Shortest Path Finding in Dynamic Graphs using Q-learning and Deep Q-Networks

Krishna Sharma[1], Pratik Rana[1], Rahul Kumar[1]
[1]*Computer Science and Artificial Intelligence, Plaksha University, Punjab, India*
krishna.sharma@plaksha.edu.in, pratik.rana@plaksha.edu.in, rahul.kumar@plaksha.edu.in

Abstract

Finding the shortest path in dynamic graphs, where edge weights change over time, poses significant challenges for traditional algorithms like Dijkstra's or A*, which require complete recalculation upon graph changes. This report investigates Reinforcement Learning (RL) techniques, specifically Q-Learning and Deep Q-Networks (DQN), as adaptive solutions for this problem. We formulate the shortest path problem within the RL framework and implement three approaches: standard Q-Learning with fixed/adaptive training episodes, Q-Learning with an expanded action space, and DQN for improved scalability. Experiments compare these methods against the D* Lite algorithm across varying graph sizes. Results demonstrate the adaptability of RL methods and their potential for efficient pathfinding in dynamic scenarios, with significantly faster inference times compared to D* Lite's replanning.

Keywords: Reinforcement Learning, Q-Learning, Deep Q-Network (DQN), Shortest Path, Dynamic Graphs, Pathfinding, Adaptive Algorithms.

1 Problem Statement and RL Formulation

1.1 The Challenge of Dynamic Shortest Paths

The shortest path problem is fundamental in graph theory, with applications from network routing to robotics. In a graph $G = (V, E)$ with edge weights $w(u, v)$, the goal is to find a path $P = (v_0, v_1, ..., v_k)$ from source $s = v_0$ to destination $d = v_k$ that minimizes the total cost $C(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$. While Dijkstra's and A* algorithms solve this optimally for static graphs, they face limitations in *dynamic* environments where graph properties change over time, particularly when edge weights $w(u, v)$ vary temporally due to factors like traffic congestion or network load.

Traditional shortest path algorithms are generally designed for static graphs. When the graph changes, they typically require partial or complete recalculation of the path from the current location or even the source. For large graphs or environments with frequent changes, this continuous replanning becomes computationally expensive and may not be feasible for real-time applications. Algorithms like D* Lite were developed to handle dynamic changes more efficiently by reusing previous computations, but they still involve explicit replanning steps when changes affecting the current optimal path are detected.

1.2 Reinforcement Learning as an Adaptive Solution

Reinforcement Learning (RL) offers a promising alternative paradigm for tackling the shortest path problem in dynamic graphs. RL involves an *agent* learning to make optimal decisions by interacting with an *environment* over time. The agent observes the state of the environment, takes actions, receives rewards (or penalties), and updates its internal *policy* (decision-making strategy) to maximize its cumulative reward.

The key advantage of RL in this context is its inherent adaptability. An RL agent can learn a policy that implicitly accounts for potential changes in the environment. By experiencing various graph states (e.g., different edge weight configurations) during training, the agent can learn robust strategies that perform well even when the environment deviates from previously seen conditions. Furthermore, once trained, executing the learned policy (i.e., deciding the next node to move to) is often computationally very fast, typically involving a simple lookup (in tabular methods like Q-learning) or a single forward pass through a neural network (in DQN). This contrasts sharply with the potentially costly replanning required by algorithms like D* Lite during execution time when graph changes occur.

1.3 Reinforcement Learning Formulation

To apply RL to the shortest path problem, we define the standard components of a Markov Decision Process (MDP):

- **Environment:** The dynamic graph $G = (V, E, W(t))$, where $W(t)$ represents the edge weights at time step t. The environment provides the current state and rewards based on the agent's actions. In our simulations, dynamism is introduced by updating edge weights at each step of an episode based on a time-dependent function (e.g., $w(u, v, t) = base\_w_{uv} + variation_{uv} \times \sin(t \times period_{uv})$), simulating fluctuating conditions like traffic.

- **Agent:** The entity navigating the graph, aiming to find the shortest path from a given start node s to a goal node d.
- **State (S):** The information the agent uses to make decisions. In our implementations, we explored two primary state representations:
  - For standard Q-Learning and expanded action space Q-Learning: $s = (current\_node, goal\_node)$. This representation explicitly informs the agent of its current location and its target.
  - For DQN: $s = \text{concat}(one\_hot(current), one\_hot(goal), adj(current))$, a richer vector including one-hot encodings of the current and goal nodes, concatenated with a binary adjacency vector indicating neighbors of the current node. This provides more structural information to the neural network.
- **Action Space ($\mathcal{A}$):** The set of possible moves the agent can take from a given state.
  - Standard QL / DQN: $\mathcal{A}(s) = \{v \mid (current\_node, v) \in E\}$, that is, the set of adjacent nodes to the current node.
  - Expanded Action Space QL: $\mathcal{A}(s) = V$, i.e., the set of all nodes in the graph. Invalid moves (to non-adjacent nodes) are handled via rewards.
- **Reward Function ($R(s, a, s')$):** A scalar value indicating the immediate desirability of taking action a in state s and transitioning to state s'. Our reward structure is designed to encourage reaching the goal quickly via low-cost edges:
  - $R = +100$ for reaching the goal node d.
  - $R = -w(current\_node, next\_node)$ for taking a valid step (moving along an existing edge), where w is the current weight of that edge. This penalizes longer paths.
  - $R = -P_{invalid}$ (e.g., -100 or -200) for attempting an invalid move (e.g., moving to a non-adjacent node in the expanded action space model).
  - (Optional, used in DQN): Small penalties for loops or exceeding step limits can be added.
- **Policy ($\pi(a|s)$):** The agent's strategy, mapping states to actions. In Q-Learning and DQN, this is derived from the learned action-value function (Q-function), using a greedy or epsilon-greedy approach. $\pi(s) = \arg\max_{a \in \mathcal{A}(s)} Q(s, a)$.
- **Agent's Goal:** Learn a policy π that maximizes the expected cumulative discounted reward (return) from any start state s: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $\gamma \in [0, 1]$ is the discount factor.
- **Episode Termination:** An episode (a single pathfinding attempt from s to d) terminates when:
  - The agent reaches the goal node d.
  - The agent exceeds a maximum number of steps (e.g., $N \times factor$).
  - The agent gets stuck (no valid actions possible, though unlikely in connected graphs with our action spaces).

This formulation allows RL algorithms to learn pathfinding strategies by balancing the immediate cost of edges with the long-term goal of reaching the destination efficiently, while adapting to the dynamically changing edge weights encountered during training episodes.

2 Methodology Used and Contributions

This project explores and compares several Reinforcement Learning approaches for shortest path finding in dynamic graphs, benchmarking them against the D* Lite algorithm. Our core methodologies focus on Q-Learning variations and Deep Q-Networks (DQN).

2.1 Q-Learning Approaches

Q-Learning is a model-free, off-policy reinforcement learning algorithm that estimates the optimal action-value function $Q(s, a)$, representing the expected discounted reward of taking action a in state s and following the optimal policy. The Q-values are updated using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{1}$$

where α is the learning rate and γ is the discount factor.

In our pathfinding setup:

- **State:** $s = (\texttt{current\_node}, \texttt{goal\_node})$

- **Q-Table:** Nested dictionary 'q_table[state][action]'
- **Exploration:** ϵ-greedy policy; ϵ decays exponentially: $\epsilon = \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}})$
- **Replay Buffer:** Transitions $(s, a, R, s', \text{done})$ stored in a deque; mini-batches sampled for updates using Eq. 1

We investigated two main variants of Q-Learning:

2.1.1 Standard Q-Learning with Adaptive Episodes

This approach uses the standard RL formulation where the action space $\mathcal{A}(s)$ consists only of nodes directly adjacent to the agent's current node. A key contribution here was experimenting with the number of training episodes. We compared:

- **Fixed Episode Count:** Training each graph size for a constant number of episodes (e.g., 20,000).
- **Adaptive Episode Count:** Scaling the number of training episodes proportionally to the graph size, e.g., $N_{episodes}$ = BaseEpisodes $\times$ (N/N_{min}), where N is the current number of nodes and N_{min} is the smallest graph size tested i.e. 20 and BaseEpisodes are 20000. The motivation is that larger, more complex graphs likely require more exploration and learning iterations to converge to a good policy. The results section analyzes the impact of this adaptive strategy on performance metrics like cost optimality and success rate as graph size increases.

2.1.2 Q-Learning with Expanded Action Space

This novel variant modifies the standard Q-Learning framework by expanding the action space $\mathcal{A}(s)$ to include *all* nodes in the graph, regardless of adjacency.

- **Action Space:** $\mathcal{A}(s) = V = \{0, 1, ..., N-1\}$.
- **Reward Modification:** To handle this, the reward function is adjusted. If the agent chooses an action a corresponding to a node v that is *not* adjacent to the current node u, a large negative penalty ($R = INVALID\_MOVE\_PENALTY = -200$) is given, and the agent's state remains unchanged (it doesn't actually move). If the chosen action a corresponds to a valid neighbor v, the standard reward based on edge weight $w(u, v)$ and goal proximity is calculated as before.

The hypothesis is that this allows the agent to directly learn Q-values comparing moves to any node, potentially discovering non-obvious shortcuts or implicitly learning graph connectivity. However, this significantly increases the size of the action space and the complexity of the learning task, requiring more exploration and potentially more episodes to learn effectively due to the frequent encountering of penalties for invalid moves. The training process involved significantly more episodes (e.g., scaling up to 120,000 for 60 nodes) compared to standard Q-learning to manage the larger action space and frequent penalties.

2.2 Deep Q-Network (DQN) Approach

For larger graphs, the state-action space of tabular Q-learning becomes prohibitively large ($O(N^2 \times |\mathcal{A}|)$). Deep Q-Networks (DQN) address this by using a neural network to approximate the Q-function: $Q(s, a; \theta) \approx Q^*(s, a)$, where θ represents the network parameters.

Our DQN implementation features:

- **State Representation:** A crucial design choice for graph-based tasks. We used an enhanced vector representation: $s_{vec} = \text{concat}(one\_hot(current), one\_hot(goal), adj(current))$.
  - $one\_hot(current)$: One-hot vector of size N for the current node.
  - $one\_hot(goal)$: One-hot vector of size N for the goal node.
  - $adj(current)$: A binary vector of size N, where the i-th element is 1 if node i is adjacent to the current node in the current graph state, and 0 otherwise.

  This results in an input vector of size $3N$ for the neural network, providing location, goal, and local connectivity information.
- **Network Architecture:** A simple Multi-Layer Perceptron (MLP) was used:
  - Input Layer: Linear layer with $3N$ inputs.
  - Hidden Layers: Two fully connected hidden layers with 128 neurons each, using ReLU activation functions ($f(x) = \max(0, x)$).
  - Output Layer: Linear layer producing N outputs, representing the Q-values for taking an action corresponding to moving towards each of the N nodes in the graph, $Q(s, \cdot; \theta)$.

- **Action Selection:** Similar to Q-learning, ϵ-greedy is used during training. During exploitation, the network predicts Q-values for all N possible target nodes. However, only actions corresponding to valid neighbors of the current node are considered. A mask is applied to the network output (setting Q-values for non-neighbors to $-\infty$) before selecting the neighbor with the highest predicted Q-value.
- **Training:**
  - *Experience Replay:* Transitions $(s_t, a_t, R_{t+1}, s_{t+1}, \text{done})$ are stored in a replay memory buffer. s_t and s_{t+1} are the state tensors. a_t is the index of the chosen action node.
  - *Optimization:* Mini-batches are sampled from memory. The loss is calculated using the difference between the predicted Q-value $Q(s_t, a_t; \theta_{\text{policy}})$ and the target Q-value y_t:

$$y_t = R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_{\text{target}}) \quad \text{(if not done)} \tag{2}$$

    where θ_{policy} are the parameters of the policy network being trained, and θ_{target} are the parameters of a separate target network. The Smooth L1 loss (Huber loss) was used between $Q(s_t, a_t; \theta_{\text{policy}})$ and y_t. The AdamW optimizer was used to update θ_{policy}.
  - *Target Network:* The target network parameters θ_{target} are periodically updated (e.g., every 150 episodes) by copying the parameters from the policy network ($\theta_{target} \leftarrow \theta_{policy}$). This stabilizes training by providing a more consistent target value during updates.

2.3 Simulation of Dynamic Environment

Across all experiments, the dynamic nature of the graph was simulated by recalculating edge weights at each time step within an episode. The weight $w_{uv}(t)$ of an edge (u, v) at time step t was determined by:

$$w_{uv}(t) = \max(0.1, \text{base\_w}_{uv} + \text{var}_{uv} \times \sin(t \times \text{period}_{uv})) \tag{3}$$

where base_w_{uv}, var_{uv}, and period_{uv} are parameters assigned during graph creation, providing variability in both the magnitude and frequency of weight changes for different edges. This models scenarios like fluctuating traffic or network load. For benchmark comparisons, both the RL agent and D* Lite operated within this same dynamic environment during their respective path executions.

2.4 Key Contributions

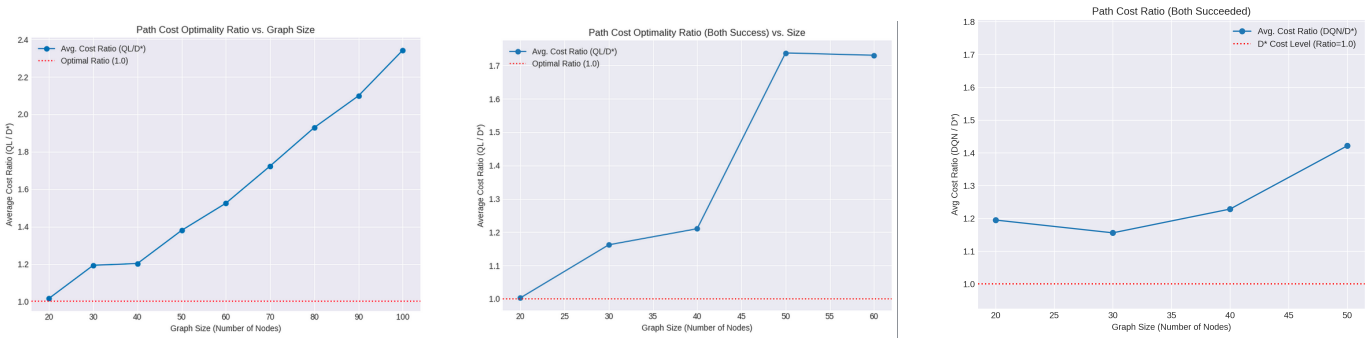
This work makes the following contributions:

1. **Adaptive Episode Strategy:** We systematically investigated the impact of adapting the number of Q-Learning training episodes based on graph size, demonstrating its effect on achieving better policy convergence for larger graphs compared to a fixed episode count.
2. **Expanded Action Space Q-Learning:** We proposed and evaluated a novel Q-Learning variant where the agent can choose any node as an action, learning implicitly about connectivity through penalties for invalid moves. This explores a different trade-off in RL design for graph problems.
3. **Comparative Analysis:** We provided a comprehensive benchmark comparing standard Q-Learning (with fixed and adaptive episodes), expanded action space Q-Learning, and DQN against the D* Lite algorithm for shortest path finding in dynamic graphs of varying sizes, analyzing performance in terms of success rate, path cost optimality, path steps, and execution time.
4. **Scalable RL Solution Exploration:** We demonstrated the potential application and evaluated the scalability of different RL methods for handling larger state spaces inherent in graph-based problems compared to tabular Q-learning.

The code implementing these methods and experiments is available at: https://github.com/RahulKumar-007/Shortest_Path_Finding_in_Dynamic_Graphs/tree/main

3 Results

This section presents the experimental results comparing the performance of the implemented Reinforcement Learning methods (Q-Learning Standard, Q-Learning Expanded Action Space, DQN) against the D* Lite baseline algorithm across dynamic graphs of varying sizes (20 to 100 nodes for standard QL, 20 to 60 nodes for expanded QL, 20 to 50 nodes for DQN, based on available output). Performance is evaluated based on success rate, path cost optimality (relative to D* Lite, shown in Figure 1), path steps, and execution time.



(a) Standard QL Cost Ratio (b) Expanded QL Cost Ratio (c) DQN Cost Ratio (up to 50 nodes)

Figure 1: Cost Ratio (RL Method Cost / D* Cost) vs. Graph Size for different RL approaches. Lower is better (closer to 1.0). Refer to text and linked repository for other metrics and detailed distributions.

3.1 Q-Learning Performance

3.1.1 Training Progress and Adaptive Episodes

The training progress for standard Q-Learning using the adaptive episode strategy (where episodes scale with graph size, e.g., $N_{episodes} = 1000 \times N$) is visualized in plots and can be found in our GitHub repository at this Link. These plots show the total reward per episode and steps per episode over the training duration for graph sizes 20 through 100.

The necessity of sufficient training episodes, especially for larger graphs, was evident and illustrated by These Plots. For size 30, performance metrics like cost optimality ratio stabilize relatively quickly (around 10k-20k episodes). However, for the larger size 60 graph, the cost ratio and step ratio continue to improve significantly even up to 40k-60k episodes, highlighting that insufficient training (e.g., only 5k episodes) leads to substantially suboptimal paths (higher cost/step ratios). This supports the rationale behind the adaptive episode strategy, which allocates more training time to larger graphs. The success rate reaches 100

3.1.2 Performance vs. Graph Size (Standard QL - Adaptive Episodes)

Figure 1a shows the overall cost ratio trends as graph size increases for standard Q-Learning trained with adaptive episodes. We also Keep track of following Metrics and corresponding plots can be found here:

- **Success Rate:** Both Q-Learning and D* Lite maintain a 100% success rate across tested sizes, indicating both methods reliably find the goal in connected graphs within the step limit.
- **Cost Ratio (QL/D*):** As seen in Fig. 1a, the average cost ratio increases steadily with graph size, starting near optimal (1.0) for small graphs but rising towards 2.0-2.5 for larger graphs (size 100). This indicates that while Q-Learning finds paths, they become progressively less cost-optimal compared to D* Lite as the state space grows, even with adaptive training episodes. The increased complexity and dynamic nature make it harder for tabular Q-learning to find the truly optimal path consistently.
- **Step Ratio (QL/D*):** A similar trend is observed for the number of steps (see linked plots), increasing from near 1.0 to around 1.8-1.9, meaning Q-Learning paths involve more hops.
- **Execution Time:** The key advantage of Q-Learning is evident here (see linked plots). The average QL decision time (policy execution) remains significantly lower (by orders of magnitude, note the log scale) than the D* Lite total execution time (which includes replanning). While D* Lite's time increases notably with graph size, QL's inference time grows much more slowly.

The distribution of these metrics is further detailed in box plots available at This Figure. The box plots confirm the increasing cost/steps for QL relative to D* Lite as size grows, but also visually emphasize the much lower and tighter distribution of QL decision times compared to D* total times.

3.1.3 Performance vs. Graph Size (Expanded Action Space QL)

Figure 1b shows the overall cost ratio trends as graph size increases for expanded Q-Learning trained with adaptive episodes. We also Keep track of following Metrics and plots can be found here:

- **Success Rate:** This method also achieved a 100% success rate across the tested sizes (20-60).

- **Cost Ratio (QL/D*):** Compared to standard QL, the cost ratio (Fig. 1b) starts slightly higher and increases more rapidly, reaching around 1.7 for size 60 (compared to 1.5 for standard QL with adaptive episodes at size 60 in Fig 1a). This suggests that learning an effective policy is harder with the vastly larger action space, despite significantly more training episodes allocated (up to 120k for size 60). The agent struggles more to avoid suboptimal paths when faced with many invalid choices carrying heavy penalties.
- **Step Ratio (QL/D*):** The step ratio (see linked plots) also climbs more steeply, indicating longer paths in terms of hops.
- **Execution Time:** Similar to standard QL, the decision time (see linked plots) remains much faster than D* Lite's total time.

While conceptually interesting, the expanded action space approach proved harder to train effectively within the tested episode counts, leading to less optimal paths compared to standard Q-learning, despite achieving reliable goal-reaching.

3.2 DQN Performance

The DQN agent was evaluated on graph sizes 20 to 50 as Larger node size Graph Neural Network took too much time for training even on Hardware acceleration. Figure 1c shows the overall cost ratio trends as graph size increases for DQN trained with adaptive episodes. We also Keep track of following Metrics and Plots can be found here:

- **Success Rate :** DQN achieved 100% success rate for sizes 20, 30, and 50, with a slight dip to 96.67% at size 40 in this specific run (potentially indicating insufficient training or suboptimal hyperparameters for that size). D* Lite achieved 100% success across all sizes (see linked plots).
- **Cost Ratio (DQN/D*) :** The average cost ratio (Fig. 1c) starts slightly higher than standard QL (around 1.2) and increases to about 1.42 at size 50. This indicates that DQN finds paths less optimal than D* Lite, and slightly less optimal than the well-trained standard QL in the smaller graph sizes tested here. However, its function approximation approach is expected to scale better to much larger graphs where tabular Q-learning becomes intractable.
- **Step Ratio (DQN/D*):** The step ratio (see linked plots) remains relatively stable, hovering between 1.07 and 1.18, suggesting DQN paths are only slightly longer in terms of hops compared to D* Lite.
- **Path Match Rate :** The rate at which the DQN path exactly matches the D* Lite path (when both succeed) decreases significantly as graph size increases, from over 56% at size 20/30 down to about 36% at size 50 (see linked plots). This indicates that while DQN finds the goal, its learned policy often diverges from the optimal path found by D* Lite, especially in larger graphs.
- **Execution Time:** The DQN execution time (forward pass through the network) was consistently very low (average 2-4ms based on benchmark summaries, see linked plots), offering a similar inference speed advantage over D* Lite as observed with Q-Learning.

3.3 Discussion Summary

- **RL vs. D* Lite:** QL and DQN effectively learned navigation in dynamic graphs with faster decision times than D* Lite. However, their paths were generally less cost-optimal, especially for QL on larger graphs (Fig. 1).
- **QL Variants:** Adaptive-episode QL maintained better cost/step ratios on large graphs than fixed-episode training (Fig. 1a). Expanded-action QL reached goals but failed to learn optimal policies efficiently due to invalid move penalties (Fig. 1b).
- **DQN Scalability:** DQN handled structural graph input well and is better suited for larger graphs where tabular QL is infeasible (Fig. 1c). Its decreasing path match rate suggests suboptimal but valid paths and indicates a need for hyperparameter tuning.
- **Training Effort:** RL methods require extensive offline training, especially on larger graphs. Adaptive-episode QL improves efficiency, but tuning duration and hyperparameters remains critical.

4 Conclusion

The project showed that Q-Learning and DQN are effective for finding shortest paths in graphs with dynamic edge weights. Both methods achieved faster inference than the D* Lite algorithm. While standard Q-Learning balanced training effort and performance well, the expanded action space variant was harder to optimize. DQN demonstrated scalability with further tuning needed for better path quality. Overall, reinforcement learning offers a valuable trade-off: higher offline training costs for faster online decisions, ideal for dynamic, known environments.