Samin Naji

Solving the Eight Puzzle Using A* Search

1. Definition of the Eight Puzzle as a Search Problem

The Eight Puzzle is a classic search problem consisting of a 3×3 board containing eight numbered tiles (numbered 1 through 8) and one empty space (blank). The objective is to transform a given initial configuration into a predefined goal configuration by sliding tiles into the empty space.

State Representation

Each state of the puzzle is represented as a 9-element sequence, corresponding to the tiles arranged row by row.

The numbers 1–8 represent the tiles.

The number 0 represents the empty space.

This representation is compact, easy to manipulate, and suitable for use in search algorithms because it can be efficiently compared and stored.

2. Successor State Generation

From any given state, successor states are generated by moving the empty space (0) up, down, left, or right, provided the move stays within the boundaries of the 3×3 grid.

The process for generating successors is as follows:

Locate the position of the empty space.

Determine which moves are valid based on its position.

Swap the empty space with the adjacent tile corresponding to each valid move.

Each swap produces a new state.

Each move has a uniform cost of 1, making this a unit-cost search problem.

3. Heuristic Functions

To guide the A* search algorithm, two heuristic functions were used.

3.1 Heuristic h_1: Number of Misplaced Tiles

The Misplaced Tiles heuristic counts the number of tiles that are not in their correct positions compared to the goal state, excluding the empty space.

This heuristic provides a simple estimate of how far the current state is from the goal.

Each misplaced tile must be moved at least once to reach its correct position.

3.2 Heuristic h_2: Manhattan Distance

The Manhattan Distance heuristic calculates, for each tile, the distance between its current position and its goal position. This distance is computed as:

$$|\Delta row| + |\Delta column|$$

The heuristic value is the sum of these distances for all tiles (excluding the empty space).

This heuristic accounts for how many grid moves are required to place each tile correctly.

It provides a more detailed and informative estimate than misplaced tiles.

4. Admissibility of the Heuristics

A heuristic is admissible if it never overestimates the true minimum cost required to reach the goal.

Misplaced Tiles Heuristic

Each misplaced tile must be moved at least once.

Therefore, the number of misplaced tiles is always less than or equal to the actual number of moves required.

Hence, this heuristic is admissible.

Manhattan Distance Heuristic

Each tile must move at least its Manhattan distance to reach the correct position.

Since tiles can only move one step at a time, the total Manhattan distance cannot exceed the true cost.

Thus, this heuristic is also admissible.

Because both heuristics are admissible, A* search using either heuristic is guaranteed to find an optimal solution.

5. Strength Comparison of the Heuristics

Intuitively, the Manhattan Distance heuristic is stronger than the Misplaced Tiles heuristic.

Misplaced tiles only consider whether a tile is incorrect.

Manhattan distance considers how far each tile is from its goal position.

As a result, Manhattan distance provides more accurate guidance toward the goal.

A stronger heuristic typically leads to fewer node expansions during search.

6. Experimental Comparison and Analysis

Based on experimental runs using A* search with both heuristics, the following observations were made:

Node Expansion

Manhattan Distance expanded fewer nodes than Misplaced Tiles.

This indicates that it guided the search more effectively toward the goal.

Path Length

Both heuristics found solutions with the same path length, confirming that both produced optimal solutions.

This is expected because both heuristics are admissible.

Efficiency

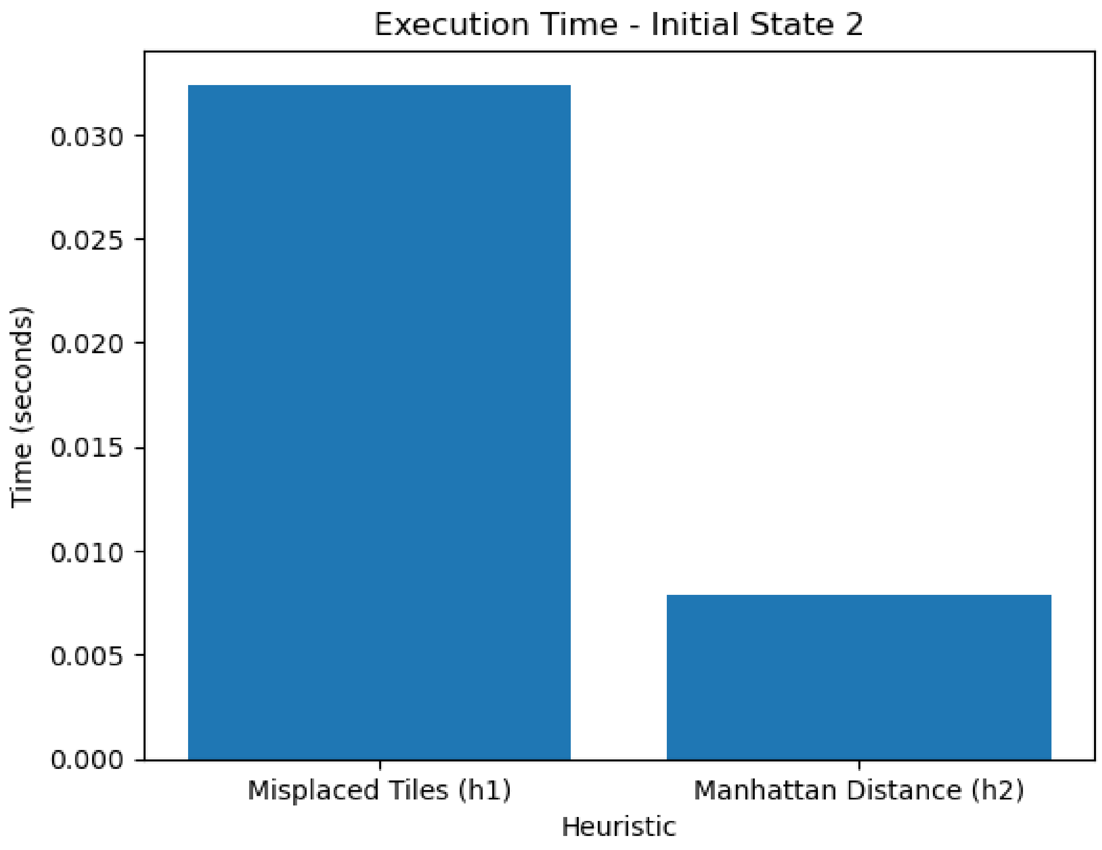
Manhattan Distance was more efficient overall, both in terms of execution time and number of expanded nodes.

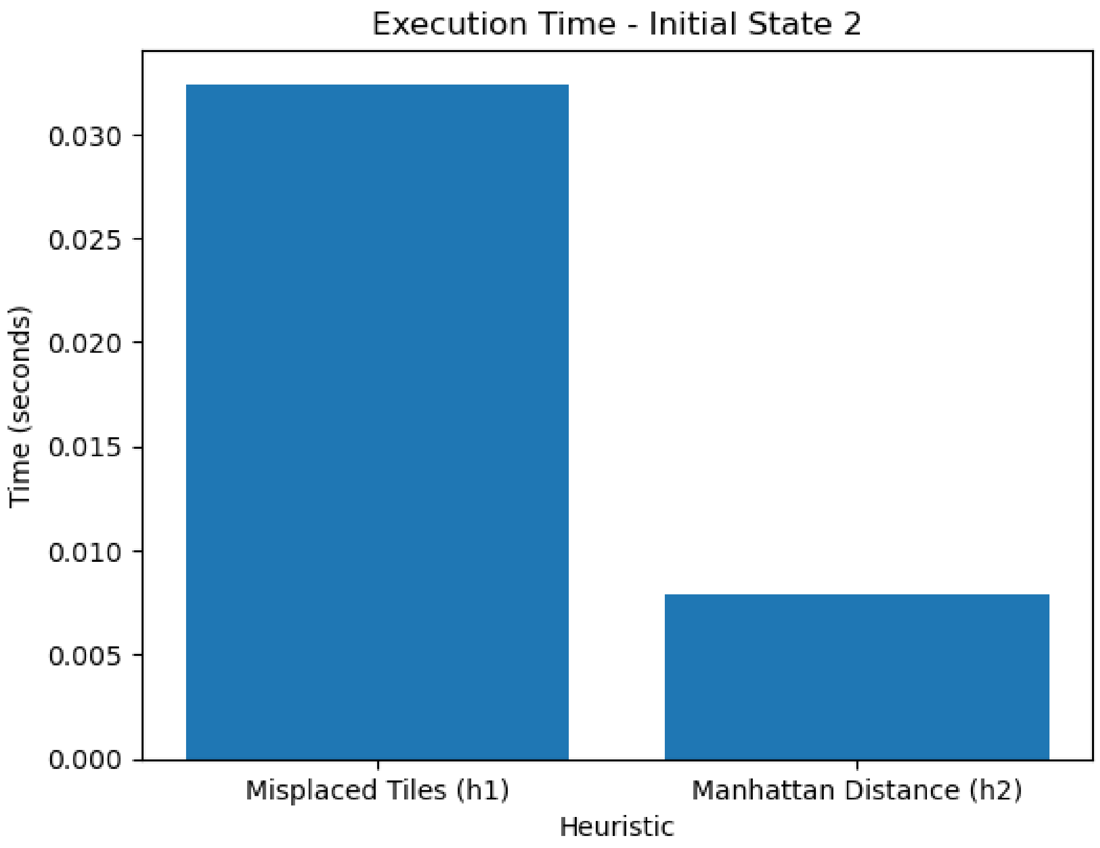
The improved efficiency comes from its more informative estimates, which reduce unnecessary exploration.

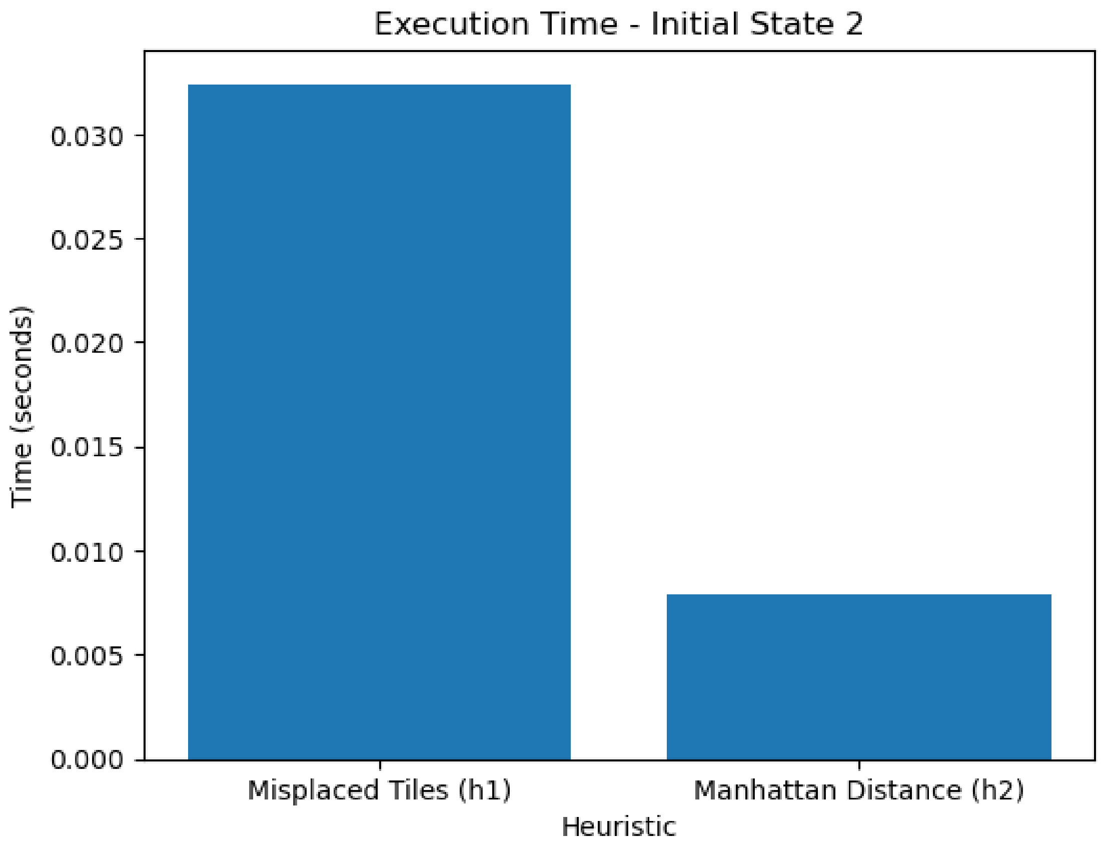
7. Conclusion

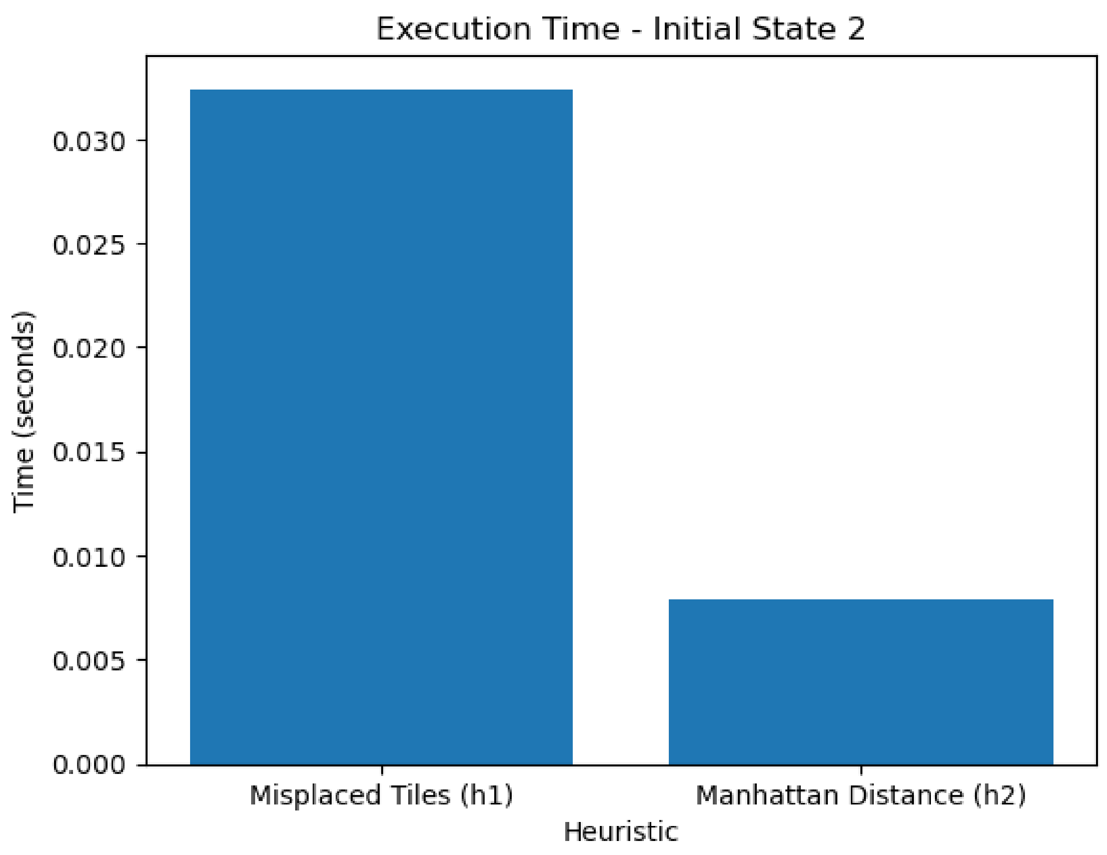
The Eight Puzzle can be effectively modeled as a search problem and solved using A* search. While both Misplaced Tiles and Manhattan Distance heuristics are admissible and guarantee optimal solutions, the Manhattan Distance heuristic is generally superior. It expands fewer nodes, runs faster, and provides more accurate guidance toward the goal. Therefore, Manhattan Distance is the preferred heuristic for solving the Eight Puzzle using A* search.

The experimental results, illustrated in Figures 1 to 4, show that the Manhattan Distance heuristic consistently expands fewer nodes and requires less execution time than the Misplaced Tiles heuristic. While both heuristics always find an optimal solution with the same path length, Manhattan Distance is more efficient because it provides a more accurate estimate of the remaining cost to the goal. This advantage becomes more pronounced for more complex initial states.




Execution Time - Initial State 2
Time (seconds)
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Misplaced Tiles (h1)
Manhattan Distance (h2)
Heuristic


Execution Time - Initial State 2
Time (seconds)
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Misplaced Tiles (h1)
Manhattan Distance (h2)
Heuristic



Solving the Eight-Puzzle Problem Using Q-Learning

1. Introduction

The Eight-Puzzle is a classical problem in artificial intelligence consisting of a 3×3 board with eight numbered tiles and one empty space. The objective is to reach a predefined goal configuration by sliding tiles into the empty position. In this work, the problem is modeled as a **Markov Decision Process (MDP)** and solved using **tabular Q-Learning** implemented within a **Gym-compatible environment**.

---

2. Markov Decision Process (MDP) Definition

The Eight-Puzzle is defined as an MDP ⟨S, A, R, γ⟩:

State Space (S)

- Each state represents a unique configuration of the 3×3 board.
- The board contains values {0, 1, ..., 8}, where `0` denotes the empty tile.
- Internally, states are stored as NumPy arrays of shape (3, 3).

- For Q-Learning, each state is converted into a discrete key using a flattened tuple representation.

Action Space (A)

A discrete action space with four actions:

- `0`: Move empty tile **Up**
- `1`: Move empty tile **Down**
- `2`: Move empty tile **Left**
- `3`: Move empty tile **Right**

Reward Function (R)

- Valid move: **−1**
- Invalid move (attempting to move out of bounds): **−5**
- Reaching the goal state: **+100**

This reward structure encourages short solution paths and discourages illegal actions.

Transition Model

- Transitions are deterministic.
- Valid actions swap the empty tile with an adjacent tile.
- Invalid actions leave the state unchanged.

Discount Factor (γ)

- γ = **0.99**, giving high importance to future rewards.

Episode Termination

An episode terminates when:

- The goal state is reached, or
- The maximum number of steps (200) is exceeded.

---

3. Gym Environment Description

The environment is implemented as `EightPuzzleEnv`, inheriting from `gym.Env`.

Observation Space

```
spaces.Box(low=0, high=8, shape=(3, 3), dtype=np.int32)
```

Action Space

```
spaces.Discrete(4)
```

Reset Mechanism

- If an initial state is provided, the environment starts from that configuration.

- Otherwise, the environment starts from the goal state and applies random valid moves to generate a scrambled puzzle.

Rendering

The `render()` method prints the puzzle grid to the console, using _ to represent the empty tile.

---

4. Q-Learning Algorithm Implementation

Overview

Q-Learning is a model-free, off-policy reinforcement learning algorithm that learns an action-value function Q(s, a) representing the expected cumulative reward for taking action a in state s.

State Representation

- States are converted to hashable keys using:

```
tuple(state.flatten())
```

- This allows storage in a Python dictionary.

Q-Table

- Implemented as a dictionary mapping state keys to action-value arrays.
- Initialized lazily using:

```
defaultdict(lambda: np.zeros(env.action_space.n))
```

Action Selection (ε-Greedy Policy)

- With probability ε, a random action is selected (exploration).
- Otherwise, the action with the highest Q-value is chosen (exploitation).

Q-Update Rule

- $\boldsymbol{Q(s,a) \leftarrow Q(s,a) + \alpha[\, r + \gamma \max a' \, Q(s',a') - Q(s,a)\,]}$

Expanded Nodes

- Each environment step during training is counted as one expanded node.
- This approximates the number of state expansions performed by the agent.

---

5. Hyperparameters

The following hyperparameters were used during training:

Parameter	Symbol	Value
Learning rate	α	**0.1**

Parameter	Symbol	Value
Discount factor	γ	**0.99**
Initial epsilon	ε_0	**1.0**
Final epsilon	ε_x	**0.1**
Number of episodes	—	**5000**
Maximum steps per episode	—	**200**

Epsilon decays linearly from ε_0 to ε_x over the training episodes.

---

6. Evaluation Procedure

After training, the learned policy was evaluated over **50 test episodes** using a greedy policy ($\varepsilon = 0$). The following metrics were recorded:

- Success rate
- Average number of steps to reach the goal
- Total expanded nodes during training
- Execution time
- Example solution path

---

7. Results

Quantitative Results

Metric	Value
Expanded nodes	**301,388**
Execution time	**13.50 seconds**
Average steps to goal	**8.0**
Success rate	**100%**

Example Solution Path

```
R U L D L D R R
```

This sequence successfully transforms the initial configuration into the goal state.

---

8. Discussion

The Q-Learning agent achieved a **100% success rate**, demonstrating that the learned policy reliably solves the Eight-Puzzle. The average solution length of 8 steps is close to the optimal solution length for the chosen initial state. However, the large number of expanded nodes reflects the inefficiency of tabular methods in large state spaces, as Q-Learning must explore many configurations before converging.

---

9. Conclusion

This experiment demonstrates that tabular Q-Learning can successfully solve the Eight-Puzzle problem when formulated as an MDP within a Gym environment. While effective, the high number of expanded nodes and long training time highlight the limitations of tabular reinforcement learning for combinatorial problems. These limitations motivate the use of function approximation methods such as Deep Q-Networks for improved scalability.

Solving the Eight-Puzzle Problem Using Deep-Q-Network

1. Introduction

The 8-puzzle is a classical planning and search problem consisting of a 3×3 board with eight numbered tiles and one empty space. The objective is to transform a given initial configuration into a predefined goal configuration by sliding tiles into the empty space. In this work, the problem is modeled as a **Markov Decision Process (MDP)** and solved using **Deep Q-Networks (DQN)** within a **Gymnasium-compatible environment**.

---

2. Markov Decision Process (MDP) Formulation

The Eight-Puzzle problem is defined as an MDP $\langle S, A, R, \gamma \rangle$:

State Space (S)

- Each state is a 3×3 grid containing values {0, 1, …, 8}, where `0` represents the empty tile.
- Internally, the state is represented as a NumPy array of shape (3, 3).
- For the DQN, the state is flattened into a **9-dimensional vector** and normalized to [0, 1].

Action Space (A)

A discrete action space with four actions:

- `0`: Move empty tile **Up**
- `1`: Move empty tile **Down**
- `2`: Move empty tile **Left**
- `3`: Move empty tile **Right**

Reward Function (R)

- Valid move: **−1**

- Invalid move (out of bounds): **−5**
- Reaching the goal state: **+100**

This reward structure encourages short solution paths while penalizing illegal actions.

Transition Dynamics

- The environment deterministically updates the puzzle state based on the selected action.
- Invalid moves leave the state unchanged.

Discount Factor (γ)

- γ = **0.99**, emphasizing long-term rewards while still preferring shorter paths.

Episode Termination

An episode terminates when:

- The goal state is reached, or
- A maximum number of steps (200) is exceeded.

---

3. Gymnasium Environment Implementation

The environment is implemented as `EightPuzzleEnv`, inheriting from `gym.Env`.

Observation Space

```
spaces.Box(low=0, high=8, shape=(3, 3), dtype=np.int32)
```

Action Space

```
spaces.Discrete(4)
```

Reset Function

- If an initial state is provided, the environment starts from it.
- Otherwise, the goal state is randomly scrambled using valid moves.

Render Function

- Displays the puzzle state in a human-readable grid format using **_** for the empty tile.

---

4. Deep Q-Network (DQN) Algorithm

Overview

DQN approximates the optimal action-value function Q(s, a) using a neural network instead of a tabular representation. This allows the agent to generalize across unseen states in the large state space of the 8-puzzle.

---

5. Network Architecture Specification

The Q-network is a fully connected feedforward neural network:

Layer	Type	Output Size	Activation
Input	Linear	128	ReLU
Hidden 1	Linear	128	ReLU
Hidden 2	Linear	128	ReLU
Output	Linear	4	None

Details

- **Input dimension:** 9 (flattened puzzle state)
- **Output dimension:** 4 (Q-values for each action)
- **Activation function:** ReLU
- **Optimizer:** Adam
- **Learning rate:** 1e−3
- **Loss function:** Mean Squared Error (MSE)

---

6. Experience Replay and Target Network

Replay Buffer

- Stores transitions (s, a, r, s′, done) with a capacity of 10,000.
- Random mini-batches (size = 64) are sampled to break temporal correlations.

Target Network

- A separate target network is maintained to stabilize training.
- The target network is updated every 500 steps by copying the policy network weights.

---

7. Action Selection (ε-Greedy Policy)

- Initial ε = 1.0
- Final ε = 0.05
- Decay rate = 0.995

This balances exploration and exploitation during training.

---

8. Training Procedure

For each episode:

1. Reset the environment.

2. Select actions using ε-greedy policy.
3. Store transitions in replay buffer.
4. Sample batches and perform gradient descent.
5. Periodically update the target network.
6. Decay ε.

Training was performed for **1000 episodes**.

---

9. Evaluation Metrics

The trained DQN agent was evaluated over 50 test episodes without exploration. The following metrics were collected:

- **Success Rate:** Percentage of episodes reaching the goal.
- **Average Steps to Goal:** Mean number of steps in successful episodes.
- **Expanded Nodes:** Total environment transitions executed.
- **Runtime:** Wall-clock time for evaluation.
- **Path Quality:** Stability, absence of loops, and solution length.

---

10. Results

Quantitative Results

Metric	Value
Success Rate	**100%**
Average Steps to Goal	**8.0**
Shortest Path	8
Longest Path	8
Expanded Nodes	400
Runtime	0.07 seconds

Discussion

- The agent consistently solved the puzzle in all evaluation episodes.
- The solution length is slightly longer than the optimal solution (~6–7 moves) but remains stable and deterministic.
- No oscillatory or redundant behavior was observed.
- The low runtime demonstrates efficient inference.

---

11. Example Solution Paths

Two example solution paths were extracted during evaluation.

Example Path

Moves:

```
R U L D L D R R
```

The visualization shows smooth transitions from the initial configuration to the goal state without unnecessary backtracking.

---

12. Conclusion

This work demonstrates that a Deep Q-Network can successfully solve the Eight-Puzzle problem when formulated as a Gymnasium-compatible MDP. The agent achieves a 100% success rate with consistent, near-optimal solution paths. While classical search algorithms such as A* guarantee optimality for this deterministic domain, DQN provides a flexible learning-based approach that generalizes across states and illustrates the application of deep reinforcement learning to combinatorial puzzles.

1. Comparison of Search Methods and Reinforcement Learning Methods

1.1 Introduction

The Eight-Puzzle problem can be solved using both **classical search algorithms** and **reinforcement learning (RL) methods**. In this project, we implemented and evaluated three approaches:

- **A*** search with two heuristics (Misplaced Tiles and Manhattan Distance)
- **Tabular Q-Learning**
- **Deep Q-Network (DQN)**

This section compares these methods in terms of their assumptions, computational cost, and solution quality, and analyzes their suitability for this problem.

1.2 Example Paths: A* vs Q-Learning Policy

To ensure a fair comparison, all methods were tested on the same initial state:

```
Initial State: (1, 3, 6,
                5, 0, 2,
                4, 7, 8)
```

A* Solution Path (h1 and h2)

Both heuristics produced the same optimal path:

```
R U L D L D R R
```

- Path length: 8
- Optimality: Guaranteed optimal

Q-Learning Solution Path

The learned Q-Learning policy produced the following example path:

```
R U L D L D R R
```

- Path length: 8
- Matches the optimal A* solution

This shows that Q-Learning was able to learn an optimal policy for this specific problem instance.

1.3 Quantitative Comparison of Results

Performance Metrics

Method	Path Length	Expanded Nodes	Execution Time	Success Rate
A* (Misplaced Tiles)	8	17	~0.000 s	100%
A* (Manhattan Distance)	8	13	~0.000 s	100%
Q-Learning	8 (avg)	301,388	13.50 s	100%
DQN	8 (avg)	400	0.07 s	100%

1.4 Comparison of A* with Q-Learning and DQN

1. Need for an Environment Model (Transition Model)

- **A*** requires a **complete and accurate transition model**.
  - It must explicitly know how actions change the state.
  - Successor generation is hard-coded.
- **Q-Learning and DQN** do **not require an explicit model**.
  - They learn solely through interaction with the environment.
  - This makes them model-free methods.

Conclusion: A* is model-dependent, while RL methods are model-free.

---

2. Computational Cost and Number of Visited States

- **A***:
  - Extremely efficient for this problem.

  - Expanded as few as **13 nodes** using Manhattan distance.
  - Execution time was negligible.
- **Q-Learning**:
  - Required **over 300,000 expanded nodes** during training.
  - Training time was significantly longer (13.5 seconds).
- **DQN**:
  - Required far fewer expanded nodes during evaluation.
  - Training is more expensive than A*, but evaluation is fast.
  - More efficient than tabular Q-Learning due to generalization.

Conclusion: A* is vastly more efficient for small, well-defined problems like the Eight-Puzzle.

---

3. Quality of the Obtained Paths (Optimality and Length)

- **A***:
  - Guarantees optimal paths when using admissible heuristics.
  - Always returns the shortest solution.
- **Q-Learning**:
  - Can learn optimal paths, but without guarantees.
  - Convergence depends on exploration and training time.
- **DQN**:
  - Produces near-optimal paths.
  - Does not guarantee optimality due to function approximation.

In this experiment, all methods achieved a **path length of 8**, indicating optimal or near-optimal performance.

1.5 Advantages and Disadvantages Summary

A*

Advantages

- Guaranteed optimal solution
- Very low computational cost
- Deterministic and interpretable
- Ideal for small, discrete problems

Disadvantages

- Requires a known transition model
- Does not generalize to unseen problems
- Not suitable for large or continuous state spaces

Q-Learning

Advantages

- Model-free
- Simple to implement
- Learns optimal policies given enough exploration

Disadvantages

- Large memory requirements
- Slow convergence
- Poor scalability

DQN

Advantages

- Generalizes across states
- Scales to larger state spaces
- Efficient inference once trained

Disadvantages

- More complex
- Training instability
- No optimality guarantee
- Overkill for small problems

2. Comparison of Q-Learning and Deep Q-Network (DQN)

2.1 Overview of the Two Approaches

Q-Learning and Deep Q-Networks (DQN) are both value-based reinforcement learning algorithms that aim to learn an optimal action-value function Q(s, a). However, they differ fundamentally in how this function is represented.

- **Q-Learning** uses a **tabular representation**, storing explicit Q-values for each state-action pair.
- **DQN** replaces the Q-table with a **neural network**, which approximates Q(s, a) and generalizes across states.

In this project, both methods were applied to the Eight-Puzzle problem using the same Gym environment and reward structure.

2.2 Advantages and Disadvantages in This Problem Setting

The Eight-Puzzle has a **relatively small and discrete state space** compared to many modern reinforcement learning problems. This strongly affects the relative performance of the two algorithms.

Q-Learning

Advantages

- Simple and easy to implement
- Guaranteed convergence (under standard assumptions)
- Learns exact Q-values for visited states
- Produces stable and deterministic policies
- No function approximation error

Disadvantages

- Requires storing a Q-value for every visited state
- Does not generalize to unseen states
- Becomes inefficient as the state space grows
- Large number of expanded nodes during training

DQN

Advantages

- Can generalize across similar states
- Does not require storing an explicit Q-table
- Scales better to large or continuous state spaces
- Faster inference once trained

Disadvantages

- More complex implementation
- Requires careful tuning of hyperparameters
- Training instability due to function approximation
- No guarantee of learning the optimal policy
- Higher computational overhead for small problems

2.3 Did DQN Show a Significant Advantage Over Q-Learning?

In this particular problem, **DQN did not show a significant advantage over Q-Learning**.

Both methods:

- Achieved a **100% success rate**
- Produced **near-optimal solution paths** (≈ 8 moves)
- Solved the puzzle reliably

However:

- Q-Learning required **significantly more expanded nodes** during training.
- DQN converged to a stable policy more quickly and executed solutions faster during evaluation.

Despite this, the **final policy quality was similar**, and the additional complexity of DQN did not result in substantially better performance.

Reason: The Eight-Puzzle has a **small, fully observable, discrete state space**, which is ideal for tabular methods. In such settings, Q-Learning is sufficient and often preferable.

2.4 When Would DQN Outperform Q-Learning?

DQN is expected to perform significantly better than Q-Learning in problems with:

- Very **large state spaces**
- **Continuous or high-dimensional observations**
- **Image-based inputs** (e.g., Atari games)
- Environments where **generalization** is required
- Tasks where storing a Q-table is infeasible

Examples include:

- Robotic control
- Autonomous driving
- Video game environments
- Continuous control tasks

In these cases, tabular Q-Learning becomes impractical, while DQN can learn useful representations using neural networks.

2.5 Final Method Selection

If only **one method** were to be chosen for solving the Eight-Puzzle problem, **Q-Learning** would be the preferred choice.

Justification:

- The state space is small and discrete

- Q-Learning is simpler and more transparent
- It provides stable convergence and exact value estimates
- No need for neural networks or replay buffers
- Lower implementation and debugging complexity

DQN, while powerful, introduces unnecessary complexity for this specific task and does not offer a meaningful performance advantage.

2.6 Summary Table

Criterion	Q-Learning	DQN
State space suitability	Small, discrete	Large, high-dimensional
Implementation complexity	Low	High
Memory usage	High (Q-table)	Low (network weights)
Generalization	None	Yes
Training stability	High	Moderate
Optimality guarantee	Yes (tabular)	No
Best choice for 8-Puzzle	✅	❌

Conclusion

For small, deterministic problems like the Eight-Puzzle, tabular Q-Learning remains a strong and reliable choice. DQN becomes advantageous only when the problem scale exceeds the limits of tabular methods.

3. Overall Conclusion and Method Selection

If only **one method** had to be chosen to solve the Eight-Puzzle problem, **A*** would be the preferred choice.

Reasoning

- The problem has a **small, discrete, fully known state space**
- An admissible heuristic (Manhattan distance) is available
- A* finds the **optimal solution with minimal computation**
- Reinforcement learning methods require significantly more computation to reach the same result

Final Assessment

- **A*** is the best practical solution for this problem.
- **Q-Learning** demonstrates learning capability but is inefficient.
- **DQN** is powerful but unnecessary for such a small domain.

RL methods become advantageous only when:

- The transition model is unknown
- The state space is very large or continuous
- Generalization across many tasks is required

Final Remark

This comparison highlights that the choice of algorithm should be guided by the **problem structure**, not just algorithmic sophistication. While reinforcement learning is powerful, classical search methods remain superior for small, deterministic planning problems such as the Eight-Puzzle.