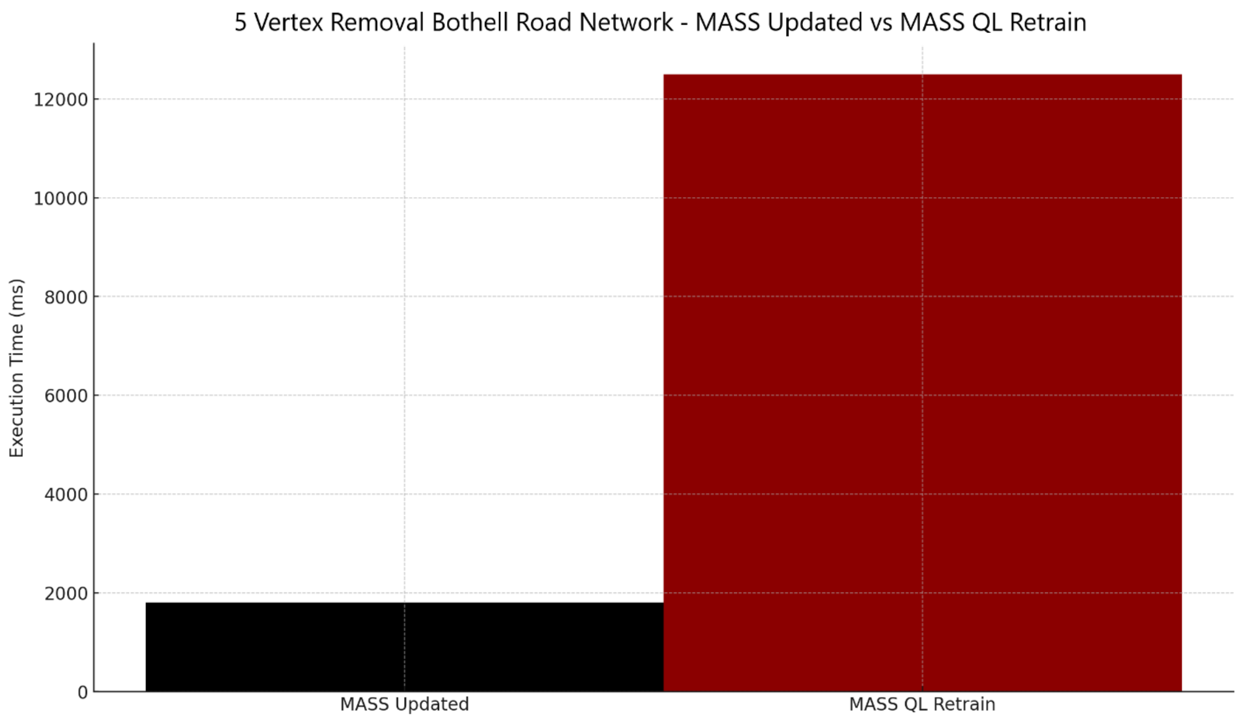


Figure 5.5 5 Vertex Removal MASS updated vs MASS QL Retrain

Figure 5.5 shows the performance of the MASS Updated compared to MASS QL with five vertex removal. The results demonstrate that while MASS QL is able to adapt to one-vertex removals fairly well and competitively, larger changes provide an issue as the values in the Q-table become more affected by the change. Significant changes in the graph require far more training to find a new optimal path in the modified graph. While the amount of training is considerably less than the original training time, the retraining time still rises to the level where rerunning the MASS Updated application is more appealing if execution time is the primary concern. Given these results on the smallest road network graph, larger graphs were not tested.

5.3.3 *Discussion*

The MASS Q-learning shortest path application demonstrates the innate ability to quickly guide agents to graph vertices that constitute the shortest path quickly once trained. The main challenge with the application is that the training time in MASS can rise quickly as graph size increases. The need for repeated traversal over the graph to understand its structure looms heavily over the total execution time.

When faced with dynamic changes to the graph, Q-learning enables agents to retrain quickly to adapt to small changes in the graph. The adaptation capabilities of Q-learning only appear to be more worthwhile when the amount of change in the graph is small. As we saw from the road network benchmarking, a single vertex change could be adapted to quickly. However, when five vertices were changed, this required significantly more training, resulting in the MASS Updated shortest path application being more efficient if run twice in response to the changes.

5.4 Closeness Centrality

Performance for closeness centrality was evaluated on the synthetic centrality graph dataset, which has graphs with vertices ranging from 8 to 256. The performance of the MASS Updated and MASS Q-learning closeness centrality applications was run over eight computing nodes, and the MASS Q-learning application utilized the hyperparameters listed in Table 5.8.

Table 5.8 MASS QL Closeness Centrality Hyperparameters

Graph Size	Alpha	Gamma	Epsilon	Window Size	Max Iterations	# Agents
8	0.5	0.9	1	5	10000	28
16	0.5	0.9	1	5	10000	120
32	0.5	0.9	1	5	10000	496
64	0.5	0.9	1	6	10000	2016
128	0.5	0.9	1	13	10000	8128

5.4.1 *Static Graphs*

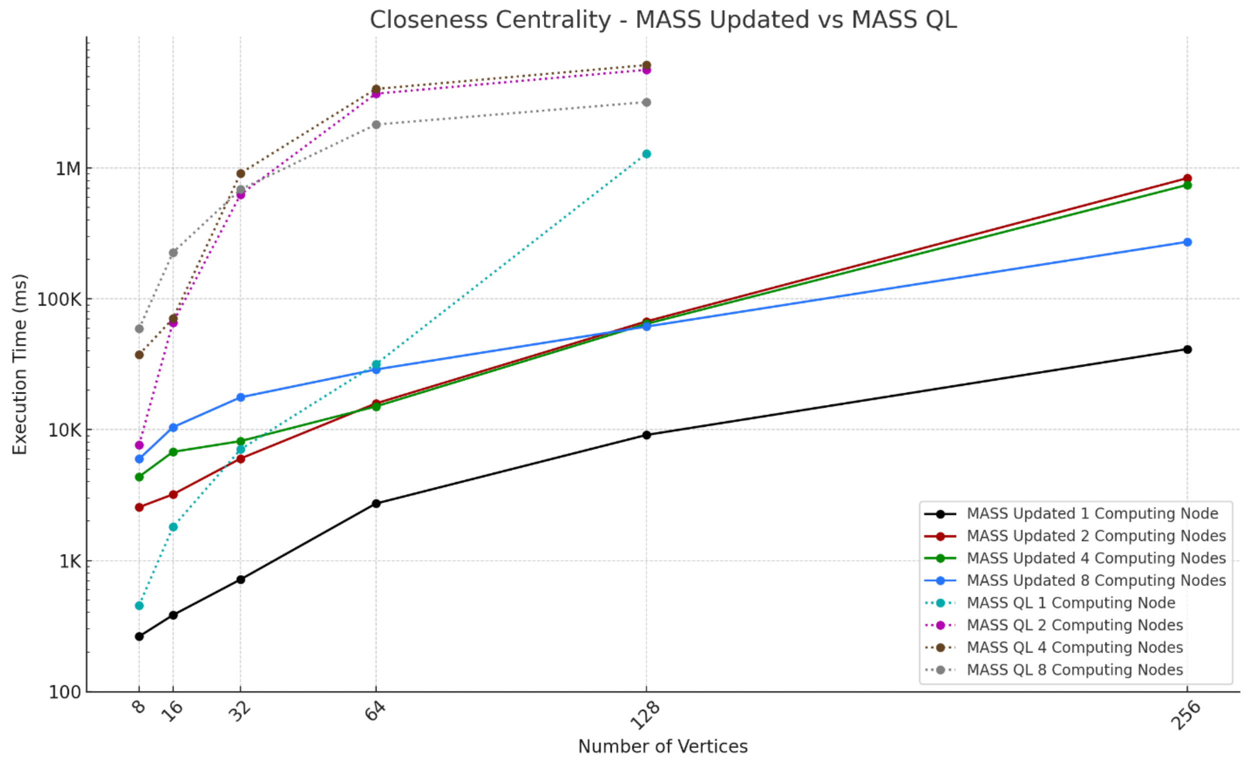


Figure 5.6 MASS Updated vs MASS QL Closeness Centrality Performance

In Figure 5.6, we can see the performance of the MASS Q-learning closeness centrality application compared to the MASS Updated closeness centrality application. The observation made here is that while the Q-learning application achieves performance comparable to the MASS Updated application initially, as the graph size increases, the performance of the MASS Updated application outperforms the Q-learning application when training time is considered. A primary reason for this performance difference comes from the need to train individual Q-tables for each vertex-to-vertex pairing in the graph. As more graph vertices are added, there are more vertex pairs to train on, driving up the overall execution time significantly.

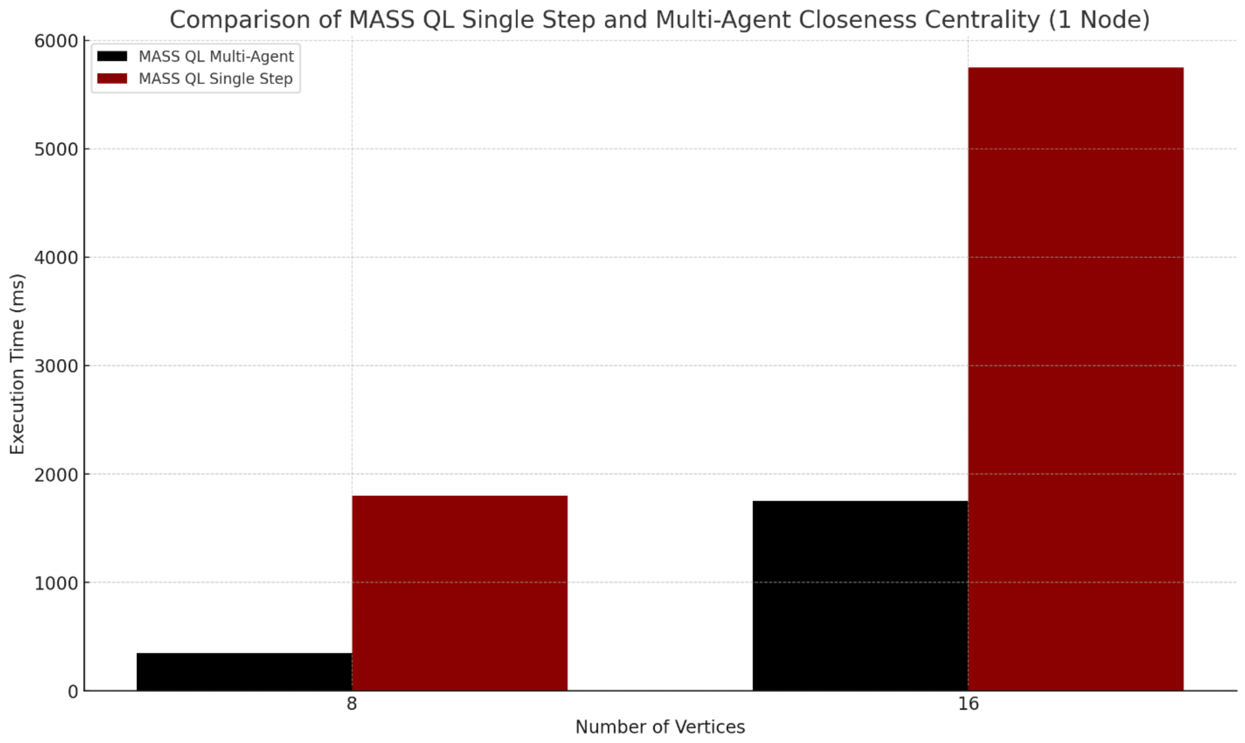


Figure 5.7 MASS QL Closeness Centrality Multi-Agent Scaling

While the performance of the MASS QL closeness centrality application falls short of the current MASS closeness centrality implementation, the multi-agent capabilities of MASS allowed us to scale the application more effectively. Shown in Figure 5.7 of the one computing node MASS QL closeness centrality application, the use of the multiple agents to simultaneously train all vertex pairings in the graph demonstrated a 75% performance increase for the 8 vertices graph and a 69% increase for the 16 vertices graph.

5.4.2 *Dynamic Graphs*

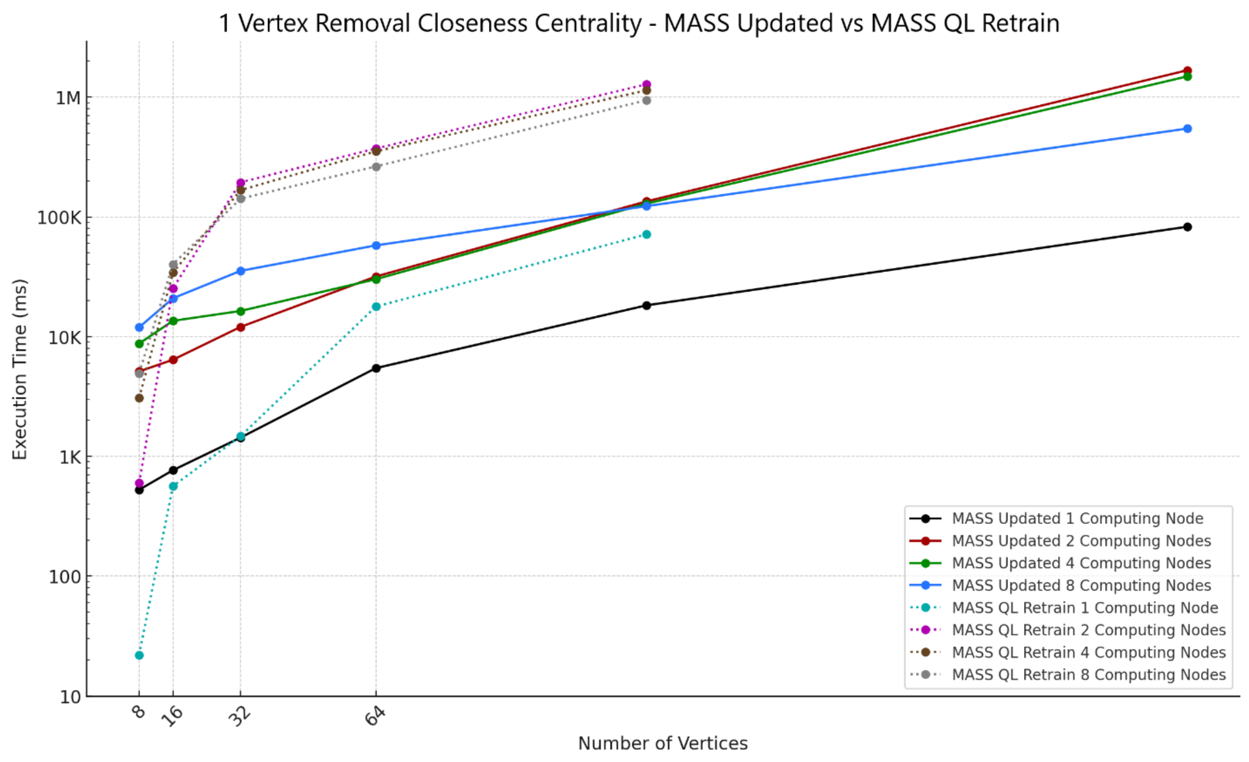


Figure 5.8 1 Vertex Removal Closeness Centrality MASS Updated vs MASS QL

Figure 5.8 shows the performance of the Q-learning closeness centrality application compared to the MASS Updated closeness centrality application when faced with the removal of one graph vertex. At eight to 16 graph vertices, the Q-learning application shows the ability to adapt to the change at a faster rate than a complete return of the MASS Updated implementation. As the graph sizes increase, the overhead associated with retraining all the vertex pairs causes the overall execution time to fall short of simply rerunning the MASS Updated version. In this scenario, because the graph change is unknown to the application, we are forced to retrain all the vertex pairs to ensure that the graph changes are updated across all the potential shortest paths.

5.4.3 *Discussion*

When reviewing the MASS Q-learning closeness centrality application, the implementation shows some issues as the graph size increases. The overhead associated with training all vertex pairings in the graph causes the execution to rise dramatically as the number of vertices increases. While the performance is acceptable in small graph sizes, the training time associated with all vertex pairings makes it ill-suited to larger graph sizes, particularly when they are distributed across multiple computing nodes. In addition, the ability of Q-learning to adapt to changes in the graph is made difficult when the changes are unknown. When a vertex is removed, many of the previous paths in the graph do not change and would not require retraining, but since these changes are unknown, we are forced to retrain to ensure that the previous shortest path is correct.

5.5 Betweenness Centrality

The performance of the betweenness centrality application was evaluated on the synthetic centrality dataset on graphs numbering from eight to 128 vertices. The performance of the MASS Updated and MASS Q-learning betweenness centrality applications was run over eight computing nodes, and the MASS Q-learning application utilized the hyperparameters listed in Table 5.9.

Table 5.9 MASS QL Betweenness Centrality Hyperparameters

Graph Size	Alpha	Gamma	Epsilon	Window Size	Max Iterations	# Agents	Q-value Threshold	Path Threshold
8	0.5	0.9	1	5	10000	28	25	0
16	0.5	0.9	1	5	10000	120	25	0
32	0.5	0.9	1	5	10000	496	25	0
64	0.5	0.9	1	6	10000	2016	25	0

128	0.5	0.9	1	13	10000	8128	25	0

5.5.1 *Static Graphs*

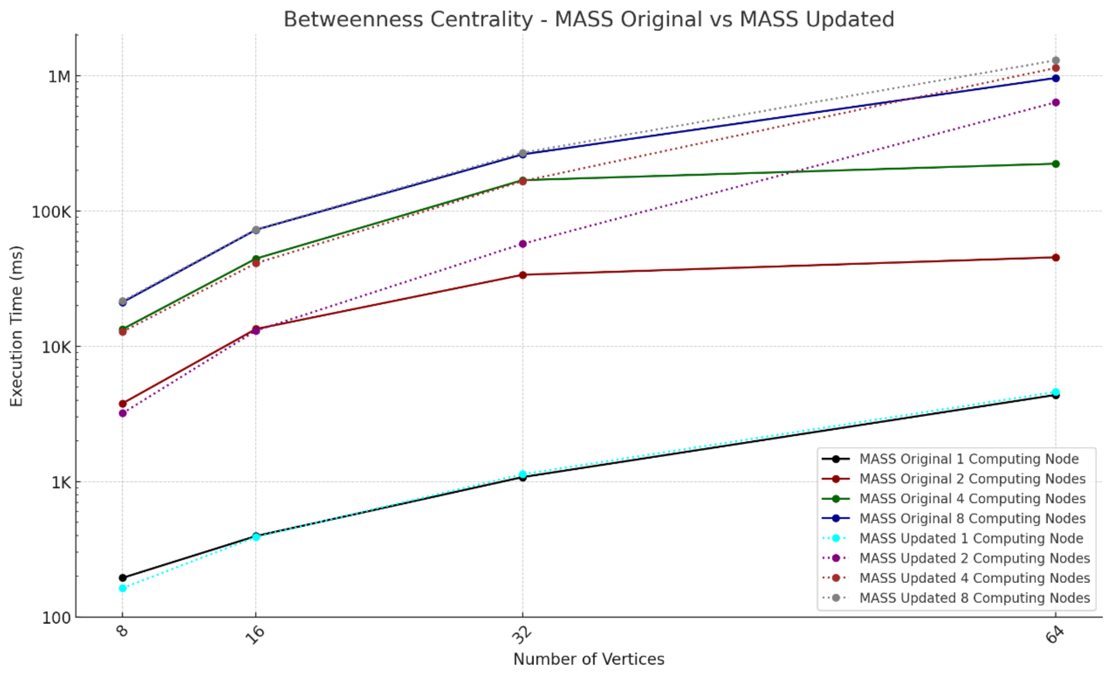


Figure 5.9 Betweenness Centrality MASS Updated vs MASS QL

In Figure 5.9, we can see the execution performance of the MASS Updated and MASS QL betweenness centrality applications over eight computing nodes. The single-node performance of the MASS Updated application remains dominant, while the multi-node executions are much more comparable in performance. Both applications do not scale well under multi-node executions and suffer from long execution times as the graph sizes scale up. The performance statistics for the MASS QL betweenness centrality application and the MASS Updated two and four-node executions at graph size 128 are omitted due to long run times.

The performance issues exhibited by the MASS QL betweenness centrality are similar to the issue outlined in section 5.4.3 with the closeness centrality application. In the betweenness centrality application, we have the added computational burden of not only training all vertex pairings in the graph but also enumerating all shortest paths for those pairings. This added burden drives a significant increase in execution time as the graph size increases.

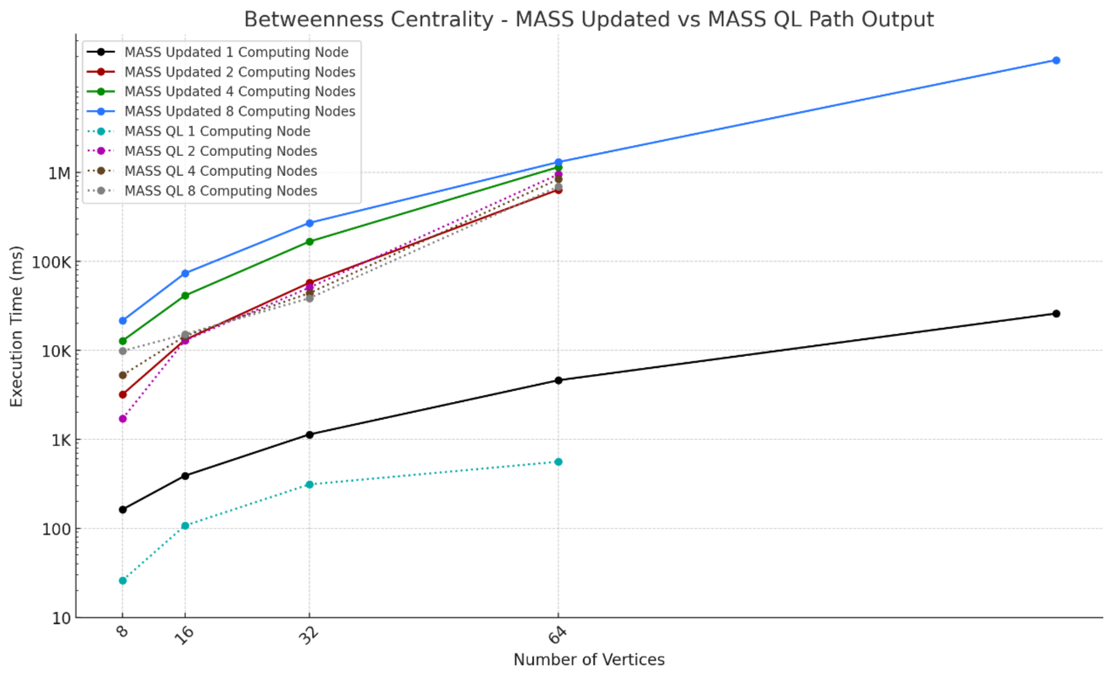


Figure 5.10 Betweenness Centrality MASS Updated vs MASS QL Path Output

In Figure 5.10, we can see the output performance of the MASS QL betweenness centrality application after training. With training complete, the MASS QL betweenness centrality application outperforms the current MASS Updated betweenness centrality application in producing the final betweenness centrality calculation. In this case, the single-node execution provides the best performance largely due to the agents not needing to traverse to remote computing nodes to produce the final paths.

5.5.2 *Dynamic Graphs*

Given the results presented in Section 5.4.2, dynamic testing was not conducted on betweenness centrality. The results from the closeness centrality dynamic testing would indicate that dynamic changes in a graph tested against the Q-learning betweenness centrality application would be significantly longer and fall far short of the MASS Updated betweenness centrality.

5.5.3 *Discussion*

The MASS QL betweenness centrality application faces similar issues to those of the closeness centrality application. In addition to the issues outlined in the closeness centrality application, the added computation burden of enumerating all shortest paths for each vertex pair adds to the execution time, making it ill-equipped to handle dynamic graph changes.

5.6 MASS Original & Updated Performance Comparison

5.6.1 *Shortest Path Execution Performance*

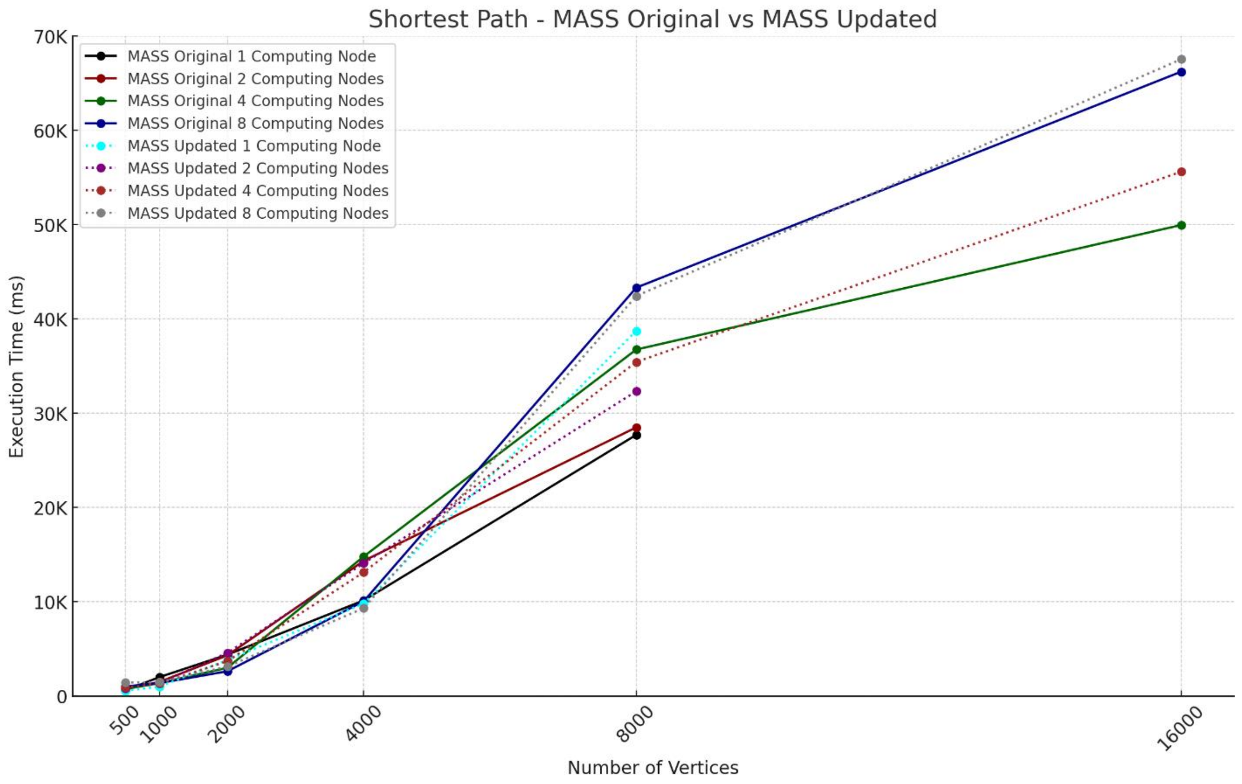


Figure 5.11 MASS Original vs MASS Updated Shortest Path Performance

Figure 5.11 outlines the execution performance of the MASS Original and MASS Updated shortest path applications. As we can see from the output, both applications track similarly in performance across all graph sizes. A notable element in this comparison is that as the graph size increases and more computing nodes are added to the cluster, the MASS Original application performs better compared to the MASS Updated. The performance difference is attributed to the added overhead associated with new features added to MASS, like GraphPlaces and GraphAgent, and integrated into the MASS Updated shortest path application. The new GraphPlaces vertex allocation method distributes the vertices in a round-robin fashion over the cluster system, leading to a larger overhead in graph navigation. While this data does indicate a performance decrease with these new features, the slight difference is negligible compared to the quality-of-life improvements they provide.

5.6.2 *Closeness Centrality Execution Performance*

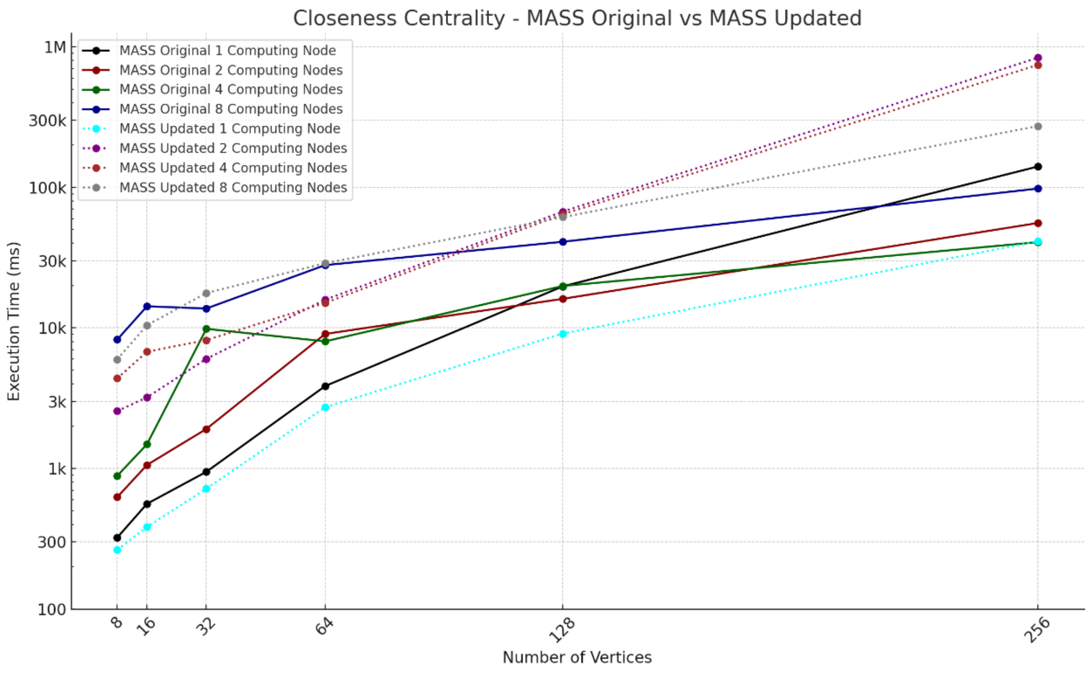


Figure 5.12 MASS Original vs MASS Updated Closeness Centrality

In Figure 5.12, we can see the performance comparison of the MASS Original and MASS Updated closeness centrality applications. The execution performance outlines an interesting trend when comparing the impact of the updated MASS features on previous applications. Up to graph size 128, the performance of both applications across one to eight computing nodes is fairly comparable. On the 256 vertex execution, the performance of the applications diverges drastically.

The MASS Updated application exhibits better performance on the single-node execution, while the two to eight-node executions reduce performance.

5.6.3 *Betweenness Centrality Execution Performance*

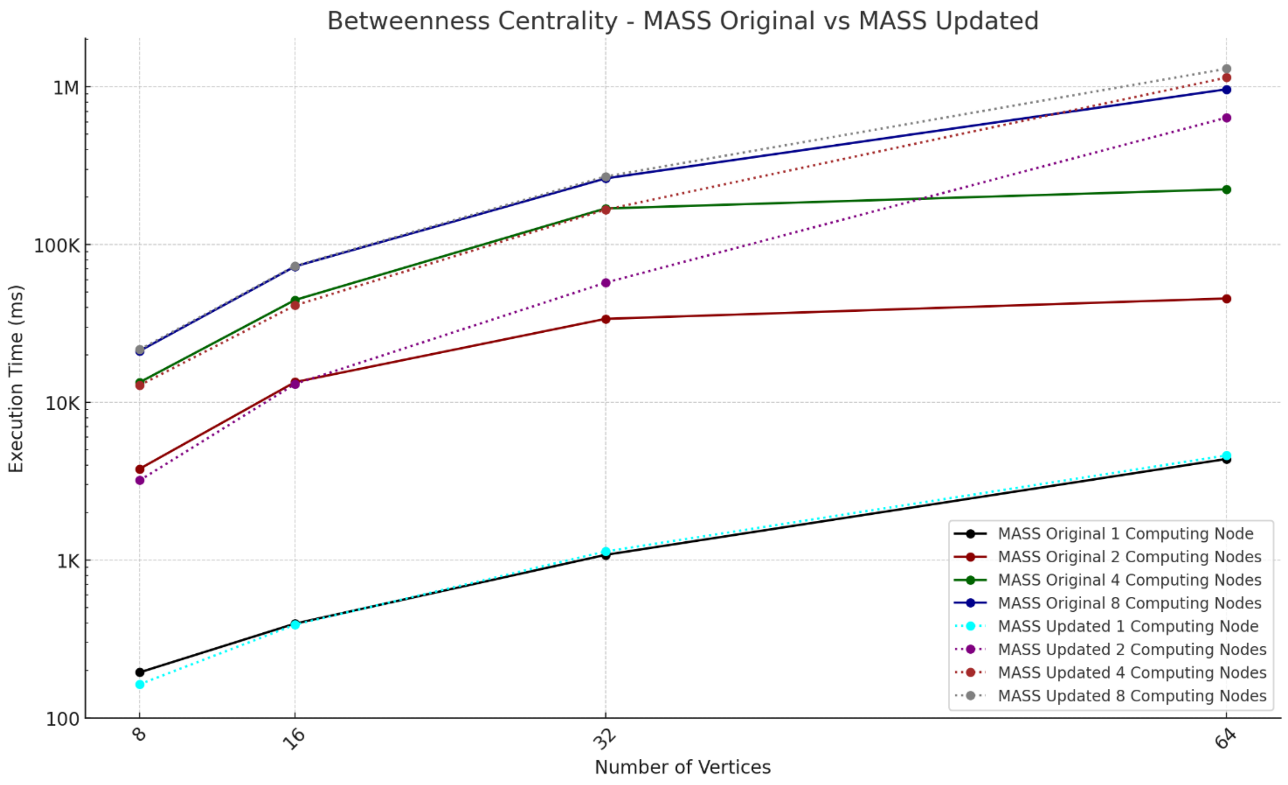


Figure 5.13 MASS Original vs MASS Updated Betweenness Centrality

Figure 5.13 compares the execution performance of the MASS Original and MASS Updated betweenness centrality applications. The performance exhibited by the different applications reinforces the findings in the closeness centrality applications from the previous sections. The performance of the two applications is largely consistent on the single computing node execution, with the MASS Updated implementation exhibiting better performance as the graph size reaches 128. On the multi-node executions, the MASS Original application scales far better than the MASS Updated application once the graph size reaches 32 vertices. The performance decrease behavior is likely associated with the overhead attached to the MASS Updated graph features, which do not scale as well to multi-node executions on small graph sizes.

5.6.4 *Discussion*

The overall performance of the MASS Original and MASS Updated applications indicates that the feature additions associated with the new version of MASS have little impact on optimized MASS applications. In scenarios like the shortest path problem where the MASS implementation is well suited to addressing the underlying problem, the impact of new features like GraphPlaces and GraphAgent do not significantly impact the program's overall performance. While there is a slight performance decrease associated with the added overhead of these new features, the impact is easily balanced with the improved user experience that comes with abstracting away some of the more tedious parts of programming in MASS. Making MASS more accessible to people without intimate knowledge of the library creates a more appealing tool for a broader audience and encourages the adoption of the library in more academic fields.

5.7 MASS Enabled Q-Learning Performance Improvements

5.7.1 *Multi-Agent Training*

In most Q-learning shortest path scenarios, like the ones referenced in Section 3.1, there is a single Q-learning agent that iteratively navigates the graph and populates the Q-table. One of the largest performance improvements presented in this project is the ability to utilize multiple agents in the training process. MASS enables the painless integration of multi-agent functionality, making it innately suitable for multi-agent training scenarios.

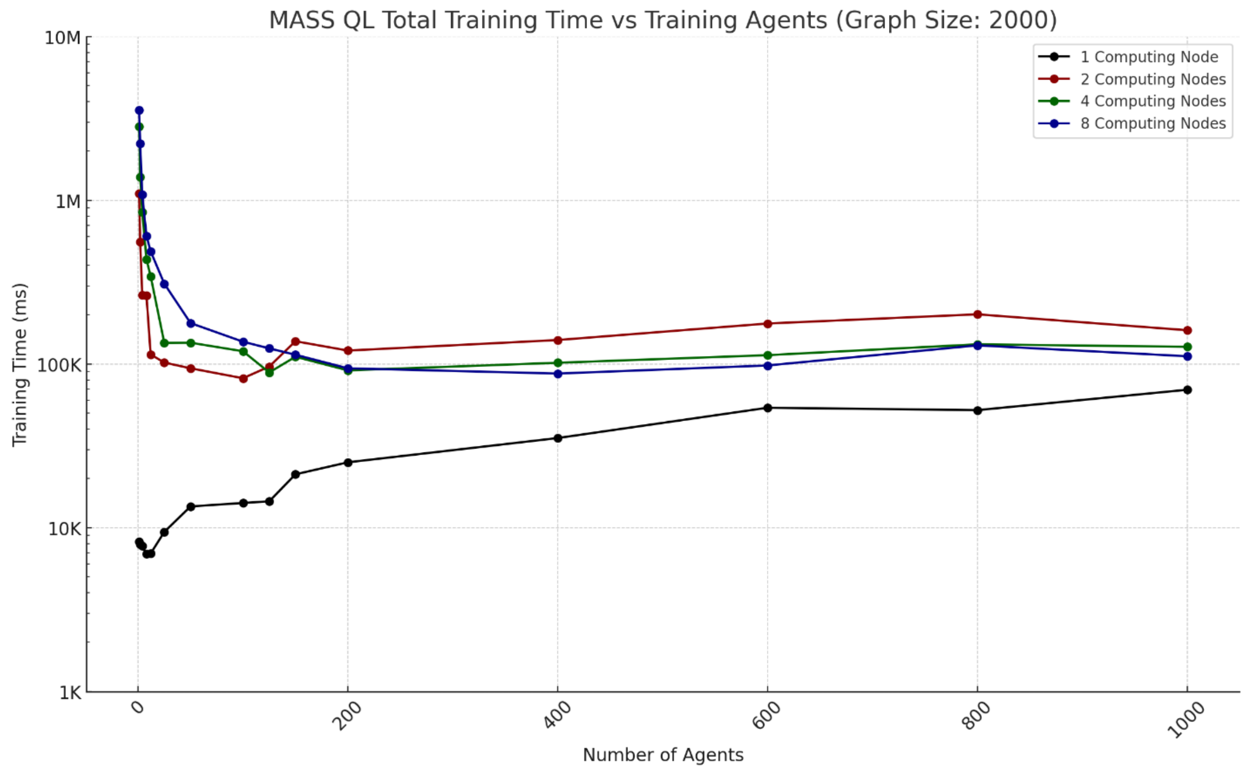


Figure 5.14 2000 Vertex MASS QL Multi-Agent Training Time

Figure 5.14 demonstrates the training performance gain across all eight nodes on a 2000 vertex graph with multi-agent training. While multi-agent training demonstrates performance increases across all computing node executions, the largest performance increase is found on eight-node executions, where we see a 190% training time decrease with 1750 training agents compared to a single training agent. A large reason for the increased performance on multi-node execution is the ability of the computing nodes to manage the agents in parallel, enabling agents to move and update the Q-table simultaneously. Multi-agent training enables the Q-learning in MASS to effectively reduce training time while exploring more of the graph structure and improving the overall performance of the algorithm.

5.7.2 *Dynamic Hyperparameter Tuning*

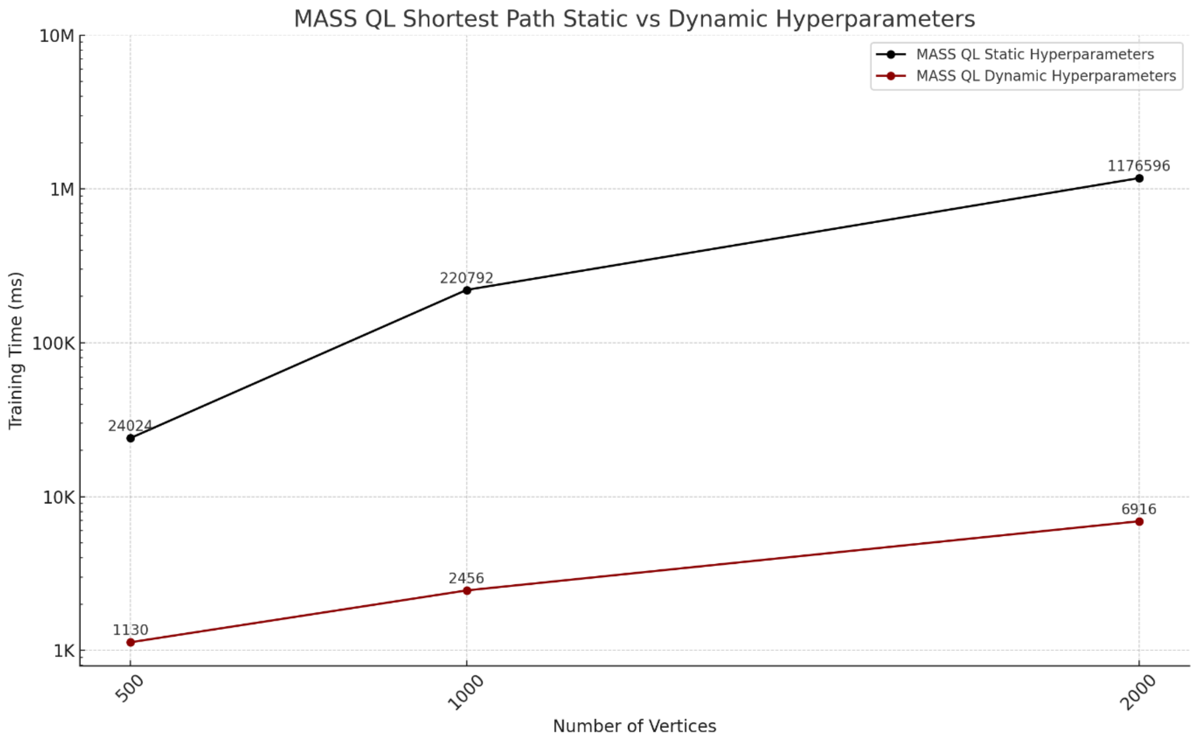


Figure 5.15 MASS QL Shortest Path Hyperparameter Tuning Performance

In many Q-learning implementations, hyperparameters like the learning rate and exploration rate are fixed throughout the training processes. While this does work, it tends to lead to longer training times than what is necessary to produce the optimal path. To improve the performance of our Q-learning implementation, we implemented a dynamic hyperparameter tuning process to reduce training times and improve the algorithm's overall performance. As shown in Figure 5.15, the dynamic hyperparameter tuning process allows our Q-learning implementation to modify the hyperparameters during the training process and reduce the overall training time.

Tuning the hyperparameters has two benefits. First, it enables us to train only when necessary, reducing the overall training time. Second, it enables the algorithm to better respond to dynamic changes in the graph. Through the reward window, the agent is able to understand better how the training process is going and can increase and decrease its hyperparameters in response to the rewards it receives while navigating the graph.

5.7.3 *Distributed Reward Window*

In multi-agent Q-learning training scenarios, one of the large impediments to performance improvements is the need for communication between the agents to share their training information with other agents. This is typically done with costly synchronization methods that result in heavy training time penalties. Initially, we implemented the reward window within each agent. This resulted in each agent training independently, and while they did share information through the distributed Q-table, additional agents resulted in increased training time.

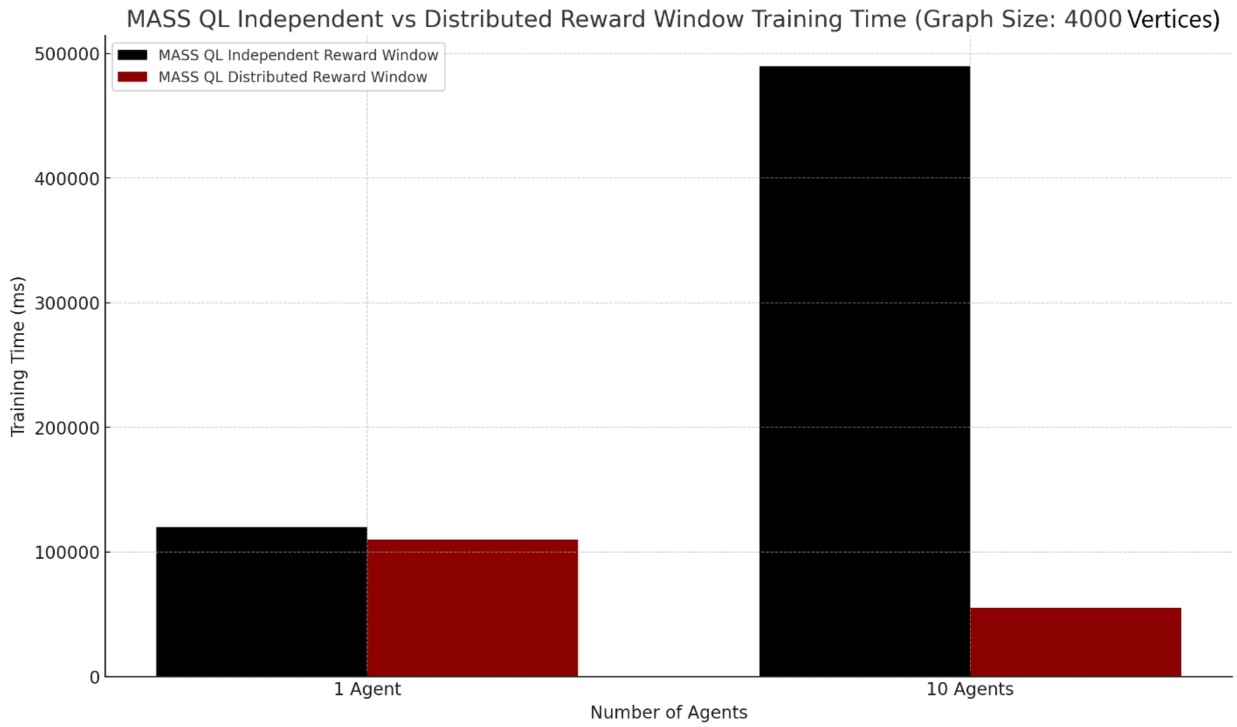


Figure 5.16 4000 Vertices MASS QL Reward Window Performance Comparison

Figure 5.16 demonstrates the performance difference between the independent reward window and the distributed reward window. The distributed reward window enables the agents to share their collective experiences over the course of each episode and tune the hyperparameters more quickly and efficiently. The MASS Node class enables a simple means for the Q-learning agent to share their cumulative episodic reward progress and provides all training agents with a much more holistic view of the aggregate training process. The distributed reward window underpins the performance gains achieved by multi-agent training.

Chapter 6. CONCLUSION

This project was successful in demonstrating the ability of MASS to accelerate reinforcement learning algorithms like Q-learning through its multi-agent programming paradigm. The ability of MASS to facilitate distributed graph computing and seamlessly employ agents to learn from the information presents a powerful tool for future work on multi-agent machine learning projects. The trained performance of the Q-learning applications demonstrated great performance and presented adaptive qualities when faced with dynamic data. While the results were positive overall, the research indicated there is still room for improvement to better attune MASS for machine learning, specifically reinforcement learning, based applications.

6.1 CHALLENGES

1. Generalized Q-learning Applications

One of the major caveats to the execution data presented is the highly tuned nature of the parameters used to run the Q-learning application. To optimize the applications for the input graphs mentioned in Sections 5.2.1 and 5.2.2, a comprehensive hyperparameter grid search was performed to find the combination of parameters that would yield the correct results while minimizing the overall execution time. While parameters proved effective for the graphs utilized in this study, they may not bore the same results for graphs that are structurally different. With that limitation noted, the parameters can be tuned in such a way that Q-learning can be applied to most, if not all, graph types, but this may lead to longer training time as parameters such as the reward window size and number of training agents may have to be set high to ensure generalization. Unoptimized hyperparameters can lead to longer training times, as demonstrated in Section 5.7.2.

2. Training Time

As mentioned in the previous section, training time comprised the most significant component of the overall execution time of the MASS Q-learning applications. While training is explicitly noted in the results for each of the applications, we also compare the output time after the algorithm has completed training. This is an important component to note when overall execution time is the primary area of concern. Except for some examples in small graph sizes, it is incredibly difficult for applications that require extensive training to produce execution times comparable to sequential or parallel solutions. While some machine learning implementations have been able to achieve improved results through methods like GPU-based acceleration, MASS Java lacks those methods in its current form.

6.2 Future Work

1. **Diverse Machine Learning Subfields**: While Reinforcement Learning algorithms are powerful tools for modeling unknown environments and deriving optimal policies, the need for repeated navigation presented a hurdle when comparing overall execution time with the current MASS implementations. Future works on machine learning in MASS should focus on strategies that minimize the need for repeated graph navigation operations. Techniques like Graph Convolutional Networks, graph embedding methods like Node2Vec, and probabilistic graphical models present an interesting area of study through an agent-based paradigm and may result in overall better performance.

2. **Agent Overhead:** As demonstrated in the training results, the most computationally costly component of all the Q-learning implementations is the repeated movement of training agents over the distributed graph. Future research into reducing the overhead associated with moving agents locally and across computing nodes could make agent-based MASS solutions more appealing for reinforcement learning algorithms and other machine learning implementations.
3. **Improved Agent Communication:** In this project, agents were able to communicate through the distributed Q-table and reward window. While this approach worked, it required agents to deposit data on a Node object to be read by other agents. This approach was effective but rather crude, as the agents could only receive the update on their next navigation cycle. Research into improved means for efficient agent communication methods would allow a more diverse pool of applications outside of the options explored in this paper.
4. **Dynamic Graph Modification Methods:** This project explored only one type of graph modification in the form of vertex removals. Other modifications, such as edge weight changes and vertex additions, would be worthwhile to explore and measure the adaptability of Q-learning. In addition, this project only explored unknown changes to the graph. Q-learning retraining could be optimized much more effectively by retraining only where changes occur, leading to much better retraining performance than was produced in this project.
5. **Online Graph Modifications:** While Q-learning demonstrated promise in terms of its ability to adapt to changes that occur after training is complete, its ability to adapt to

changes is not exclusive to offline changes. Further research into changes that occur while the algorithm is still in its training phase could demonstrate even more adaptability.

BIBLIOGRAPHY

[1] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “GraphX: Unifying Data-Parallel and Graph-Parallel Analytics,” Feb. 11, 2014, *arXiv*: arXiv:1402.2394. Accessed: Jul. 19, 2024. [Online]. Available: http://arxiv.org/abs/1402.2394

[2] 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*, Indianapolis Indiana USA: ACM, Jun. 2010, pp. 135–146. doi: 10.1145/1807167.1807184.

[3] M. Fukuda, *MASS: A Parallelizing Library for Multi-Agent Spatial Simulation*. [Online]. Available: https://depts.washington.edu/dslab/MASS/

[4] M. Kipps, W. Kim, and M. Fukuda, “Agent and Spatial Based Parallelization of Biological Network Motif Search,” in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, New York, NY: IEEE, Aug. 2015, pp. 786–791. doi: 10.1109/HPCC-CSS-ICESS.2015.222.

[5] Zhiyuan Ma and M. Fukuda, “A multi-agent spatial simulation library for parallelizing transport simulations,” in *2015 Winter Simulation Conference (WSC)*, Huntington Beach, CA, USA: IEEE, Dec. 2015, pp. 115–126. doi: 10.1109/WSC.2015.7408157.

[6] J. Woodring, M. Sell, M. Fukuda, H. Asuncion, and E. Salathe, “A Multi-agent Parallel Approach to Analyzing Large Climate Data Sets,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, GA, USA: IEEE, Jun. 2017, pp. 1639–1648. doi: 10.1109/ICDCS.2017.106.

[7] J. Zhang and Y. Luo, “Degree Centrality, Betweenness Centrality, and Closeness Centrality in Social Network,” in *Proceedings of the 2017 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017)*, Bankog, Thailand: Atlantis Press, 2017. doi: 10.2991/msam-17.2017.68.

[8] B. Mullen, C. Johnson, and E. Salas, “Effects of communication network structure: Components of positional centrality,” *Social Networks*, vol. 13, no. 2, pp. 169–185, Jun. 1991, doi: 10.1016/0378-8733(91)90019-P.

[9] J. Golbeck, “Network Structure and Measures,” in *Analyzing the Social Web*, Elsevier, 2013, pp. 25–44. doi: 10.1016/B978-0-12-405531-5.00003-1.

[10] A. Soofi *et al.*, “Centrality analysis of protein-protein interaction networks and molecular docking prioritize potential drug-targets in type 1 diabetes,” *IJPR*, vol. 19, no. 4, Nov. 2020, doi: 10.22037/ijpr.2020.113342.14242.

[11] K. Wang, W. Quan, N. Cheng, M. Liu, Y. Liu, and H. A. Chan, “Betweenness Centrality Based Software Defined Routing: Observation from Practical Internet Datasets,” *ACM Trans. Internet Technol.*, vol. 19, no. 4, pp. 1–19, Nov. 2019, doi: 10.1145/3355605.

[12] T. Phan and P. Do, “Improving the shortest path finding algorithm in apache spark graphX,” in *Proceedings of the 2nd International Conference on Machine Learning and Soft Computing*, Phu Quoc Island Viet Nam: ACM, Feb. 2018, pp. 67–71. doi: 10.1145/3184066.3184083.

[13] M. Sell and M. Fukuda, “Agent Programmability Enhancement for Rambling over a Scientific Dataset,” in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection*, vol. 12092, Y. Demazeau, T. Holvoet, J. M. Corchado, and S. Costantini, Eds., in Lecture Notes in Computer Science, vol. 12092. , Cham: Springer International Publishing, 2020, pp. 251–263. doi: 10.1007/978-3-030-49778-1_20.

[14] J. Emau, T. Chuang, and M. Fukuda, “A multi-process library for multi-agent and spatial simulation,” in *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada: IEEE, Aug. 2011, pp. 369–375. doi: 10.1109/PACRIM.2011.6032921.

[15] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, “Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory,” in *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA: IEEE, Dec. 2020, pp. 2957–2966. doi: 10.1109/BigData50022.2020.9378298.

[16] A. Lamba, “An introduction to Q-Learning: reinforcement learning.” Accessed: Dec. 01, 2023. [Online]. Available: https://medium.com/free-code-camp/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc

[17] X. Wang, L. Jin, and H. Wei, “The Shortest Path Planning Based on Reinforcement Learning,” *J. Phys.: Conf. Ser.*, vol. 1584, no. 1, p. 012006, Jul. 2020, doi: 10.1088/1742-6596/1584/1/012006.

[18] Z. Sun, “Applying Reinforcement Learning for Shortest Path Problem,” in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, Sanya, China: IEEE, Jan. 2022, pp. 514–518. doi: 10.1109/BDICN55575.2022.00100.

[19] R. Nannapaneni, “Optimal Path Routing Using Reinforcement Learning,” *Dell Technologies Proven Professional Knowledge Sharing*, 2020, Accessed: May 03, 2024. [Online]. Available: https://education.dell.com/content/dam/dell-emc/documents/en-us/2020KS_Nannapaneni-Optimal_path_routing_using_Reinforcement_Learning.pdf

[20] 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*, Dublin Ireland: ACM, Oct. 2014, pp. 37–50. doi: 10.1145/2660460.2660465.

[21] 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*, Barcelona Spain: ACM, Jun. 2020, pp. 1–12. doi: 10.1145/3392717.3392743.

[22] D. Ferone, P. Festa, A. Napoletano, and T. Pastore, “SHORTEST PATHS ON DYNAMIC GRAPHS: A SURVEY,” *Pesqui. Oper.*, vol. 37, no. 3, pp. 487–508, Sep. 2017, doi: 10.1590/0101-7438.2017.037.03.0487.

[23] E. P. F. Chan and Yaya Yang, “Shortest Path Tree Computation in Dynamic Graphs,” *IEEE Trans. Comput.*, vol. 58, no. 4, pp. 541–557, Apr. 2009, doi: 10.1109/TC.2008.198.

[24] A. Stentz, “Optimal and efficient path planning for partially-known environments,” in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, San Diego, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 3310–3317. doi: 10.1109/ROBOT.1994.351061.

[25] T. Akiba, Y. Iwata, and Y. Yoshida, “Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling,” in *Proceedings of the 23rd international conference on World wide web*, Seoul Korea: ACM, Apr. 2014, pp. 237–248. doi: 10.1145/2566486.2568007.

[26] H. Cui, R. Liu, S. Xu, and C. Zhou, “DMGA: A Distributed Shortest Path Algorithm for Multistage Graph,” *Scientific Programming*, vol. 2021, pp. 1–14, May 2021, doi: 10.1155/2021/6639008.

[27] G. Boeing, “OSMnx: A Python package to work with graph-theoretic OpenStreetMap street networks,” *JOSS*, vol. 2, no. 12, p. 215, Apr. 2017, doi: 10.21105/joss.00215.

[28] M. Haklay and P. Weber, “OpenStreetMap: User-Generated Street Maps,” *IEEE Pervasive Comput.*, vol. 7, no. 4, pp. 12–18, Oct. 2008, doi: 10.1109/MPRV.2008.80.

[29] G. Storchi, P. Dell’Olmo, and M. Gentili, “Rome99 Dataset: Directed Road Network of Rome, Italy.” University of Rome “La Sapienza,” Mar. 2006. [Online]. Available: http://www.diag.uniroma1.it/challenge9/benchmarks/rome99

APPENDIX

APPENDIX A: Graph Generation

A.1 Overview

This section provides a detailed description of the random graph generation process implemented in this project. Random graph generation is handled by two main classes: ‘GraphGen’ and ‘Map’. These classes are responsible for creating nodes, assigning neighbors, and defining distances between nodes to form a synthetic graph used for testing the shortest path and centrality applications.

A.2 GraphGen Class

The ‘GraphGen’ class initializes the graph generation process by creating instances of the ‘Map class for each node. Below is the code for the ‘GraphGen’ class:

```
public class GraphGen {
    public static void main(String[] args) {
        GraphGen g = new GraphGen(Integer.parseInt(args[0]));
    }

    public GraphGen(int nNodes) {
        System.out.println("nNodes = " + nNodes + "\n");

        Map[] node = new Map[nNodes];
        for (int i = 0; i < nNodes; i++) {
            System.out.println("Node " + i + ":");
            node[i] = new Map(nNodes, i);

            for (int j = 0; j < node[i].neighbors.length; j++) {
                System.out.println("to " + node[i].neighbors[j] + " with
distance " + node[i].distances[j]);
            }
        }
    }
}
```

A.3 Map Class

The ‘Map’ class is responsible for generating neighbors and distances for each node. It uses the Random class to assign a random number of neighbors and random distances, ensuring that there are no self-loops or duplicate edges. Below is the code for the ‘Map’ class:

```
public class Map {
    public int[] neighbors = null;
    public int[] distances = null;

    public Map(int nNodes, int nodeId) {
        // Prepare temporal storages for neighboring information
        ArrayList<Integer> neighborsList = new ArrayList<Integer>();
        ArrayList<Integer> distancesList = new ArrayList<Integer>();

        // All nodes have the same random number generator
        Random rand = new Random(0);
        // All nodes have the same upper limit
        int upperLimit = (int) (nNodes * 0.3);

        for (int i = 0; i < nNodes; i++) {
            // Generate # neighbors from node i
            int nNeighbors = rand.nextInt(upperLimit);
            if (nNeighbors == 0) nNeighbors = 1;

            // Generate neighboring nodes from node i
            for (int j = 0; j < nNeighbors; j++) {
                int neighbor = rand.nextInt(nNodes);
                int distance = rand.nextInt(nNodes);
                if (distance == 0) distance = 1;
                if (i == nodeId) {
                    // check if this is a self-directed link
                    if (neighbor == nodeId) continue;
                    // check if this is a duplication
                    if (neighborsList.indexOf(new Integer(neighbor)) == -1) {
                        // store my neighbor and its distance
                        neighborsList.add(new Integer(neighbor));
                        distancesList.add(new Integer(distance));
                    }
                } else if (neighbor == nodeId) {
                    // check if this is a duplication
                    if (neighborsList.indexOf(new Integer(i)) == -1) {
                        // store my neighbor and its distance
                        neighborsList.add(new Integer(i));
                        distancesList.add(new Integer(distance));
                    }
                }
            }
        }

        neighbors = new int[neighborsList.size()];
        distances = new int[distancesList.size()];
        for (int i = 0; i < neighborsList.size(); i++) {
            neighbors[i] = neighborsList.get(i);
            distances[i] = distancesList.get(i);
        }
    }
}
```

APPENDIX B: Road Network Graph Generation

B.1 Overview

This section provides a description of the process used to generate road network graphs using the ‘osmnx’ and ‘networkx’ libraries. The script downloads road network data for a specific city, remaps node IDs to a sequential formation, and outputs the graph in DSL format.

B.2 Road Network Graph Generation Script

The following Python script is used to generate road network graphs for a specified city. It downloads the road network details, uses the ‘osmnx’ library, remaps node IDs, ensures all nodes are bidirectional, and writes the graph to a DSL file. Below is the script:

```
import osmnx as ox
import networkx as nx

# Define city to create graph from
```

```
place_name = "Tacoma, Washington, USA"

# Download the road network graph
G = ox.graph_from_place(place_name, network_type='drive')

# Remap nodes
original_nodes = list(G.nodes())
node_mapping = {old_id: new_id for new_id, old_id in
enumerate(original_nodes)}

# Create a new graph with remapped node IDs
new_graph = nx.relabel_nodes(G, node_mapping)

# Ensure all nodes are included and set edge weights to 1, make graph non-
directional
graph_dict = {new_id: set() for new_id in range(len(original_nodes))}
for u, v, data in new_graph.edges(data=True):
    graph_dict[u].add((v, 1))  # Set the weight to 1
    graph_dict[v].add((u, 1))  # Ensure the reverse edge is also added

# Get the number of nodes and edges
num_nodes = len(new_graph.nodes())
num_edges = len(new_graph.edges())

# Create the filename based on the city prefix, number of nodes, and number
of edges
city_prefix = place_name.split(',')[0].replace(" ", "_")
filename =
f"{city_prefix}_road_network_{num_nodes}nodes_{num_edges}edges.dsl"

# Write to a file in DSL format
with open(filename, 'w') as f:
    for node, edges in sorted(graph_dict.items()):
        edges_str = ';'.join([f"{v},{weight}" for v, weight in edges])
        f.write(f"{node}={edges_str}\n")
```

APPENDIX C: MASS Applications Execution Instructions

C.1 Overview

The MASS Updated and MASS Q-learning applications are located in the MASS Application Developers Bitbucket at: https://bitbucket.org/mass_application_developers/mass_java_appl. To run the applications, clone the branch “Jeff/Qlearning” using the command: git clone --branch Jeff/Qlearning https://bitbucket.org/mass_application_developers/mass_java_app.git.

This command will clone the Q-learning repository into two folders: ‘MASS_Updated’ and ‘MASS_Q-learning.’

C.2 MASS Environment

The MASS Updated and MASS Q-Learning applications utilize MASS Java Core version 1.4.3. All versions of MASS Java are available at: https://bitbucket.org/mass_library_developers/mass_java_core. To install MASS Java core version 1.4.3, navigate to the desired folder and execute the command “git clone --branch 1.4.3-SNAPSHOT https://bitbucket.org/mass_library_developers/mass_java_core.git”. This command will clone the 1.4.3 branch to your local NETID account. After the download is complete, navigate into the ‘mass_java_core’ folder and execute the command “mvn -DskipTests clean package install”. This will install the MASS Java Core version 1.4.3 in your local environment.

For the dynamic graph testing with a pre-trained Q-table and DSLQ file format support, use “git clone --branch Jeff/DSLQ_1.4.3 https://bitbucket.org/mass_library_developers/mass_java_core.git”. This command will clone the modified 1.4.3 branch to your local NETID account. After the download is complete, navigate into the “mass_java_core” folder and execute the command “mvn -DskipTests clean package install” and wait for maven to package and install MASS. This will install the modified MASS Java Core version 1.4.3 in your local environment.

C.3 Graph Files

The graph files used in this project are available here: https://drive.google.com/drive/folders/15NVPUWw0CJ7FH-VqmgIvpRTI6ZkZOn9e?usp=sharing. The graph files are divided into two folders: ‘Graphs’ and ‘Graphs with Qtables'. The ‘Graphs’ directory contains the three datasets used in this project: Synthetic, Road Network, and Centrality, in DSL format. The ‘Graphs with Qtables’ directory contains the synthetic and road network graph files, which include the pre-trained Q-values for each vertex-to-vertex transition. The is an updated file formatted dubbed “DSLQ” and is used for the dynamic shortest path testing.

C.4 MASS Updated Applications

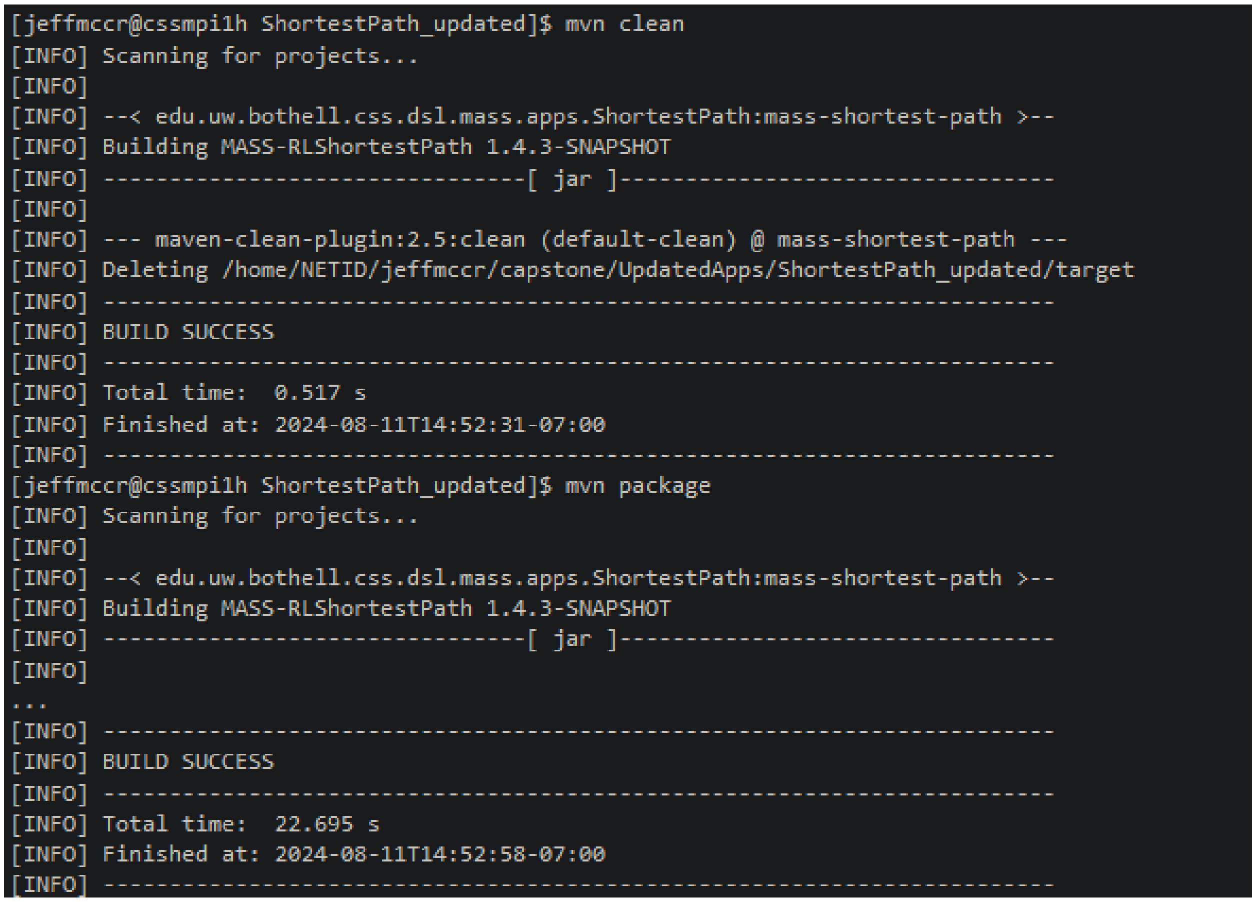
The ‘MASS Updated’ folder includes three subdirectories: ‘MASS_Shortestpath_Updated’, ‘MASS_Closeness_Updated’, and ‘MASS_Betweenness_Updated’. Each folder represents the previous MASS applications updated to utilize MASS Core version 1.4.3 with GraphPlaces and GraphAgents, including a pom.xml file with the required dependencies. Ensure all dependencies are the most recent versions.

With the appropriate MASS Core version installed per Section C.2, each application will need to be compiled.

To compile each application:

1. Execute ‘mvn clean’ to clean previous compilations.
2. Execute ‘mvn package’ to compile the program into an executable jar file.

If successful, a BUILD SUCCESS message will appear, and the target folder will contain the jar file, such as mass-shortest-path-1.4.3-SNAPSHOT.jar. Move the jar file into the top-level application directory before executing the program.

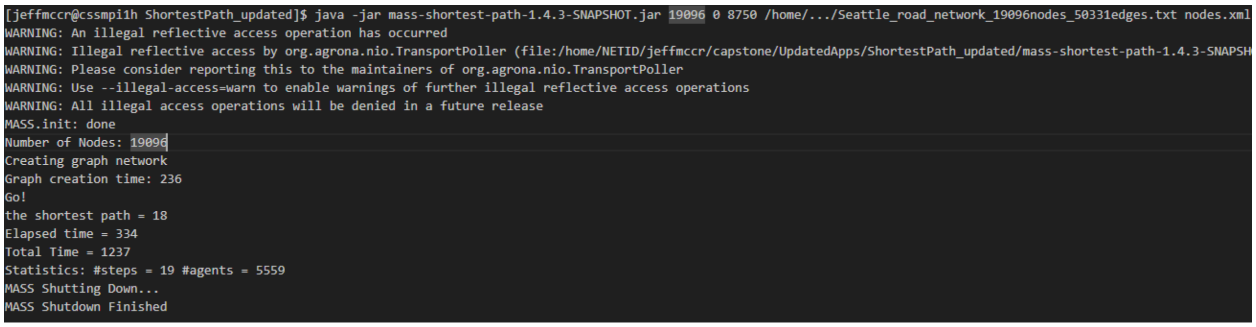


```
[jeffmccr@cssmpi1h ShortestPath_updated]$ mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] --< edu.uw.bothell.css.dsl.mass.apps.ShortestPath:mass-shortest-path >--
[INFO] Building MASS-RLShortestPath 1.4.3-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ mass-shortest-path ---
[INFO] Deleting /home/NETID/jeffmccr/capstone/UpdatedApps/ShortestPath_updated/target
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.517 s
[INFO] Finished at: 2024-08-11T14:52:31-07:00
[INFO] ------------------------------------------------------------------------
[jeffmccr@cssmpi1h ShortestPath_updated]$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] --< edu.uw.bothell.css.dsl.mass.apps.ShortestPath:mass-shortest-path >--
[INFO] Building MASS-RLShortestPath 1.4.3-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  22.695 s
[INFO] Finished at: 2024-08-11T14:52:58-07:00
[INFO] ------------------------------------------------------------------------
```

Figure C.1 MASS Program Compilation

C.4.1 Shortest Path

The MASS Updated shortest path application takes in five arguments: the number of graph vertices, source node, destination node, graph file path, and computing node file path. It is important to note that for multi-computing node executions, the graph file path must be the full file path, even if the graph file is stored in the current working directory. Failure to use the full file path will result in MASS execution issues. The application is executed using the following command: “java -jar ‘jarfile.jar’ ‘number of graph vertices’ ‘source node’ ‘destination node’ ‘graph file path’ ‘node file path’”.



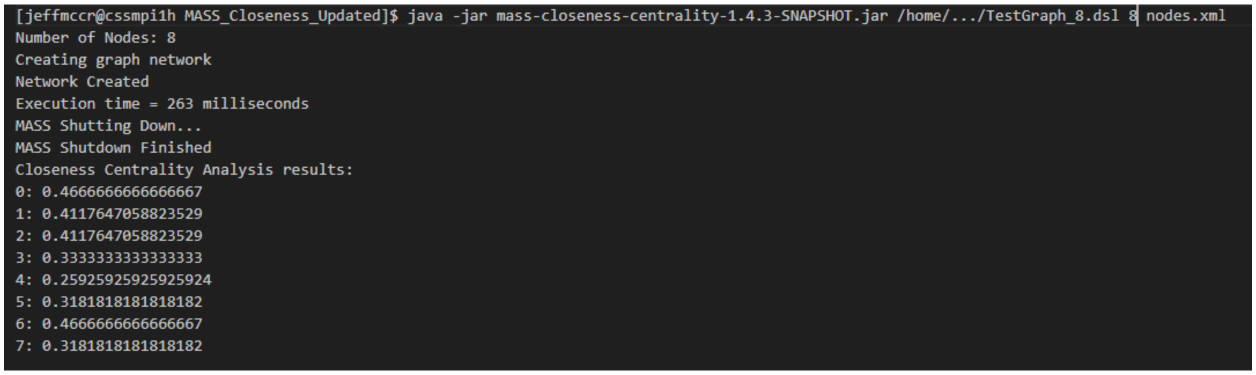
```
[jeffmccr@cssmpi1h ShortestPath_updated]$ java -jar mass-shortest-path-1.4.3-SNAPSHOT.jar 19096 0 8750 /home/.../Seattle_road_network_19096nodes_50331edges.txt nodes.xml
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/NETID/jeffmccr/capstone/UpdatedApps/ShortestPath_updated/mass-shortest-path-1.4.3-SNAPSH
WARNING: Please consider reporting this to the maintainers of org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
MASS.init: done
Number of Nodes: 19096
Creating graph network
Graph creation time: 236
Go!
the shortest path = 18
Elapsed time = 334
Total Time = 1237
Statistics: #steps = 19 #agents = 5559
MASS Shutting Down...
MASS Shutdown Finished
```

Figure C.2 MASS Updated Shortest Path Execution

If the program executes correctly, the MASS Updated shortest path application will output the shortest path length and runtime statistics. Please note that MASS debug messages are disabled by default. If debug messages are desired, the ShortestPath.java file must be modified to enable them, and the program must be recompiled.

C.4.2 Closeness Centrality

The MASS Updated closeness centrality application takes in three arguments: the graph file path, the number of vertices, and the node file path. As with the shortest path application, multi-computing node executions require the full file path for the graph file. The application is executed using the following command: "java -jar 'jarfile.jar' 'graph file path' 'number of vertices' 'node file path'".



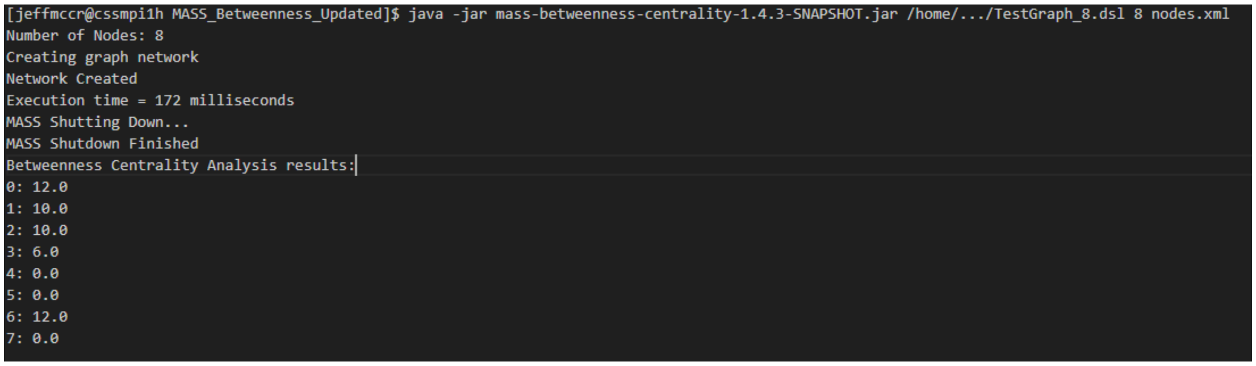
```
[jeffmccr@cssmpi1h MASS_Closeness_Updated]$ java -jar mass-closeness-centrality-1.4.3-SNAPSHOT.jar /home/.../TestGraph_8.dsl 8 nodes.xml
Number of Nodes: 8
Creating graph network
Network Created
Execution time = 263 milliseconds
MASS Shutting Down...
MASS Shutdown Finished
Closeness Centrality Analysis results:
0: 0.4666666666666667
1: 0.4117647058823529
2: 0.4117647058823529
3: 0.3333333333333333
4: 0.25925925925925924
5: 0.3181818181818182
6: 0.4666666666666667
7: 0.3181818181818182
```

Figure C.3 MASS Updated Closeness Centrality Execution

If the program executes correctly, the MASS Updated closeness centrality application will output the closeness centrality for each node in the graph and runtime statistics. Please note that MASS debug messages are disabled by default. If debug messages are desired, the Driver.java file will have to be modified to enable them, and the program will need to be recompiled.

C.4.3 Betweenness Centrality

The MASS Updated betweenness centrality application takes in three arguments: the graph file path, the number of vertices, and the node file path. As with the shortest path application, multi-computing node executions require the full file path for the graph file. The application is executed using the following command: "java -jar 'jarfile.jar' 'graph file path' 'number of vertices' 'node file path'".



```
[jeffmccr@cssmpi1h MASS_Betweenness_Updated]$ java -jar mass-betweenness-centrality-1.4.3-SNAPSHOT.jar /home/.../TestGraph_8.dsl 8 nodes.xml
Number of Nodes: 8
Creating graph network
Network Created
Execution time = 172 milliseconds
MASS Shutting Down...
MASS Shutdown Finished
Betweenness Centrality Analysis results:
0: 12.0
1: 10.0
2: 10.0
3: 6.0
4: 0.0
5: 0.0
6: 12.0
7: 0.0
```

Figure C.4 MASS Updated Betweenness Centrality Execution

If the program executes correctly, the MASS Updated betweenness centrality application will output the betweenness centrality for each node in the graph and runtime statistics. Please note that by default, MASS debug messages are disabled. If debug messages are desired, the Driver.java file will have to be modified to enable debug messages, and the program will need to be recompiled.

C.5 MASS Q-Learning Applications

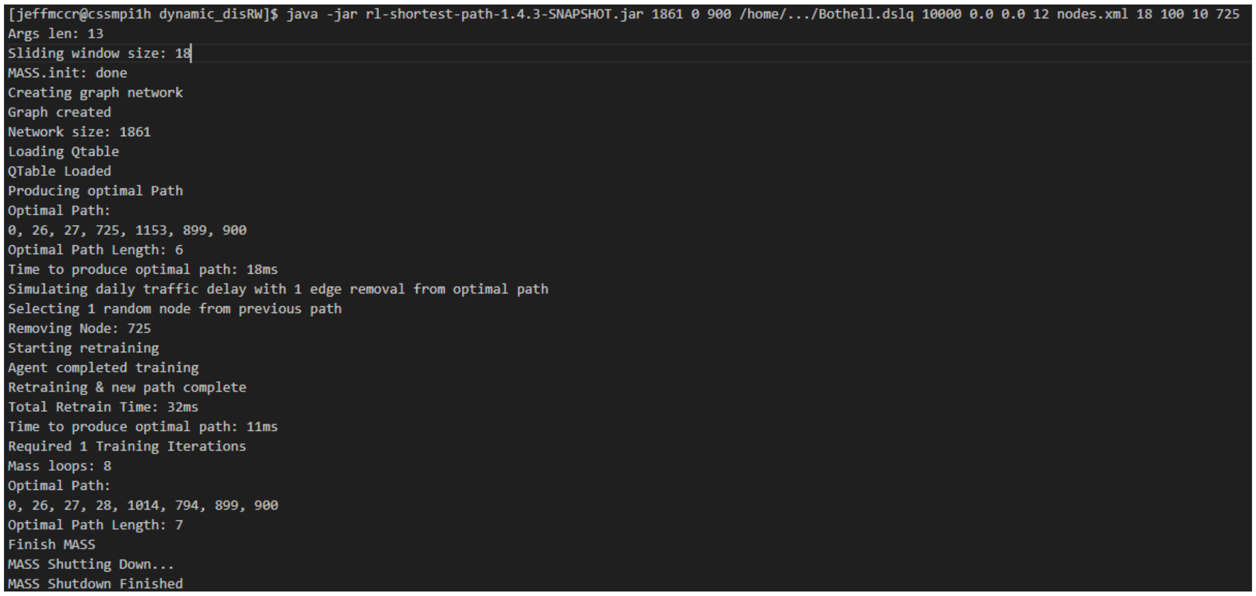
The MASS Q-learning applications are divided into three folders: ‘QLShortestPath’, ‘QLClosenessCentrality’, and ‘QLBetweennessCentrality’. Each folder represents the Q-learning-based MASS applications and will need to be packaged separately. The folders include a pom.xml file with the required dependencies. Ensure that all dependencies are the most recent versions.

If successful, a BUILD SUCCESS message will appear, and the target folder will contain the jar file, such as ql-shortest-path-1.4.3-SNAPSHOT.jar. Move the jar file into the top-level application directory before executing the program.

C.5.1 Shortest Path

The QLShortestPath application takes in 10 arguments: number of vertices, source vertex, destination vertex, graph file path, maximum training episodes, Q-value threshold, path threshold, number of training agents, sliding window size, and node file path. While most arguments are self-explanatory, the Q-value threshold and path threshold control the optimal and suboptimal path enumeration. If multi-path enumeration is not required, set both parameters to 0.0 to bypass enumeration. For the graph file path, ensure that the path is complete when executing on multiple computing nodes. The application is executed using the following command: “java -jar ‘jarfile.jar’ ‘number of vertices’ ‘source vertex’ ‘destination vertex’ ‘graph file path’ ‘max training episodes’ ‘Q-value threshold’ ‘path threshold’ ‘number of training agents’ ‘sliding window size’ ‘node file path’”.

For dynamic testing, the QLShortestPath application takes in additional arguments. First, the graph file, as previously mentioned, should be in DSLQ format, which includes a pre-trained Q-table with the graph table. The dynamic testing QLShortestPath takes in potentially 13 arguments: number of vertices, source vertex, destination vertex, graph file path, maximum training episodes, Q-value threshold, path threshold, number of training agents, sliding window size, node file path, number of retraining agents, number of retraining episodes, and optionally, a specific node or nodes to delete. If a vertex(s) is not specified, a random vertex from the previous optimal path will be selected and removed before retraining. The application is executed using the following command: “java -jar ‘jarfile.jar’ ‘number of vertices’ ‘source vertex’ ‘destination vertex’ ‘graph file path’ ‘max training episodes’ ‘Q-value threshold’ ‘path threshold’ ‘number of training agents’ ‘sliding window size’ ‘node file path’ ‘number of retraining agents’ ‘number of retraining episodes’ ‘vertex/vertices to delete…’”.



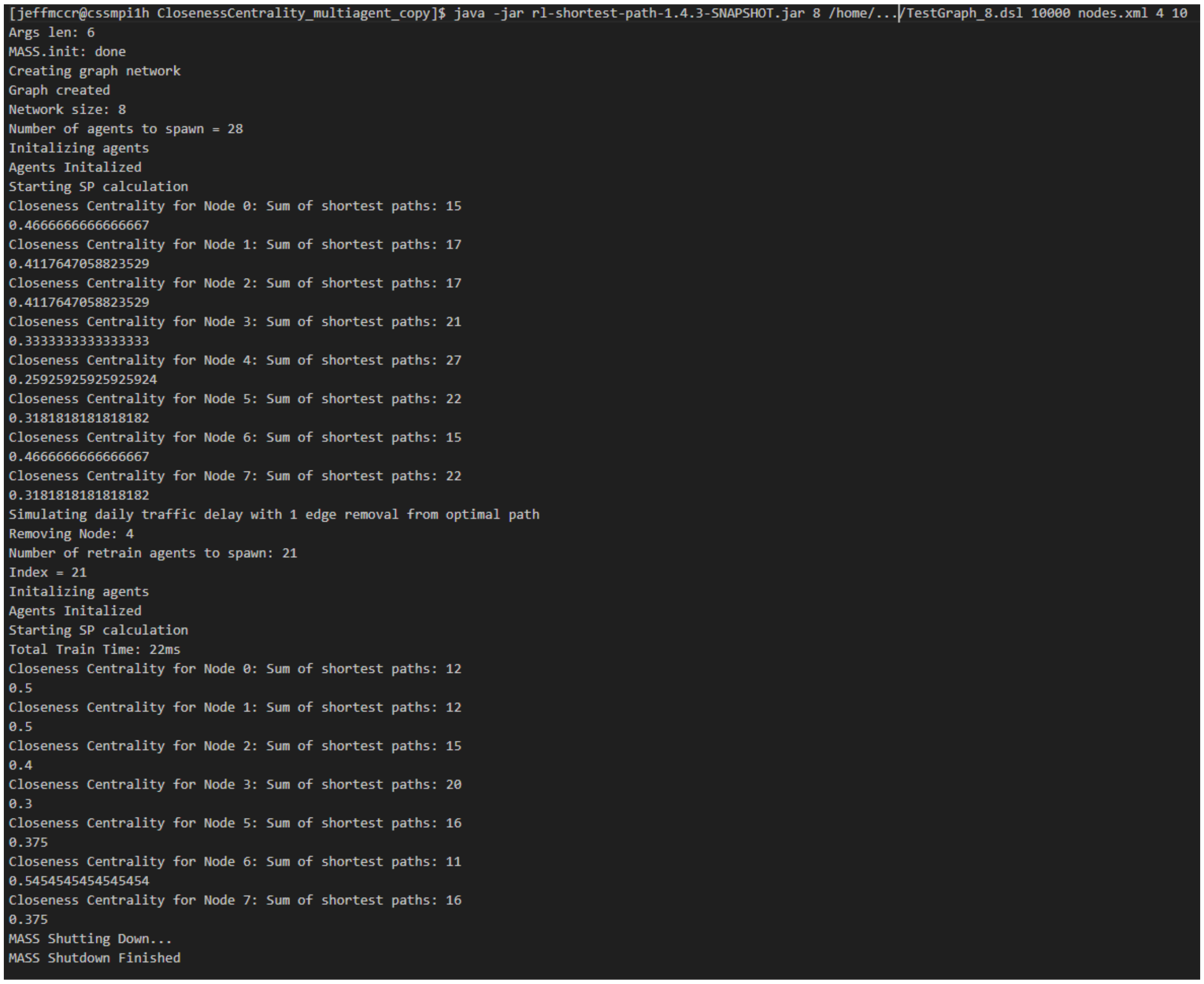
```
[jeffmccr@cssmpi1h dynamic_disRW]$ java -jar rl-shortest-path-1.4.3-SNAPSHOT.jar 1861 0 900 /home/.../Bothell.dslq 10000 0.0 0.0 12 nodes.xml 18 100 10 725
Args len: 13
Sliding window size: 18
MASS.init: done
Creating graph network
Graph created
Network size: 1861
Loading Qtable
QTable Loaded
Producing optimal Path
Optimal Path:
0, 26, 27, 725, 1153, 899, 900
Optimal Path Length: 6
Time to produce optimal path: 18ms
Simulating daily traffic delay with 1 edge removal from optimal path
Selecting 1 random node from previous path
Removing Node: 725
Starting retraining
Agent completed training
Retraining & new path complete
Total Retrain Time: 32ms
Time to produce optimal path: 11ms
Required 1 Training Iterations
Mass loops: 8
Optimal Path:
0, 26, 27, 28, 1014, 794, 899, 900
Optimal Path Length: 7
Finish MASS
MASS Shutting Down...
MASS Shutdown Finished
```

Figure C.5 MASS QL Shortest Path Execution

If the program executes correctly, the QLShortestPath application will output the shortest path and runtime statistics. Please note that MASS debug messages are disabled by default. If debug messages are desired, the ShortestPath.java file must be modified to enable them, and the program must be recompiled.

C.4.2 Closeness Centrality

The QLClosenessCentrality takes in up to six arguments: number of vertices, graph file path, maximum number of training episodes, sliding window size, node file path, and optionally, the number of retraining episodes and a vertex to remove. For dynamic testing. If the number of retraining episodes is not specified, dynamic testing will not be performed. If a vertex to remove is not specified, a random vertex will be selected. The application is executed using the following command: “java -jar ‘jarfile.jar’ ‘number of vertices’ ‘graph file path’ ‘maximum training episodes’ ‘sliding window size’ ‘node file path’ ‘retraining episodes’ ‘vertex to remove’”.



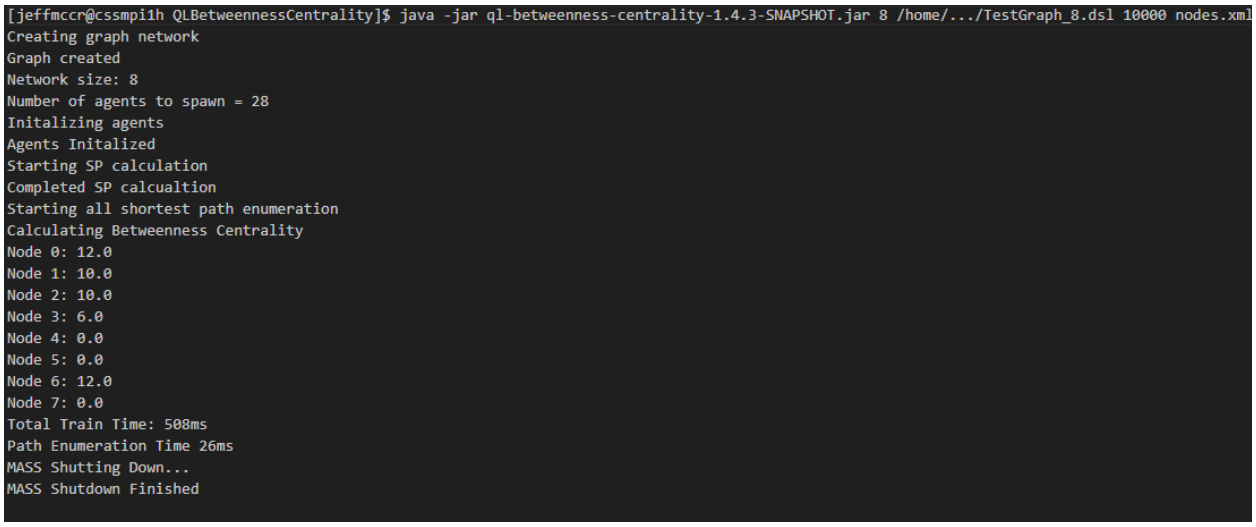
```
[jeffmccr@cssmpi1h ClosenessCentrality_multiagent_copy]$ java -jar rl-shortest-path-1.4.3-SNAPSHOT.jar 8 /home/.../TestGraph_8.dsl 10000 nodes.xml 4 10
Args len: 6
MASS.init: done
Creating graph network
Graph created
Network size: 8
Number of agents to spawn = 28
Initalizing agents
Agents Initalized
Starting SP calculation
Closeness Centrality for Node 0: Sum of shortest paths: 15
0.4666666666666667
Closeness Centrality for Node 1: Sum of shortest paths: 17
0.4117647058823529
Closeness Centrality for Node 2: Sum of shortest paths: 17
0.4117647058823529
Closeness Centrality for Node 3: Sum of shortest paths: 21
0.3333333333333333
Closeness Centrality for Node 4: Sum of shortest paths: 27
0.25925925925925924
Closeness Centrality for Node 5: Sum of shortest paths: 22
0.3181818181818182
Closeness Centrality for Node 6: Sum of shortest paths: 15
0.4666666666666667
Closeness Centrality for Node 7: Sum of shortest paths: 22
0.3181818181818182
Simulating daily traffic delay with 1 edge removal from optimal path
Removing Node: 4
Number of retrain agents to spawn: 21
Index = 21
Initalizing agents
Agents Initalized
Starting SP calculation
Total Train Time: 22ms
Closeness Centrality for Node 0: Sum of shortest paths: 12
0.5
Closeness Centrality for Node 1: Sum of shortest paths: 12
0.5
Closeness Centrality for Node 2: Sum of shortest paths: 15
0.4
Closeness Centrality for Node 3: Sum of shortest paths: 20
0.3
Closeness Centrality for Node 5: Sum of shortest paths: 16
0.375
Closeness Centrality for Node 6: Sum of shortest paths: 11
0.5454545454545454
Closeness Centrality for Node 7: Sum of shortest paths: 16
0.375
MASS Shutting Down...
MASS Shutdown Finished
```

Figure C.6 MASS QL Closeness Centrality Execution

If the program executes correctly, the QLClosenessCentrality application will output the closeness centrality for each node and runtime statistics. Please note that MASS debug messages are disabled by default. If debug messages are desired, the ClosenessCentrality.java file must be modified to enable them, and the program must be recompiled.

C.4.3 Betweenness Centrality

The QLBetweennessCentrality takes in four arguments: the number of vertices, the graph file path, the maximum number of training episodes, and the node file path. The application is executed using the following command: “java -jar ‘jarfile.jar’ ‘number of vertices’ ‘graph file path’ ‘maximum training episodes’ ‘node file path’”.



```
[jeffmccr@cssmpi1h QLBetweennessCentrality]$ java -jar ql-betweenness-centrality-1.4.3-SNAPSHOT.jar 8 /home/.../TestGraph_8.dsl 10000 nodes.xml
Creating graph network
Graph created
Network size: 8
Number of agents to spawn = 28
Initalizing agents
Agents Initalized
Starting SP calculation
Completed SP calcualtion
Starting all shortest path enumeration
Calculating Betweenness Centrality
Node 0: 12.0
Node 1: 10.0
Node 2: 10.0
Node 3: 6.0
Node 4: 0.0
Node 5: 0.0
Node 6: 12.0
Node 7: 0.0
Total Train Time: 508ms
Path Enumeration Time 26ms
MASS Shutting Down...
MASS Shutdown Finished
```

Figure C.7 MASS QL Betweenness Centrality Execution

If the program executes correctly, the QLBetweennessCentrality application will output the betweenness centrality for each node and runtime statistics. Please note that MASS debug messages are disabled by default. If debug messages are desired, the BetweennessCentrality.java file must be modified to enable them, and the program must be recompiled.