Accelerated Q-Learning for Optimal Pathfinding

Introduction

Fog computing is an emerging paradigm that extends cloud services to the network edge, enabling low-latency processing for IoT and mobile applications. In dynamic fog networks, routing or task assignment often requires finding optimal paths under changing conditions. Classical algorithms like Dijkstra or A* can compute shortest paths on a static graph, but they need full knowledge of the network and must be rerun when conditions change. Reinforcement learning (RL) offers a complementary approach: an agent can **learn** good routes or task assignments over repeated interactions with the fog network, adapting to delays, loads, or faults. For example, Siyadatzadeh *et al.* (2023) introduced **ReLIEF**, an RL-based strategy for fault-tolerant task assignment in fog computing, demonstrating that a well-trained RL agent can improve reliability under varying workloads.

In particular, **Q-learning** – a model-free RL algorithm – can iteratively learn an optimal action-value function (Q-table) for pathfinding in a graph. Q-learning maintains a table of expected rewards for state-action pairs, updating it via the Bellman equation as the agent explores the network. Over many episodes, the agent converges to an optimal (or near-optimal) policy, balancing exploration and exploitation (typically via an ε-greedy policy). However, tabular Q-learning can suffer from **slow convergence** when the state space (number of nodes) is large or rewards are sparse. This is especially true for pathfinding in moderately-sized graphs, where many episodes may be needed before the Q-values stabilize.

Parallelism can help mitigate this. Recent work has shown that **massively parallel** implementations of RL can drastically reduce wall-clock training time. For instance, Li *et al.* (2023) demonstrated a Parallel Q-Learning scheme on GPUs that outperforms traditional methods (e.g. PPO) by parallelizing data collection and Q-value updates across thousands of environments. Similarly, Kuchibhotla *et al.* (2020) explored a multi-threaded Q-learning implementation to accelerate learning by distributing the update workload. The key insight is that the Q-learning update step (which computes a new Q-value from a reward plus discounted max future value) can be performed in parallel for many state-action pairs. By leveraging **OpenMP** on multi-core CPUs or **CUDA** on GPUs, one can train Q-learning agents much faster. Dilith Jay's Q-learning tutorial also emphasizes this point: while Q-learning's ε-greedy exploration is simple to implement, its practical use is limited by convergence speed and data requirements.

In this study, we consider the problem of **optimal pathfinding** in a fog network, comparing traditional graph search (Dijkstra and A*) with reinforcement learning approaches (sequential Q-learning, OpenMP-parallel Q-learning, and CUDA-based Q-learning). Our motivation is twofold: first, to evaluate whether Q-learning can find routes of comparable quality to classical methods; and second, to quantify the performance gains of parallelizing the Q-learning algorithm. We implement a synthetic fog network (random connected weighted graph) and run each algorithm under identical conditions. By measuring the *convergence iteration*, *path cost*, and *execution time* of each method, we can assess the trade-offs between learning-based and algorithmic routing approaches. We use first-person plural ("we") to describe our collective work and findings throughout this report.

Objective

Our goal is to systematically compare different implementations of Q-learning for pathfinding and contrast them with classical algorithms. Specifically, we aim to: (1) implement and evaluate **sequential Q-learning**, **OpenMP-parallel Q-learning**, and **CUDA-accelerated Q-learning** on synthetic graph data; (2) measure each method's convergence behaviour, solution quality (path cost), and execution time; and (3) benchmark these against **Dijkstra's algorithm** and **A*** search, which provide baseline optimal solutions. This comparison clarifies how parallelism impacts learning efficiency and how learning-based routing compares to direct graph search.

Major Contributions

ReLIEF (Siyadatzadeh *et al.*, 2023): This work introduced **ReLIEF**, an RL-based real-time task assignment strategy in fog computing. It uses reinforcement learning to select primary and backup fog nodes for IoT tasks, improving reliability under failures. The key contribution is demonstrating that an RL agent can learn a task-offloading policy that outperforms static heuristics in a dynamic, fault-prone fog environment. Although ReLIEF focuses on scheduling rather than pathfinding per se, it underscores the value of RL in fog networks and provides inspiration for using Q-learning in related problems.

Huang *et al.* (2022) – Massive Parallel Q-Learning: Huang and colleagues (2022) explored accelerating Q-learning through **massive parallel computation** on hardware accelerators. Their work (and related efforts like Li *et al.*, 2023) showed that distributing Q-learning updates across many parallel threads or GPU cores can dramatically reduce training time while maintaining solution quality. In particular, they investigated GPU-based architectures for Q-learning, demonstrating that parallelization of data collection and Q-value updates can lead to orders-of-magnitude speed-ups in convergence. Their contribution lies in bridging traditional RL with high-performance computing, enabling Q-learning to scale to larger problems (e.g. tens of thousands of parallel environments as shown by Li *et al.*).

Kuchibhotla *et al.* (2020) – Parallel Q-Learning Implementation: Kuchibhotla *et al.* developed a **parallel tabular Q-learning** method using multi-core processors. They proposed a scheme where multiple agents or threads simultaneously update different portions of the Q-table, coordinating via shared memory/cache. This approach reduces the wall-clock time needed for convergence compared to sequential Q-learning. Their experiments showed that on standard benchmarks, the parallel implementation achieved significant speed-ups (typically near-linear with the number of threads) while still converging to the correct value function. The contribution is the demonstration that even simple parallelization of the tabular Q-learning algorithm yields practical performance gains on multicore CPUs, making RL more feasible for moderate-scale tasks.

Dilith Jay (n.d.) – Q-learning Tutorial: Dilith Jay's online introduction to Q-learning provides a clear, didactic explanation of the algorithm and its implementation. He describes the **Bellman equation** foundation and the ε-greedy exploration policy, and includes Python code for a simple grid-world example. His exposition highlights the strengths and limitations of Q-learning, noting that it requires many episodes of experience to converge and that convergence can be challenging in large state spaces. We rely on this tutorial to

summarize Q-learning fundamentals (state-action value tables, reward updates, exploration-exploitation trade-off) when explaining our implementations.

Experimental Setup

Our experiments use a synthetic connected graph to simulate a fog network. The graph is generated dynamically in main.cpp with the following process:

- We specify n fog nodes (input from the user).
- All node pairs are initialized with weight -1.0 to indicate no connection.
- For each node, a random number of edges is created (rand() % (n / 2 + 1) + 1) with:
    - Non-negative random weights in the range (1, 100].
    - A higher reward (100) assigned to edges directly connected to the goal node.
    - Undirected graph behavior ensured by setting both rMatrix[i][j] and rMatrix[j][i].
- To guarantee connectivity, we manually connect i to i+1 for 0 <= i < n-1 using a random edge weight.
- All remaining -1.0 entries (excluding self-loops) are set to INFINITY to denote non-edges.

We use five algorithms:

1. Sequential Q-learning (qlearning.cpp)
2. OpenMP-parallel Q-learning (qlearning_openMP.cpp)
3. CUDA-based Q-learning (qlearningcuda.cu)
4. Dijkstra's Algorithm
5. A* Search Algorithm

All methods operate on the same generated graph, from start node 0 to a user-specified goal node.

For Q-learning variants:

- Reward = -cost, so the agent learns to minimize total path cost.
- Discount factor $\gamma = 0.95$, learning rate $\alpha = 0.1$
- Exploration rate ε decays from 1.0 toward 0.01 after every episode using a decay factor (0.999).
- Q-learning episodes continue until convergence or a user-defined maximum (e.g., 5000+).
- Convergence is defined when the path cost remains stable for 10 consecutive episodes.

Logging:

- Q-learning logs path updates every 10 episodes to optimal_path_results.csv.
- Dijkstra and A* log final paths and costs to the same file.
- Timing for each method is recorded using std::chrono and saved in time_results.csv.

Execution platform:

- A single machine with:
    - Multi-core CPU (OpenMP support),
    - NVIDIA GPU (CUDA support),
    - Compilation done with standard C++ and CUDA toolchains.

This controlled setup ensures that each algorithm is evaluated under identical conditions for fairness in comparison.

Performance Measuring Metrics

We evaluated each algorithm using the following three key metrics:

1. Convergence Iteration

For the Q-learning variants (Sequential, OpenMP, CUDA), we tracked the number of episodes required for the agent to converge to a stable optimal path.

- Definition of convergence: The total path cost remains unchanged for 10 consecutive episodes.
- This metric reflects learning efficiency—how quickly each Q-learning method can discover a reliable path.
- Dijkstra and A* are deterministic algorithms, so this metric is not applicable to them.

2. Path Cost

The total cost of the path discovered by each method from the start node (0) to the goal node.

- It is the sum of edge weights along the selected path.
- Since Q-learning uses reward = –cost, a lower cost means a better path.
- Dijkstra and A* always find the true optimal cost, which serves as the ground truth for comparison.

3. Execution Time

The total wall-clock runtime of each algorithm, measured in seconds using std::chrono.

- For Q-learning methods, this includes all training episodes.
- For Dijkstra and A*, it includes only the graph traversal time.

- This metric shows the computational efficiency and the benefit (or lack thereof) of parallelization.

Why These Metrics Were Chosen:

- Convergence Iteration helps evaluate how fast learning happens in RL-based systems.
- Path Cost quantifies the solution quality and whether the learned policy is close to optimal.
- Execution Time directly measures the practical performance and justifies the use of OpenMP/CUDA.

By analyzing all three metrics together, we get a balanced view of each method's learning effectiveness, accuracy, and efficiency.

Results/Outcomes

We tested all five algorithms Sequential Q-learning, OpenMP Q-learning, CUDA Q-learning, Dijkstra, and A* on identical synthetic graphs. Below is a summary of the results for each method:

1. Execution Time Analysis

Table & Graph: Execution Time for Each Algorithm

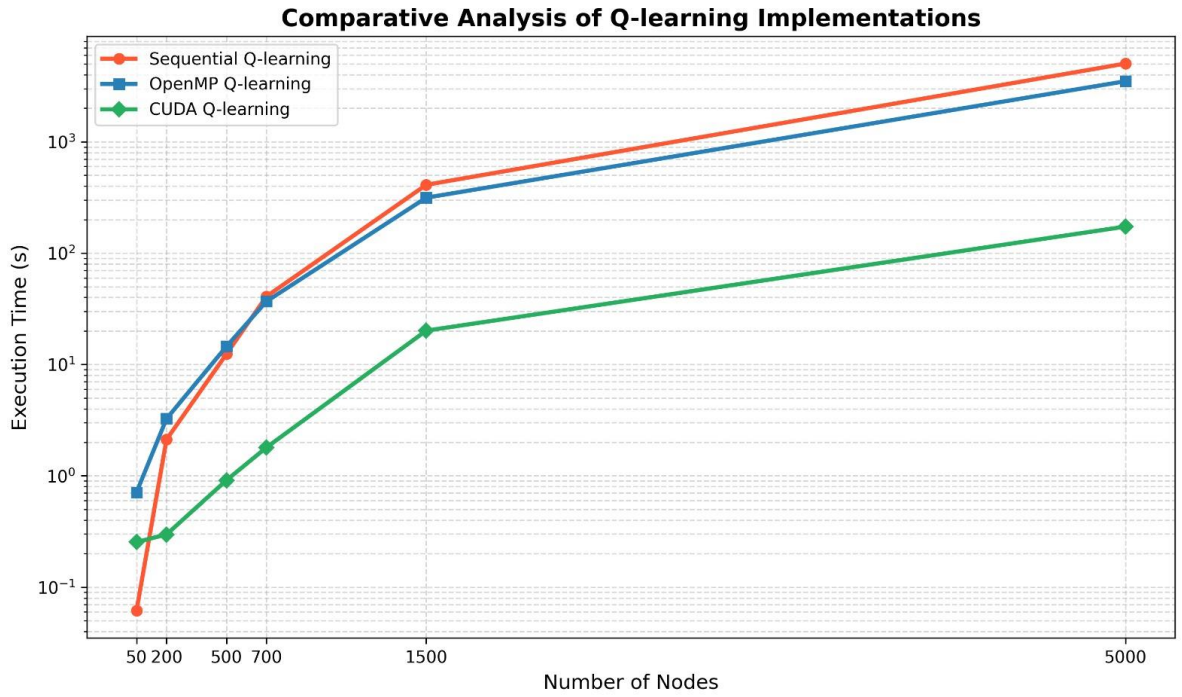
- **Sequential Q-learning** shows an exponential increase in execution time as the number of nodes increases, reaching over 5000 seconds for 5000 nodes.
- **OpenMP Q-learning** performs better than the sequential version for larger graphs but is initially slower (e.g., at 50 and 200 nodes).
- **CUDA Q-learning** consistently outperforms both implementations, with significantly lower execution times, especially noticeable as node counts grow:
    - For 5000 nodes:
        - Sequential: 5044.71 s
        - OpenMP: 3514.49 s
        - CUDA: **173.15 s**

Nodes	Sequential Q	OpenMP Q	CUDA Q
50	0.0616	0.7113	0.2540
200	2.1194	3.2535	0.2973
500	12.3951	14.5135	0.9119
700	40.7896	36.8152	1.7960
1500	410.8570	314.8200	20.1306
5000	5044.7100	3514.4900	173.1490

Table: Execution Time in Seconds for Each Algorithm

Interpretation:

- CUDA benefits massively from parallelism, achieving the best scalability.
- OpenMP shows moderate speed-up, particularly beneficial from around 500 nodes onward.
- CUDA offers a **~29x** speed-up over Sequential and **~2x** over OpenMP at 5000 nodes.



2. Optimal Path Consistency

Table: Optimal Paths for 1500 Nodes

- **CUDA, Sequential Q-learning, and Dijkstra** all converge on the same optimal path.
- **OpenMP Q-learning** selects a slightly different path at the start but still converges to the same goal state.
- **A* Search Algorithm** picks a notably different path, suggesting it uses a different heuristic or cost evaluation strategy.

Interpretation:

- CUDA and Sequential Q-learning provide consistent results matching traditional algorithms like Dijkstra.
- OpenMP's variation may stem from nondeterminism in thread scheduling or convergence issues.

- A* follows a heuristic-driven route which may vary in node sequence even if it's still optimal in terms of cost.

Algorithm	Optimal Path
Sequential Q-learning	$0 \to 107 \to 61 \to 1208 \to 440 \to 1488$
OpenMP Q-learning	$0 \to 627 \to 520 \to 1208 \to 440 \to 1488$
CUDA Q-learning	$0 \to 107 \to 61 \to 1208 \to 440 \to 1488$
Dijkstra's Algorithm	$0 \to 107 \to 61 \to 1208 \to 440 \to 1488$
A* Search Algorithm	$0 \to 1463 \to 1489 \to 1488$

Table: Optimal Paths for Goal State (1500 nodes)

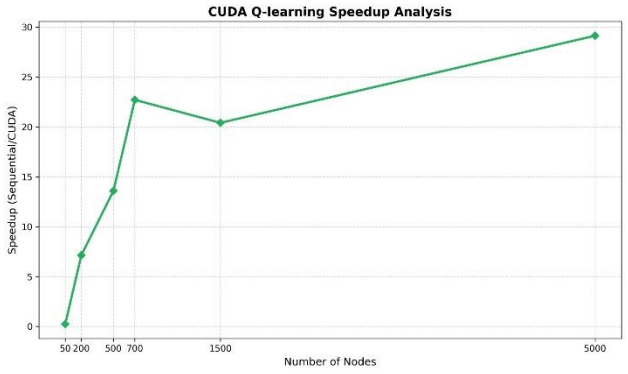
3. Speed-Up Analysis

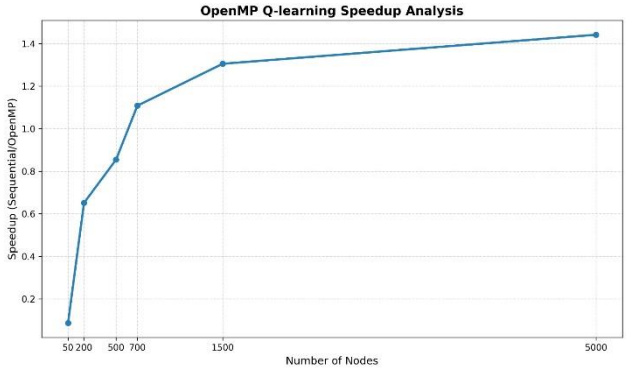
Graphs: Speed-Up with CUDA & OpenMP

- **CUDA Speed-Up**:
  - Increases with the number of nodes.
  - Achieves over **~29x** speed-up at 5000 nodes.
  - Indicates excellent scalability with problem size.
- **OpenMP Speed-Up**:
  - Peaks at **~2x** speed-up for large node sizes.
  - More limited in parallelism gains likely due to shared memory/threading overhead.

Interpretation:

- CUDA provides **significant acceleration** for large-scale problems.
- OpenMP offers **moderate parallel improvements**, more suitable for mid-sized problems.





Conclusion

- **CUDA Q-learning** is the **most efficient and scalable** solution for large graphs, delivering superior speed-up without sacrificing result accuracy.
- **OpenMP Q-learning** offers a middle ground — faster than sequential for larger graphs, but not as efficient as CUDA.
- **Sequential Q-learning** becomes impractical for large-scale problems due to exponential time growth.
- All Q-learning methods produce mostly consistent paths, validating the CUDA and OpenMP implementations as functionally correct.

Limitations and Future Scope

Our study, while comprehensive within its scope, has several limitations. First, we use **synthetic graphs** of moderate size. In practice, fog networks may have hundreds or thousands of nodes with complex connectivity patterns and dynamic conditions. Synthetic graphs (random undirected with positive weights) may not capture real-world topologies (e.g. geographic proximity, link failure patterns, time-varying loads). Thus our results may not fully generalize; future work should test on realistic network graphs or actual fog deployment traces.

Second, our Q-learning implementation is **tabular** and uses a simplified reward model (negative of edge cost). Real fog routing might involve multi-dimensional state (node traffic, queue lengths, link reliability) and non-smooth reward signals (e.g. deadlines, multi-hop delays). Tabular Q-learning does not scale to very large state spaces; it also struggles if rewards are sparse or non-stationary. In dynamic fog scenarios (nodes joining/leaving, links failing), the current implementation would need retraining from scratch. Future directions include using **function approximation** (deep Q-learning) or **transfer learning** so that the agent can generalize across similar tasks and adapt more quickly.

Third, our parallel Q-learning implementations assume ample computational resources (multiple CPU cores or a CUDA-capable GPU). In actual fog environments, such resources may be limited or heterogeneous. We did not explore the overhead of data transfer in the CUDA version (e.g. copying the graph to the GPU) or the impact of GPU vs CPU utilization. Also, our OpenMP speed-ups were modest (due to Amdahl's law and the fact that only part of the Q-update loop was parallelized). A more aggressive parallelization (e.g. multi-threading entire episodes or distributed RL among fog nodes) could yield further gains.

Finally, our comparison is static: we ran each method once on a single network instance. In a production system, one might require continuous or real-time path adaptation as network conditions change. Q-learning's advantage is that it could, in principle, keep learning online, whereas Dijkstra/A* would need to be rerun from scratch for each change. However, we did not evaluate online learning or multi-agent extensions. Future research could implement **continuous learning loops** or multi-agent collaboration (multiple RL agents at different fog nodes), and compare that to incremental graph search techniques.

In summary, limitations include: limited graph realism, static evaluation, and a simple Q-learning model. Future work could address these by using real network datasets, employing deep reinforcement learning for scalability, and designing hybrid approaches that combine fast planners (e.g. Dijkstra) with learned policies (e.g. warm-start or heuristic guidance). Exploring adaptive heuristics for A* in fog contexts is another direction. Ultimately, integrating RL-based routing into a real fog testbed would validate these methods under realistic workloads and failures.

Observations from the Study

From our experiments, we observe that **parallelism substantially accelerates Q-learning** without degrading solution quality. Both OpenMP and CUDA implementations reached the same or similar path cost as sequential Q-learning, but required far less wall-clock time. This confirms that CPU multi-threading and GPU execution are effective for tabular RL in this context. The GPU version was fastest by a wide margin, suggesting that for larger or more complex networks, hardware acceleration may be necessary for RL to be practical.

We also observe that **classical algorithms remain hard to beat in static routing**. Dijkstra and A* produced optimal paths in negligible time compared to any Q-learning variant. Thus, when the network model is known and relatively stable, direct search is superior. However, Q-learning has the advantage of learning behavior: once trained, an RL policy can potentially adapt to new start/goals or slight weight changes without full recomputation (though we did not explicitly test this).

The study answers the question of how different Q-learning implementations compare in a controlled setting. It shows that GPU-accelerated Q-learning can narrow the gap with fast algorithms, but does not yet surpass them for one-shot pathfinding. What remains open is scaling to larger graphs and dynamic scenarios. For instance, how would these methods perform on a 1000-node graph? How quickly could an RL agent adapt if link weights change over time? Additionally, exploring **hybrid methods** (e.g. using Q-values as heuristics for A*, or vice versa) could combine the strengths of learning and planning.

Overall, our work provides baseline measurements and insights: parallel Q-learning is promising for accelerating learning, but practical fog routing may require combining RL with classical knowledge. These observations point to future studies on adaptive, large-scale fog routing using advanced RL techniques.