Computers & Industrial Engineering 171 (2022) 108452

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

Q-learning and traditional methods on solving the pocket Rubik's cube

Zefeng Lyu, Zeyu Liu, Anahita Khojandi, Andrew Junfang Yu *

Department of Industrial and Systems Engineering, The University of Tennessee, Knoxville, Knoxville, TN 37996, USA

ARTICLE INFO

Keywords:
Pocket Rubik's cube
Reinforcement learning
Q-learning
Breadth-first search
Linear programming
Dijkstra's algorithm

ABSTRACT

Rubik's cube is a famous mathematical puzzle invented by Ern Rubik in 1974. The goal of the puzzle is to restore the cube from a scrambled state to its original state, where each face is occupied by a single color. Human players focus on solving the Rubik's cube in the shortest time using memorized algorithms, while researchers are more interested in the optimal solution, i.e., the fewest moves to solve the cube. Recently, reinforcement learning methods have been widely used to solve mathematical puzzles, including the Rubik's cube. However, the benefits and drawbacks of using a learning method are unclear. This paper fills this gap by comprehensively comparing a reinforcement learning method with three traditional methods, namely breadth-first search, linear programming, and Dijkstra's algorithm, for solving the pocket Rubik's cube. Computational results show that Q-learning with fine-tuned parameters learns to solve the problem after 7.3 hours of training. In contrast, two traditional methods are found to be more efficient in solving the pocket cube. The proposed breadth-first search algorithm solves all possible states in 2 minutes with optimality guaranteed. The linear programming formulation solves all states in 12 minutes. In conclusion, the advantage of Q-learning is that it does not require domain knowledge and can scale to solve larger problems. However, traditional methods work better for solving the pocket Rubik's cube.

1. Introduction

The pocket Rubik's cube is a 2x2x2 variant of the Rubik's cube, invented by Hungarian sculptor Ern Rubik. This paper refers to the pocket Rubik's cube as the pocket cube or cube for simplicity. A pocket cube is comprised of eight small cubelets and has six faces. Each face can be rotated independently to change the pocket cube from one state to another. The goal of the puzzle is to restore the cube from a scrambled state to its original state, where a single color occupies each face. Using memorized algorithms, human players attempt to solve the cube in the shortest time. However, researchers are more interested in the solution that requires the minimum number of moves. Fig. 1 shows a pocket cube in its goal state and the rotation dynamics of the cube.

There are two metrics to count the number of moves used in the Rubik's cube community: the half-turn metric (HTM) and the quarter-turn metric (QTM). In HTM, one move refers to rotations that turn a face by any degree. In QTM, one move only refers to rotations that rotate a face by 90 degrees, either clockwise or counterclockwise. Without further explanation, all the moves in this paper follow the QTM criteria. The total number of distinct states of the Rubik's cube is approximately 4.33×10^{19}. In comparison, the pocket cube only contains 3,674,160 states, making it much easier to solve. Note that two states are regarded as the same state if they differ only in spatial placement. For instance, rotating the whole cube at any degree does not change the state of the cube. In addition, two moves are equivalent if they function the same. For example, rotating the upper level of a cube clockwise is identical to rotating the bottom level of that cube counterclockwise.

The distance of a state refers to the minimum number of moves required to restore a pocket cube from that state to the goal state. Table 1 shows the distribution of the distances for all distinct states. As is shown, the majority of states have a distance distributed between 9 and 12, accounting for 93 % of the cases. In addition, 14 moves are sufficient to restore any scrambled pocket cube to the goal state. It is worth mentioning that fourteen moves are known in the Rubik's cube community as God's number, which represents the minimum number of moves guaranteed to solve any scrambled cube.

Recently, reinforcement learning methods have been widely used to solve mathematical puzzles such as the Rubik's cube. However, the benefits and drawbacks of using a learning-based method are unclear. This paper aims to fill this gap by comprehensively comparing a reinforcement learning method with three traditional methods, namely breadth-first search, linear programming, and Dijkstra's algorithm, for solving the pocket Rubik's cube. The challenge of applying Q-learning is that the learned policy may not be generalized to other untrained states.

* Corresponding author.
E-mail address: ajyu@utk.edu (A. Junfang Yu).

https://doi.org/10.1016/j.cie.2022.108452
Received 26 December 2021; Received in revised form 8 June 2022; Accepted 10 July 2022
Available online 19 July 2022

We address this issue by repeatedly starting the training process from different states to make sure that the Markov Decision Process (MDP) environment is fully explored. In addition, the pocket cube has very sparse rewards and no guarantee of termination. The Q-agent is rewarded only when the goal state is reached. Another challenge is that the number of iterations for termination is unknown in advance. This issue can be alleviated by setting up several sub-goals or gradually increasing the difficulty of the initial states.

The contributions of this paper are threefold. First, we propose a reinforcement learning method and three traditional methods to solve the pocket cube. To the best knowledge of the authors, this is the first work to formulate and solve the pocket cube as a network flow problem. Second, several new metrics are proposed to evaluate the learned policy and the training progress. These metrics and the computational results present in this paper can be used as benchmarks for future comparisons. Third, it is found that both the breadth-first search (BFS) algorithm and the linear programming (LP) method outperform Q-learning in terms of computing time. The advantage of Q-learning is that it does not require domain knowledge and can be extended to solve large problems.

The remainder of this paper is structured as follows: Section 2 reviews the related literature. Section 3 introduces the Q-learning algorithm covering the encoder for the pocket cube, the MDP environment, and the Q-table update rules. Section 4 presents the mathematical formulation for the LP method and the pseudocode for the breadth-first search algorithm and Dijkstra's algorithm. Computational experiments for Q-learning and the traditional methods are conducted in Section 5. Finally, observations, conclusions, and future research are discussed in Section 6.

2. Literature review

The first popular research topic about Rubik's cube is determining the so-called God's number, which refers to the fewest moves required to restore an arbitrary scrambled Rubik's cube. Morwen Thistlethwaite developed the first group-theory-based algorithm for solving the Rubik's cube. In 1992, Herbert Kociemba improved Thistlethwaite's algorithm by reducing the number of subgroups to two. Korf (1997) later presented an algorithm to search for the shortest path based on IDA* and pattern databases. Nowadays, Kociemba's algorithm is one of the most widely used algorithms for solving the Rubik's cube. Using Kociemba's algorithm, Rokicki et al. (2013) decreased the upper bound of God's number to twenty. By proposing the state called "superflip," Michael Reid proved that the lower bound of Rubik's cube is twenty. As a result, the God number is finally determined to be twenty. Other literature in terms of determining God's number includes Kunkle & Cooperman (2007), Rokicki (2010), and Rokicki, Kociemba, Davidson, & Dethridge (2013).

Table 1
The number and percentage of different states for a pocket cube.

Distance	Count	Percentage	Distance	Count	Percentage
0	1	0.00 %	8	114,149	3.11 %
1	6	0.00 %	9	360,508	9.81 %
2	27	0.00 %	10	930,588	25.33 %
3	120	0.00 %	11	1,350,852	36.77 %
4	534	0.01 %	12	782,536	21.30 %
5	2,256	0.06 %	13	90,280	2.46 %
6	8,969	0.24 %	14	276	0.01 %
7	33,058	0.90 %	Total	3,674,160	0.00 %

After the breakthrough of using deep reinforcement learning (DRL) to solve the chess game Go, the learning approaches are widely applied to solve mathematical puzzles, including Rubik's cube. Smith, Kelly, & Heywood (2016) presented the SARSA $\in$ - greedy value approximation algorithm. Agostinelli, McAleer, Shmakov, & Baldi (2019) introduced the DeepCubeA, a learning-based searching algorithm that combines deep reinforcement learning (DRL) and weighted A* search. DeepCubeA achieved to restore scrambled cubes 60.3 % of the time in the shortest path. Apart from the optimal solutions, 36.4 % and 3.3 % of the solutions require two and four extra moves, respectively. Other popular research includes how to solve the Rubik's cube using a robot hand. Relevant research can be found in Toshniwal (2019), Varga, Durovsky, & Kovac (2014), and Dan, Harja, & Nașcu (2021).

However, limited attention has been paid to solving the pocket cube, either using traditional methods or reinforcement learning. Bowman, Guo, & Jones (2018) presented the autodidactic iteration (ADI) algorithm to solve the pocket cube. They solved pocket cubes with initial states that are at most four moves away from the solved state. According to Table 1, the number of states they solved is 688. By comparison, the proposed Q-learning method in this paper solves all 3,674,160 states. Hukmani, Kolekar, & Vobugari (2021) propose a parallel Q-learning algorithm to solve the pocket cube. The Q-agent is trained to achieve a success rate of 99.8 % with 6 hours of training. The method needed an average of 21 moves to solve the pocket cube. Compared with them, the proposed algorithm achieves a better success rate with a smaller number of moves needed.

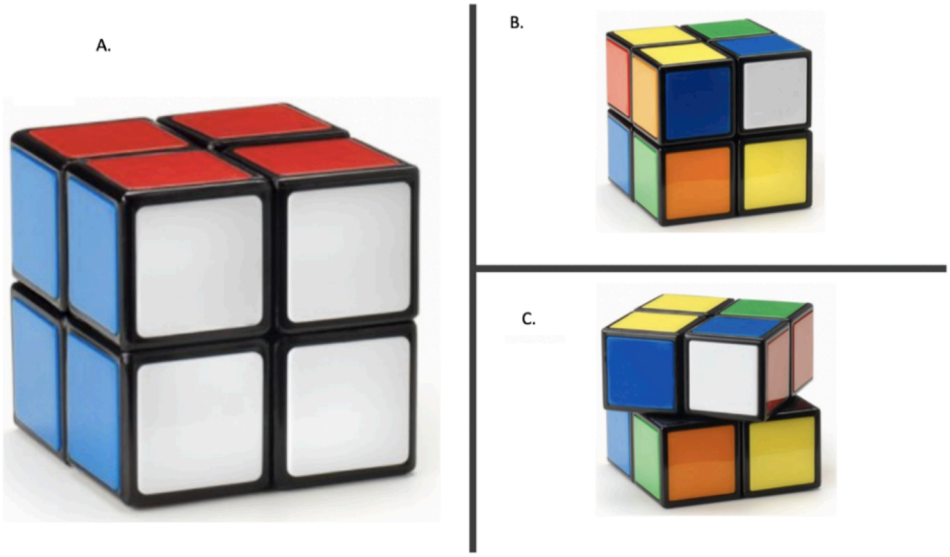

Fig. 1. A. The goal state of a pocket cube. B. A randomly scrambled pocket cube. C. The rotation dynamic of a pocket cube that illustrates a quarter-turn move.

3. Q-learning

As a fully off-policy temporal difference control algorithm, Q-learning was presented by Watkins & Dayan (1992). The terminology Q-learning is derived from the goal of learning Q-value, a state-action pair that indicates the benefit of taking a particular action in a specific state. The advantage of Q-learning is that it does not require any domain knowledge to solve a problem. After establishing the MDP environment, Q-agent can learn the optimal strategy by exploring the environment. This section begins with a discussion of the encoder for the states. Then, the MDP environment is established. Finally, we discuss the rules for updating the Q-values.

3.1. State encoder

The states are encoded by the color-orientation encoder (COE). Specifically, each cubelet is encoded by a color-orientation pair (i, j), where i represents the color, and j represents the orientation. As is shown in Fig. 2, there are twenty-four such pairs, consisting of eight color patterns and three orientations. For example, $i = 0$ denotes the color pattern of yellow-blue-red, and $j = 0$ denotes the orientation shown in the first cubelet in the first row of the figure.

The pocket cube consists of eight subspaces. As illustrated in Fig. 3, these subspaces are indexed as $C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7$. Each subspace is then occupied by a cubelet encoded by the COE encoder. Taking the goal state as an example, it is encoded as ((0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0)).

3.2. Markov decision process

The first step of Q-learning is to model the MDP environment. The decisions are made by the Q-agent at every epoch, but the total number of epochs needed is unknown in advance. The variable S denotes the states. We should exercise caution about duplicate states and unreachable states. For example, rotating the pocket cube as a whole does not change the state. To address this issue, we fix the upper left corner of the pocket cube, as shown in Fig. 4. In addition, several states are not reachable because the cubelets on the same face are restricted from rotating together. Golomb (2014) proposed a practical way to remove these unreachable states. They pointed out that the states are not reachable if the summation of the index j cannot be divided by 3. After removing the duplicate states and unreachable states, the total number of unique states is as follows.

$$\text{Number of states} = \frac{8! \cdot 3^7}{24} = 3,674,160.$$

Fig. 4 shows the actions that can be taken by the Q-agent. As is indicated in the right subfigure, notations R, D, and B represent the actions that rotate the right face, bottom face, and back face by 90 degrees clockwise, respectively. Notations R', D', and B' denote the corresponding counterclockwise rotations. The actions are consistent across all states and epochs.

The pocket cube transitions from one state to another state after taking an action. Fig. 5 shows an example of transitions for the goal state. The left subfigure shows the goal state and its six neighboring states. The right subfigure shows the transitions for the goal state. For example, the pocket cube transmits from the goal state S_0 to state S_1 by performing the action R. This transmission is reversible. The cube can transmit back to the goal state by performing action R'.

A reward should be assigned to the Q-agent after every action it takes. The reward can be positive to encourage the action or negative to discourage the action. This paper rewards the Q-agent by 1000 if it restores the pocket cube to its goal state. According to our preliminary experiments, the value of the final reward, also called the winning reward, does not affect the training process. Additionally, we expect the Q-agent to restore the pocket cube in the fewest possible moves. To achieve that, we penalize the Q-agent by one for each move used. The number of moves used to restore the pocket cube can be determined by

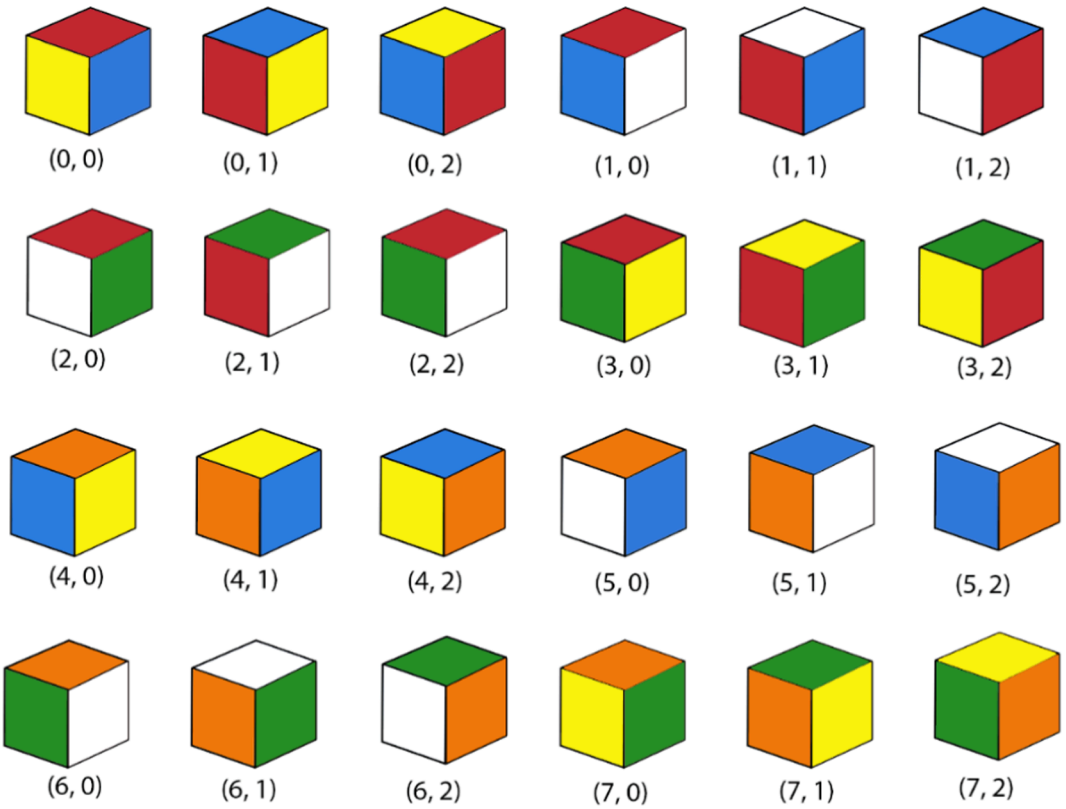

Fig. 2. Twenty-four color-orientation pairs.

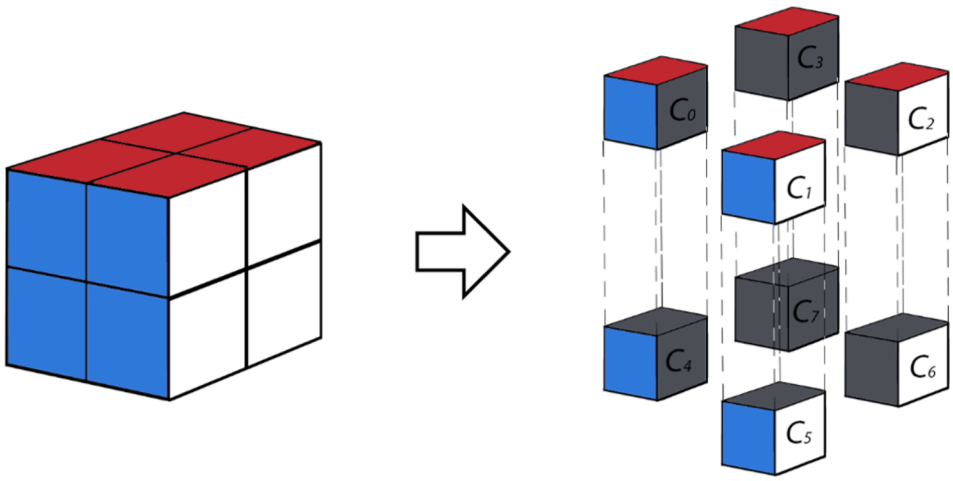

Fig. 3. The left figure shows a pocket cube in its goal condition. The right figure illustrates the eight cubelets of the pocket cube. The greyed-out face indicates that it is not visible.

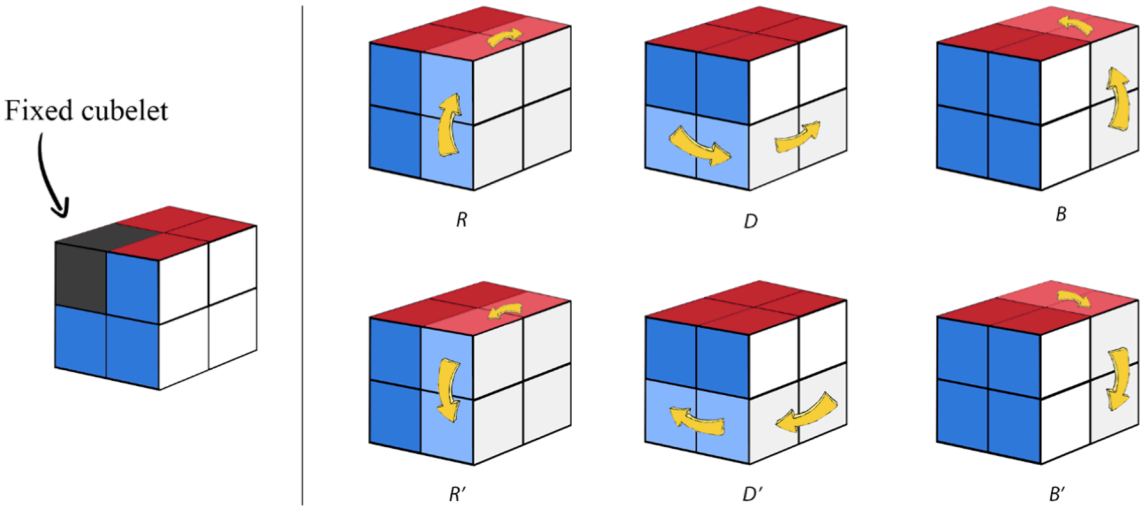

Fig. 4. In the left figure, the cubelet highlighted in grey is fixed. Its position and orientation remain constant regardless of the actions conducted. The right figure shows the actions.

adding up the penalties.

3.3. Q-values update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[r_t + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{1}$$

The rule for updating the Q-values is shown in Equation (1). $Q(S_t, A_t)$ refers to the Q-value at the current epoch. It presents the utility of performing action A_t at the state S_t. Bellman established that Q-values converge to a constant value over time. There are two parameters to determine in the equation, namely the learning rate α and discount factor γ.

The optimal learning rate for a completely deterministic MDP has been proved to be one. The pocket cube is such a problem as we can predict the next state knowing the current state and the action to take. As a result, we set the learning rate $\alpha = 1$.

The discount factor establishes the significance of future value. A greater γ would cause the agent to prioritize the long-term payoff. Conversely, a smaller γ makes the agent focus more on short-term gains. The discount factor should be less or equal to one for infinite MDPs. Otherwise, Q-values cannot converge. Although the Q-agent does not know how many moves it will take to restore the pocket cube, our problem is not to deal with an infinite MDP. Through a series of experiments, it is found that the Q-agent requires a significant number of moves to restore the cube during the early stages. However, regardless of the beginning situation, the Q-agent will always be able to discover a feasible solution. Additionally, it is observed that the Q-agent performs best when we do not discount the future reward. Therefore, the discount factor is set as $\gamma = 1$. As a result, Equation (1) can be simplified to Equation (2).

$$Q(S_t, A_t) \leftarrow r_t + \max_{a \in A} Q(S_{t+1}, a) \tag{2}$$

4. Traditional methods

This section presents three traditional methods to solve the pocket cube problem. The problem can be formulated as a network flow problem. To achieve that, we need to create a graph $G = (V, A)$ where V denotes the states and A denotes the arcs connecting two neighboring states. The length of the arcs is one. Then, restoring a pocket cube with the fewest possible moves is identical to finding the shortest path in the graph from the initial state to the goal state. The rest of this section introduces the three traditional methods in detail. To begin with, we

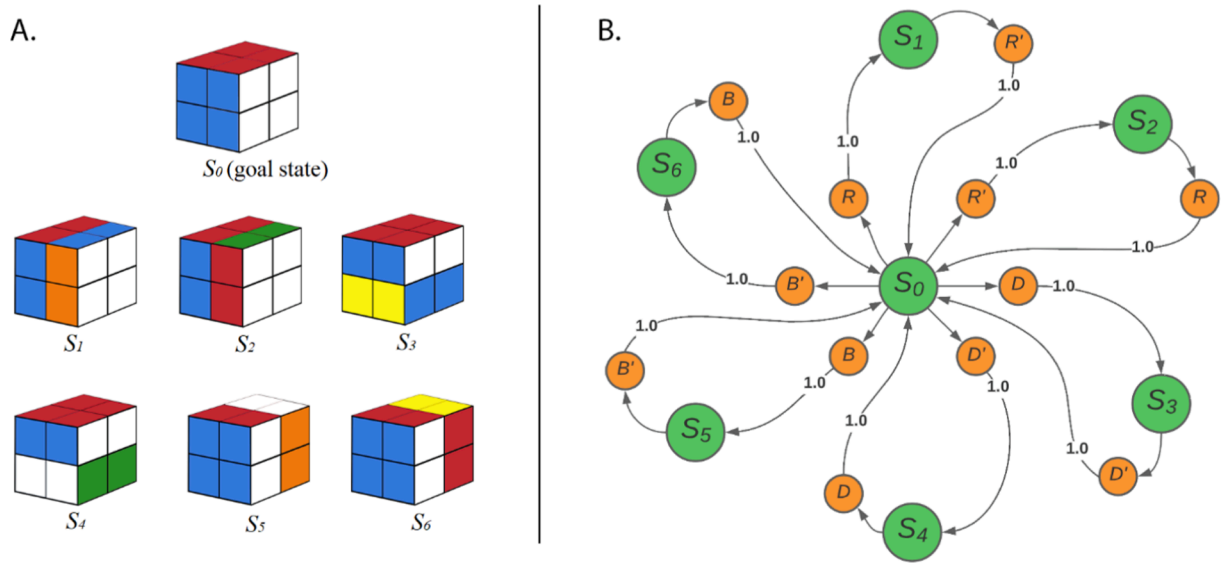

Fig. 5. A. This subfigure shows the goal state and its six neighborhood states that are reachable by one move. B. The subfigure shows the transitions for the goal state. The green circles denote the states. The orange circles denote the actions. The transition probability is 1.0. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

propose a linear programming (LP) formulation. Secondly, we present the breadth-first search (BFS) algorithm. Thirdly, the Dijkstra's algorithm is implemented. Related approaches can be found in Bangyal et al. (2022), Bangyal et al. (2021), and Bangyal, Ahmed, & Rauf (2020).

4.1. LP formulation

The LP formulation is a well-known model for solving network flow problems. However, to the best of our knowledge, this is the first work using it to solve the pocket cube problem. The general idea is as follows. First of all, $n-1$ units of resource flow out from the source node to the other nodes, where n is the number of nodes. Then, each node receiving the resources will retain one unit of resource and distribute the remainder to the other nodes. The objective is to minimize the overall flow. The LP model is formulated as follows.

$$min \sum_{(i,j)\in A} x_{ij} \tag{3}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} n-1, i = s \\ -1, \forall i \in N \backslash \{s\} \end{cases} \tag{4}$$

$$x_{ij} \in \{0, 1, \cdots\}, \forall (i,j) \in A \tag{5}$$

Equation (3) is the objective function, which aims to minimize the total flow presented in the graph. Equation (4) ensures that the outbound flow minus the inbound flow equals one for all nodes except for the source node, where inbound flow is equal to zero and outbound flow is equal to $n-1$. Equation (5) restricts that the decision variables must be non-negative integers. After solving the LP formulation, the shortest paths from all nodes to the source can be interpreted by selecting the neighbors with the highest flow as the predecessor.

4.2. Breadth-first search

The general idea of the BFS algorithm is to find the shortest paths for states, layer by layer. The states that directly connect to the goal state are called states in the first layer. Obviously, the shortest paths to the goal state of these states are equal to one. Then, the algorithm searches for states that directly connect to the states in the first layer. These newly discovered states are called states in the second layer. The augmentation process terminates until all the states have been evaluated.

Note that states that have been evaluated should be excluded from the following search. However, it is not necessary to create a look-up table to record all previously evaluated states. According to the features of the pocket cube, the neighbors of a state can be either one step further or one step closer to that state. Therefore, we only need to check the last layer to determine whether the new states have already been evaluated. The pseudocode of the BFS algorithm is shown as follows.

Algorithm 1 Breadth-first Search

```
// Initialization
set the number of visited state n = 0
set the number of total states N = 3,674,160
create LAST to record the states in the last layer
create CRET to record states in the current layer
create NEXT to record states in the next layer
add source node to CRET
layer = 0

// Main loop
while n < N
  for i in CRET
    dist[i] = layer
    n++
    for j in neighbors of state i
      if j not in LAST nor in NEXT
        add state j to NEXT

// Move to the next layer
LAST = CRET
CRET = NEXT
NEXT = {}
layer ++

return dist
```

4.3. Dijkstra's algorithm

Dijkstra's algorithm is a well-known approach for finding the shortest path from a source node to all other nodes in a graph, which matches the pocket cube problem perfectly, i.e., finding the shortest path from the goal state to all the other states. The idea of the original Dijkstra's algorithm is as follows. All nodes start with an initial label with a distance of infinity. With the goal state as the first selected node,

in each iteration, one unvisited node with the shortest distance to the last selected node is selected. Then, the distance between the selected node and its unvisited neighbors is calculated. The labels of the neighbors are updated if their new distances are smaller than the current distances.

The original Dijkstra's algorithm maintains a look-up table for all unvisited nodes. Our preliminary experiments indicate that most of the computational time is spent on selecting the next node with the smallest label. The selection process is time-consuming because the number of unvisited nodes is enormous. However, most of the unvisited nodes have labels of infinity. It is not necessary to take these nodes into consideration. Therefore, we modify Dijkstra's algorithm by maintaining an empty subset to record the unvisited nodes and expanding the subset dynamically. As a result, the computational time is reduced because only a small subset of the unvisited nodes is evaluated in each iteration.

The pseudocode of Dijkstra's algorithm is shown in Algorithm 2. S denotes the set of visited nodes and $\overline{S}$ denotes the set of unvisited nodes. s represents the goal state. *dist* refers to the distance between a current state and the goal state. *pred* refers to the predecessor of a node. Notation $|\cdot|$ counts the number of elements inside the set.

Algorithm 2 Dijkstra's Algorithm

```
//Initialization
S = {}
S̄ = {s}
dist[s] = 0
pred[s] = 0
for node ν in V
    dist[ν] = +∞
    pred[ν] = NONE

//Main loop
while |S̄| < |V|
    find node i with the smallest dist in S̄
    for j in neighbors of node i
        Add node j into S̄ if the node is not visited
        if dist[j]⟩dist[i] + 1
            dist[j] = dist[i] + 1
            pred[j] = i
    Add node i into the visited set S and remove i from the unvisited set S̄

return dist, pred
```

5. Computational experiments

This section conducts computational experiments for Q-learning and the traditional methods. New metrics for evaluating the learned policy and the training process are defined. Additionally, the parameters for Q-learning are fine-tuned. The proposed algorithms are compared with each other and with the best-known benchmark in the literature. The proposed algorithms are coded in Python and executed on an x64-based workstation with an Intel Core i7-3770 processor. The workstation runs Microsoft Windows 10 Pro and operates at 3.40 GHz with 32 GB of RAM. The LP formulation is solved by the commercial solver Gurobi with version 8.1.0.

5.1. Results of Q-learning

Training on one single initial state.

Fourteen preliminary experiments are conducted to evaluate Q-learning when the Q-agent is trained with a single initial state. The initial states are fixed for each individual experiment but vary between experiments. The distance of the initial states to the goal state ranges from one to fourteen as shown in Table 2. Each experiment is trained for 500 episodes with parameters $\in = 0.1, \gamma = 1, \alpha = 1$. We use a relatively small $\in$ for the preliminary experiments to reduce the training time. Parameter fine-tuning will be discussed in a subsequent section.

The results of the preliminary experiments are shown in Fig. 6, which depicts the relationship between episodes and rewards throughout the training process. The reward is equal to the winning reward minus the number of moves used to solve the pocket cube. The larger the reward, the fewer moves are needed to restore the pocket cube. It is found that the rewards gradually converge to some large values, demonstrating that Q-learning learns to solve the pocket cube with fewer and fewer moves.

Table 2
Initial states for the fourteen preliminary experiments.

Initial states encoded by the COE encoder	Distance to the goal state
((0, 0), (1, 0), (2, 0), (3, 0), (5, 0), (6, 0), (7, 0), (4, 0))	1
((0, 0), (1, 0), (2, 0), (3, 0), (6, 0), (7, 0), (4, 0), (5, 0))	2
((0, 0), (1, 0), (2, 0), (4, 1), (5, 0), (7, 1), (6, 0), (3, 1))	3
((0, 0), (1, 0), (2, 0), (4, 1), (7, 1), (6, 0), (3, 1), (5, 0))	4
((0, 0), (1, 0), (2, 0), (4, 1), (7, 2), (6, 0), (5, 2), (3, 1))	5
((0, 0), (1, 0), (2, 0), (5, 2), (3, 2), (4, 0), (6, 1), (7, 1))	6
((0, 0), (1, 0), (2, 0), (5, 1), (3, 1), (4, 2), (7, 0), (6, 2))	7
((0, 0), (1, 0), (2, 0), (5, 0), (3, 1), (6, 2), (7, 0), (4, 0))	8
((0, 0), (1, 0), (2, 0), (5, 0), (3, 0), (7, 0), (6, 0), (4, 0))	9
((0, 0), (1, 0), (2, 0), (5, 0), (4, 0), (3, 0), (7, 0), (6, 0))	10
((0, 0), (1, 0), (2, 0), (5, 0), (4, 0), (6, 0), (7, 2), (3, 1))	11
((0, 0), (1, 0), (2, 0), (5, 0), (4, 0), (7, 0), (6, 0), (3, 0))	12
((0, 0), (1, 0), (2, 1), (5, 1), (6, 2), (3, 2), (7, 0), (4, 0))	13
((0, 0), (1, 1), (2, 0), (6, 1), (3, 1), (5, 0), (4, 1), (7, 2))	14

An issue of this approach is that the learned policy does not generalize to other states that have not been trained. To overcome this issue, we launch the training process from multiple initial states until the entire graph G is explored. Note that it is unnecessary to exhaust all states because the intermediate states are trained simultaneously in every single episode. The challenge is to train the Q-table thoughtfully and efficiently. Furthermore, how to evaluate the Q-table is another question to be answered as the initial states change over time.

Training on multiple initial states.

As stated in the previous section, the learned policy cannot be generalized to other states if the model is trained from a single initial state. To make the learned policy work universally, the Q-agent must explore the entire environment and update the Q-table for every state-action pair. Therefore, we restart the learning process from multiple initial states. This paper generates the initial states by shuffling the pocket cube from the goal state enough times. During the experiments, it is found that gradually increasing the difficulty of the initial states by increasing the number of shuffles can slightly reduce the training time, particularly for the early training period. This paper does not utilize this feature because the improvement is not very significant. Instead, we shuffle the pocket cube by 100 moves to generate the initial states because this number of moves is enough to ensure that the initial states are sufficiently far away from the goal state.

The reward can be used to evaluate the learned policy, but it only works for evaluating the performance of one state, not the overall performance of all states. Therefore, we propose two new metrics, namely "success rate" and "unnecessary moves." The success rate refers to the fraction of states that can be restored to the total number of states under the guidance of the learned policy. Restoring the pocket cube with a learned policy is called replay. We set a limit of 100 moves for each replay to ensure program termination. Note that all states need to be checked to calculate the percentage of success. For these states that can be restored successfully, we also define the metric, unnecessary moves, to measure the difference between the moves used by Q-learning and the optimal moves. Note that the optimal moves are obtained by using the exact methods introduced in Section 4. The calculations for the success rate and unnecessary moves are shown as follows.

$$\text{Success rate} = \frac{\text{Number of states that are solved}}{\text{Number of all initial states}},$$

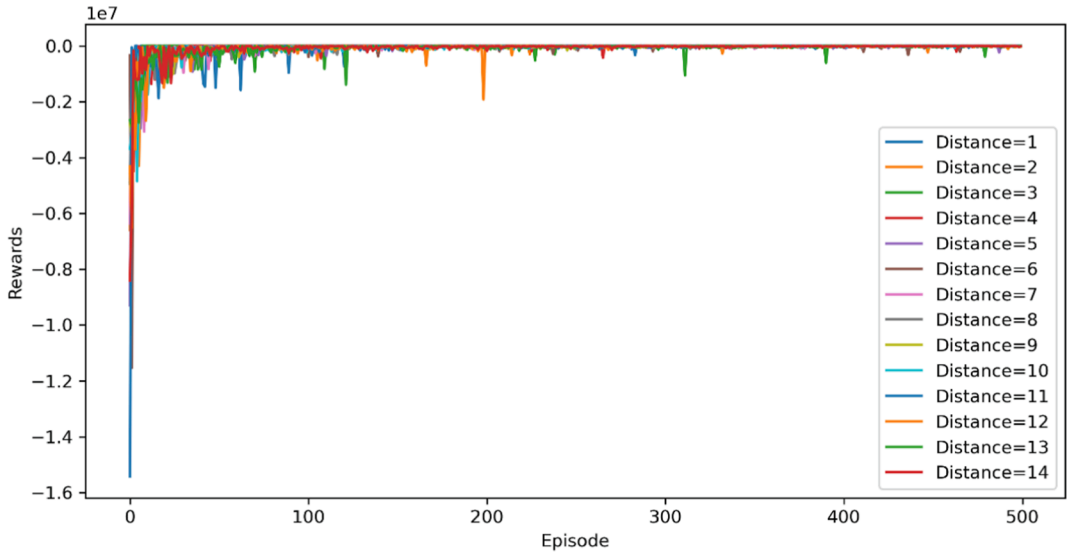

Fig. 6. Q-learning performance on fourteen preliminary experiments.

$$\text{Unnecessary moves} = \frac{\text{Moves used by the proposed algorithm} - \text{Optimal moves}}{\text{Number of states that are solved}}.$$

Fig. 7 shows the performance of the learned policy by using the success rate and unnecessary moves. The results indicate that the learned policy performs better in terms of the two proposed metrics when a larger exploration factor is used. When the exploration factor is increased to 0.6, the success rate approaches 100 %, indicating that Q-learning has learned to solve the pocket cube from any initial state. The corresponding unnecessary moves show that the average additional move is 0.79 compared with the optimal moves. If we further increase the exploration factor to 0.8, the unnecessary moves can be reduced to zero, indicating that the Q-agent has learned to solve the pocket cube in an optimal way. However, it is worth mentioning that the larger the exploration, the longer the training time is needed. Therefore, we conduct several parameter tunings in the following section to achieve a good trade-off between performance and efficiency.

Parameter fine-tuning.

This section fine-tunes the exploration factor, discount factor, and learning rate. First, we need to define a new metric, Q-score, to evaluate the training process, as the two metrics defined in the previous section can only evaluate the learned policy. The Q-score measures the gap between the current Q-table and the optimal Q-table. Specifically, it is equal to the portion of states where the predicted action matches the optimal action. The initial Q-score is 0.5, indicating that the Q-agent makes correct decisions for half of the states without training. The Q-score converges to its upper bound during the training episodes, which is 1 when all state-action pairs converge to their optimal values. The equation for calculating the Q-score is shown as follows.

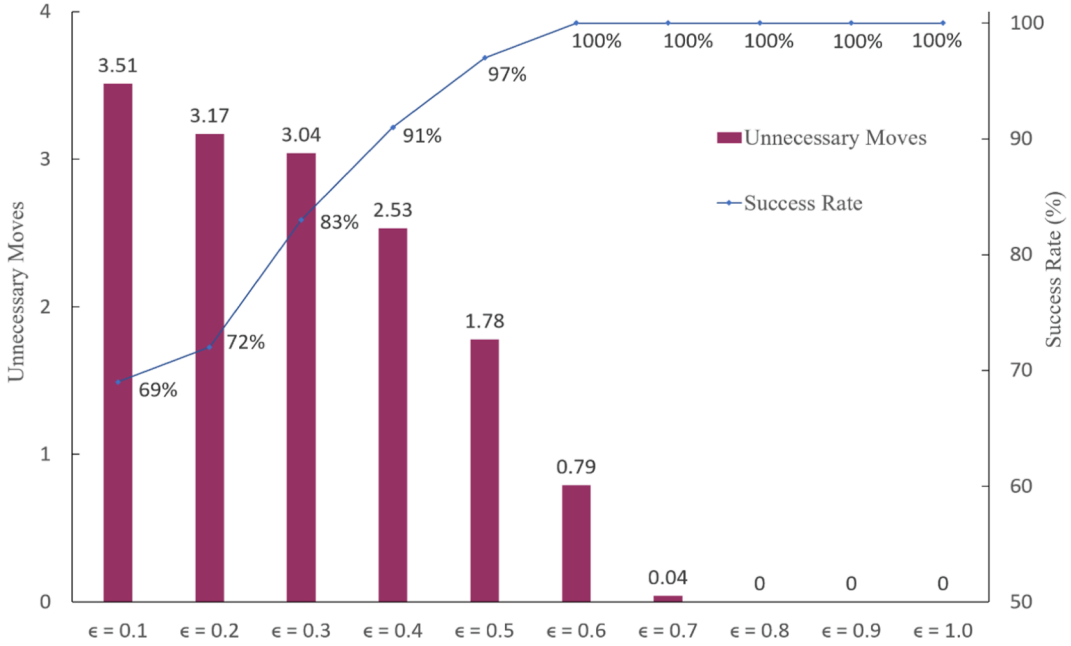

Fig. 7. The performance of the learned policy when different exploration factors are used. Each experiment is trained for 100,000 episodes with $\gamma = 1$, $\alpha = 1$.

$$\mathrm{Q-score} = \frac{\text{The number of states that the predicted action matches the optimal action}}{\text{Total number of states}}$$

Fine-tuning the exploration factor is essential because its value maintains the trade-off between exploitation and exploration. If the exploration factor is set to a small value, Q-learning prefers to exploit the best-known policy. As a result, the training process for each episode would be short, but the improvement for each episode is also limited. In contrast, if the exploration factor is set to a large value, Q-learning prefers to conduct more explorations, which would increase the training time per episode because exploring new solutions can be very time-consuming. Therefore, it is crucial to maintain a good compromise between exploitation and exploration to train the Q-learning model efficiently.

Fig. 8 shows the change in Q-scores over 500 training episodes using different exploration factors. The discount factor and learning rate are set as $\gamma = 1$ and $\alpha = 1$, respectively. It is found that the Q-score increases faster when a large exploration factor is used. However, the training time also increases significantly at the same time. For instance, the final Q-score is 0.52 while the training time is 51 minutes when the exploration factor is equal to 0.8. By contrast, if the exploration factor is increased to 0.9, the final Q-score increases to 0.72 while the training time is increased to 88 minutes. Finally, if the exploration factor is further increased to 1, the Q-score approaches 1 in only around 100 episodes. However, it takes more than 15 hours to complete the 500 training episodes.

Fig. 9 illustrates the correlation between training time and Q-score when different exploration factors are used. The Q-score improves very slowly during the first 40 minutes of training. The reason may be that the best-known policy is ineffective in the early stages, causing the Q-agent to make numerous moves to reach the goal state. After the early period, the Q-agent acquires more knowledge about solving the pocket cube. Thus, the training process is subsequently accelerated. In addition, we observe that the Q-agent performs better with a relatively large exploration factor. For example, the final Q-score is smaller than 0.6 when the exploration factor is smaller than 0.3. By contrast, the final Q-score is greater than 0.88 when the exploration factor is greater than 0.8. Note that it is not ideal to set the exploration factor too high. For example, if we set the exploration factor to 1, the Q-agent would completely disregard the learned policy and focus solely on exploring new policies, thereby decreasing the training efficiency. According to our experiments, $\epsilon = 0.9$ is the best compromise between exploitation and exploration. This value is also employed to fine-tune the discount factor and learning rate.

Fig. 10 illustrates the correlation between training time and Q-score when different discount factors are used. The function of the discount factor is to control the significance of future rewards. The Q-agent would prioritize long-term rewards if the discount factor is set to a large value. In contrast, a smaller discount factor would cause the Q-agent to prioritize immediate rewards. The discount factor cannot exceed one to ensure that the training process converges to the optimal policy. According to the figure, the final Q-score increases as the discount factor increases. The best discount factor for solving the pocket cube problem is $\gamma = 1$.

Fig. 11 illustrates the correlation between training time and Q-score when different learning rates are used. The function of the learning rate is to control the speed at which the Q-agent learns. As described in Section 3, the optimal learning rate for a completely deterministic MDP is 1. The pocket cube is a deterministic problem as the next state can be determined given the current state and an action to be taken. Consequently, $\alpha = 1$ should be the best learning rate. This assumption is supported by our experiments. Compared to other learning rates, the performance of Q-learning is significantly better when the learning rate is one.

In conclusion, the best parameters for solving the pocket cube problem are $\epsilon = 0.9, \alpha = 1, \text{and} \gamma = 1$. An additional experiment is conducted with the fine-tuned parameters. It is observed that an optimal Q-table can be trained in 7.3 hours.

5.2. Results of traditional methods

The traditional methods are compared with Q-learning and the best-known benchmark in this section. Table 3 shows the computing time, success rate, and unnecessary moves of the methods compared. Note

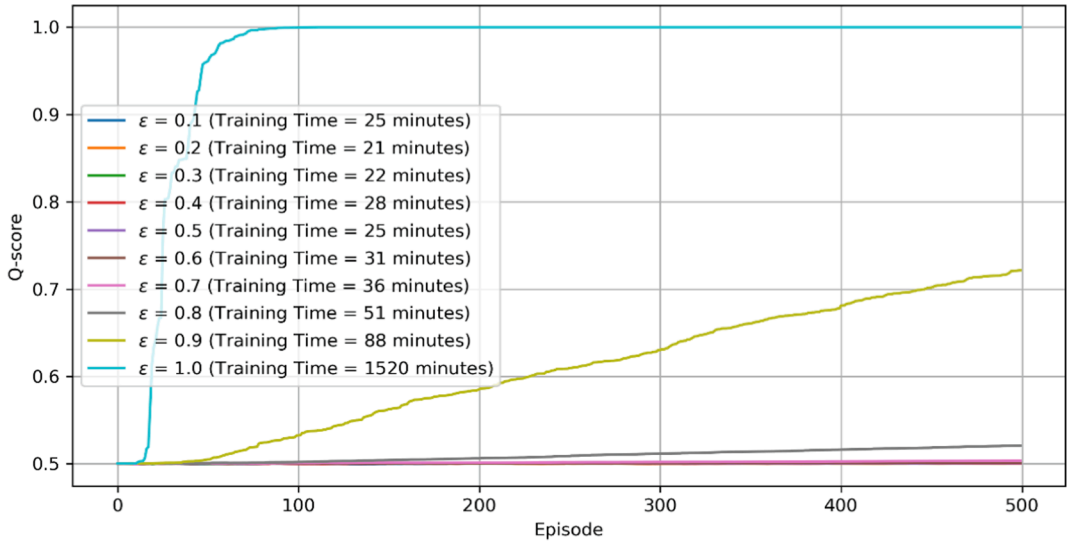

Fig. 8. Q-score over episode when different exploration factor ϵ is used. Each experiment is trained for 500 episodes with $\gamma = 1$, $\alpha = 1$. The total training time is shown with the legend.

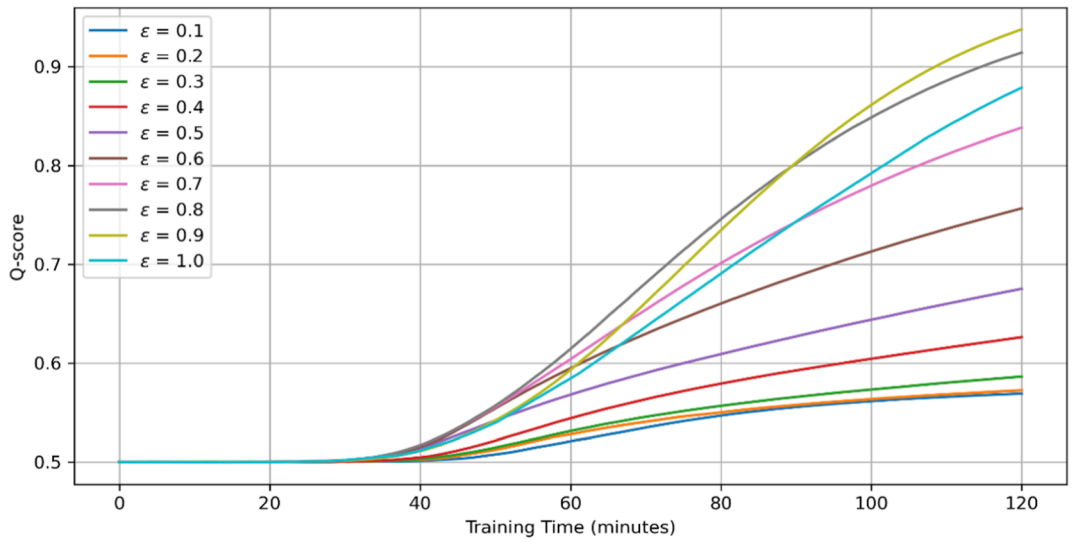

Fig. 9. Q-score over training time when different exploration factors are used. Each experiment is trained for two hours with $\gamma = 1, \alpha = 1$.

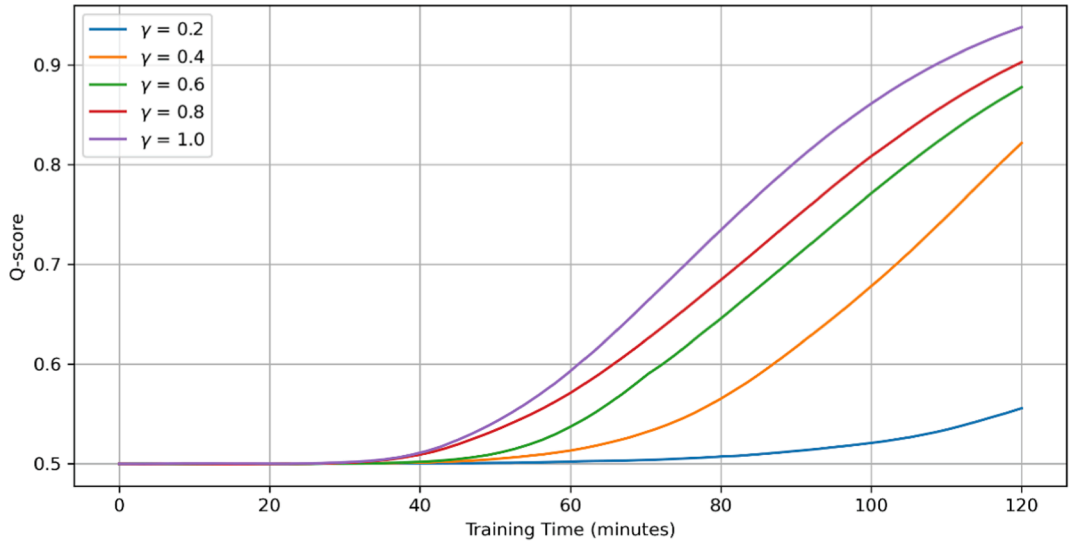

Fig. 10. Q-score over training time when different discount factors are used. Each experiment is trained for two hours with $\epsilon = 0.9$ and $\alpha = 1$.

that the computing time refers to the total training time for the reinforcement learning methods. In contrast, the computing time refers to the total time of solving all states for the traditional methods. The success rate refers to the percentage that a pocket cube is solved by using the method. The unnecessary moves represent the differences between the moves used by the methods and the optimal moves to restore the pocket cube.

The benchmark is presented by Hukmani et al. (2021). They achieved a success rate of 99.8 % using the parallel Q-learning method. The training time for the algorithm is 6 hours. Although they did not provide information about the unnecessary moves, it is mentioned that the average number of moves the algorithm used is 21. Since they used the HTM metric, 11 moves are the maximum number of moves needed to solve any pocket cube. Therefore, it can be inferred that the unnecessary moves are at least 10. By comparison, our Q-learning algorithm achieves a success rate of 100 % with a training time of 7.3 hours. More importantly, the unnecessary moves are reduced to zero, which means that our model finds optimal solutions for all states. Note that the computing time refers to the training time. Pocket cubes can be restored instantly with the guidance of a learned policy.

The computational results for the three traditional methods are interesting. It is found that the three traditional methods can solve the pocket cube problem. The BFS algorithm is the most efficient one. It takes only 1.3 minutes to solve all states. The LP formulation performs second with a computational time of 12 minutes. Dijkstra's algorithm is not desirable to solve the pocket cube problem. It takes 2.7 days to complete the work. This might be because Dijkstra's algorithm spends too much time selecting the node with the smallest label. In conclusion, the proposed Q-learning algorithm improves the success rate from 99.8 % to 100 % and reduces the unnecessary moves from more than ten moves to zero moves. In addition, it is found that the most efficient algorithm is the BFS algorithm.

6. Conclusion

This paper studies solving the pocket cube by reinforcement learning and three traditional methods. The computational results indicate that the BFS algorithm achieves the best performance. It solves the problem within 2 minutes. The LP formulation achieves the second-best performance. The optimal solution is found in 12 minutes using the commercial solver Gurobi. It is interesting to convert the pocket cube to a network flow problem and solve it using Dijkstra's algorithm, but the

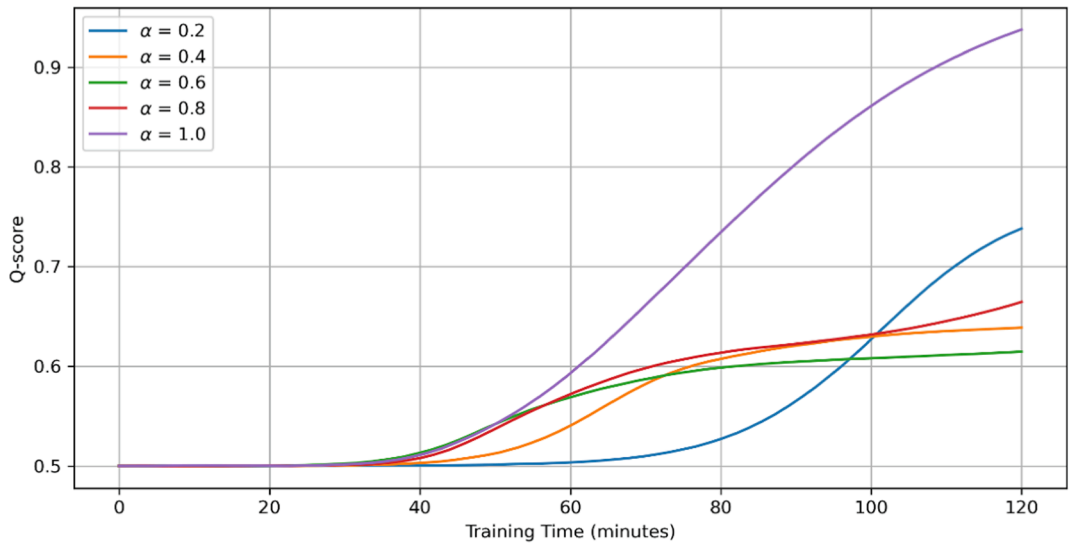

Fig. 11. Q-score over training time when different learning rates are used. Each experiment is trained for two hours with $\epsilon = 0.9$ and $\gamma = 1$.

Table 3
Comparison of the proposed algorithms with the best-known benchmark.

Method	Computing time	Success rate	Unnecessary moves
Parallel Q-learning (Best-known benchmark)	6 hours	99.8 %	≥ 10
Q-learning	7.3 hours	100 %	0
LP method	11.9 minutes	100 %	0
Breadth-first search	1.3 minutes	100 %	0
Dijkstra's algorithm	2.7 days	100 %	0

results are not desirable. It takes almost three days to solve the problem. In comparison, Q-learning with fine-tuned parameters takes around 7 hours to learn the optimal policy.

The advantage of the Q-learning algorithm is that it does not require any domain knowledge. Furthermore, Q-learning can be extended to solve the 3x3x3 Rubik's cube. Traditional methods cannot solve the 3x3x3 Rubik's cube due to the huge consumption of computing memory. Q-learning can overcome this issue by approximating the Q-table using a deep neural network. The disadvantage of Q-learning is that it does not guarantee optimality. However, this may not be a severe issue if the Q-table is trained for enough episodes. According to our experiments, an optimal policy can be learned after training the Q-table for 7 hours using the fine-tuned parameters. With the learned policy, a pocket cube can be restored to its goal state from any initial state using the fewest moves. Traditional methods have the exact opposite advantages and disadvantages of reinforcement learning. They can solve the pocket cube efficiently with a guarantee of optimality, but they cannot be generalized to solve the 3x3x3 Rubik's cube.

Future research may include but not be limited to the following aspects. First, we have reason to believe that Q-learning methods can be enhanced to solve 3x3x3 Rubik's cubes by working with deep learning. Second, we may design acceleration techniques to improve the training process. It is found that training during the early period is inefficient due to the low success rate. Therefore, we may boost the training process by reducing the time spent in the early period. Possible approaches might include pretraining Q-tables, parallel training, and avoiding duplicate training.

CRediT authorship contribution statement

Zefeng Lyu: Methodology, Writing – original draft, Software, Validation, Formal analysis, Visualization, Investigation. **Zeyu Liu:** Methodology, Writing – original draft, Software. **Anahita Khojandi:** Methodology, Investigation. **Andrew Junfang Yu:** Conceptualization, Methodology, Writing – review & editing, Supervision.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors would like to acknowledge and thank Jeremy Watts for his initial contribution to the work presented in this manuscript.

References

Agostinelli, F., McAleer, S., Shmakov, A., & Baldi, P. (2019). Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence, 1*(8), 356–363. https://doi.org/10.1038/s42256-019-0070-z

Bangyal, W. H., Ahmed, J., & Rauf, H. T. (2020). A modified bat algorithm with torus walk for solving global optimisation problems. *International Journal of Bio-Inspired Computation, 15*(1), 1–13.

Bangyal, W. H., Hameed, A., Ahmad, J., Nisar, K., Haque, M. R., Asri, A., … Etengu, R. (2022). New Modified Controlled Bat Algorithm for Numerical Optimization Problem. *Computers, Materials & Continua, 70*(2), 2241–2259. https://doi.org/10.32604/cmc.2022.017789

Bangyal, W. H., Qasim, R., Rehman, N., Ahmad, Z., Dar, H., Rukhsar, L., … Ahmad, J. (2021). Detection of Fake News Text Classification on COVID-19 Using Deep Learning Approaches. *Computational and Mathematical Methods in Medicine, 2021*.

Bowman, N. W., Guo, J. L., & Jones, R. M. J. (2018). *BetaCube: A Deep Reinforcement Learning Approach to Solving 2x2x2 Rubik's cubes Without Human Knowledge*.

Dan, V., Harja, G., & Nașcu, I. (2021). Advanced Rubik ' s Cube Algorithmic Solver. In *2021 7th International Conference on Automation, Robotics and Applications (ICARA)* (pp. 90–94). IEEE.

Golomb, S. W. (1982). Rubik ' s Cube and Quarks Twists on the eight corner cells of Rubik ' s Cube provide a model for many aspects of quark behavior. *American Scientist, 70*(3), 257–259.

Hukmani, K., Kolekar, S., & Vobugari, S. (2021). Solving Twisty Puzzles Using Parallel Q-learning. *Engineering Letters, 29*(4).

Korf, R. E. (1997). Finding optimal solutions to Rubik's cube using pattern databases. *AAAI/IAAI*, 700–705.

Kunkle, D., & Cooperman, G. (2007). Twenty-six moves suffice for Rubik's cube. *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, 235–242. https://doi.org/10.1145/1277548.1277581

Rokicki, T. (2010). Twenty-two moves suffice for Rubik's cube®. *The Mathematical Intelligencer, 32*(1), 33–40. https://doi.org/10.1007/s00283-009-9105-3

Rokicki, T., Kociemba, H., Davidson, M., & Dethridge, J. (2013). The diameter of the Rubik's cube group is twenty. *SIAM Journal on Discrete Mathematics, 27*(2), 1082–1105. https://doi.org/10.1137/120867366

Smith, R. J., Kelly, S., & Heywood, M. I. (2016). Discovering Rubik's cube subgroups using coevolutionary GP - A five twist experiment. *GECCO 2016 - Proceedings of the 2016 Genetic and Evolutionary Computation Conference*, 789–796. https://doi.org/10.1145/2908812.2908887

Toshniwal, M. E. S. (2019). Rubik's cube Solver: A Review. *2019 9th International Conference on Emerging Trends in Engineering and Technology-Signal and Information Processing (ICETET-SIP-19)*, 1–5. IEEE.

Varga, J., Durovsky, F., & Kovac, J. (2014). Design of pneumatical Rubik ' s cube solver. *Applied Mechanics and Materials, 613*, 265–272. https://doi.org/10.4028/www.scientific.net/AMM.613.265

Watkins, C. J., & Dayan, P. (1992). Q-Learning. *Machine Learning, 8*, 279–292. https://doi.org/10.4018/978-1-59140-993-9.ch026