

Solving the Rubik's cube with deep reinforcement learning and search

Forest Agostinelli^{1,3}, Stephen McAleer^{2,3}, Alexander Shmakov^{1,3} and Pierre Baldi^{1,2*}

The Rubik's cube is a prototypical combinatorial puzzle that has a large state space with a single goal state. The goal state is unlikely to be accessed using sequences of randomly generated moves, posing unique challenges for machine learning. We solve the Rubik's cube with DeepCubeA, a deep reinforcement learning approach that learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge. DeepCubeA solves 100% of all test configurations, finding a shortest path to the goal state 60.3% of the time. DeepCubeA generalizes to other combinatorial puzzles and is able to solve the 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban, finding a shortest path in the majority of verifiable cases.

The Rubik's cube is a classic combinatorial puzzle that poses unique and interesting challenges for artificial intelligence and machine learning. Although the state space is exceptionally large (4.3×10^{19} different states), there is only one goal state. Furthermore, the Rubik's cube is a single-player game and a sequence of random moves, no matter how long, is unlikely to end in the goal state. Developing machine learning algorithms to deal with this property of the Rubik's cube might provide insights into learning to solve planning problems with large state spaces. Although machine learning methods have previously been applied to the Rubik's cube, these methods have either failed to reliably solve the cube^{1–4} or have had to rely on specific domain knowledge^{5,6}. Outside of machine learning methods, methods based on pattern databases (PDBs) have been effective at solving puzzles such as the Rubik's cube, the 15 puzzle and the 24 puzzle^{7,8}, but these methods can be memory-intensive and puzzle-specific.

More broadly, a major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge. The classical $3 \times 3 \times 3$ Rubik's cube is only one representative of a larger family of possible environments that broadly share the characteristics described above, including (1) cubes with longer edges or higher dimension (for example, $4 \times 4 \times 4$ or $2 \times 2 \times 2 \times 2$), (2) sliding tile puzzles (for example the 15 puzzle, 24 puzzle, 35 puzzle and 48 puzzle), (3) Lights Out and (4) Sokoban. As the size and dimensions are increased, the complexity of the underlying combinatorial problems rapidly increases. For example, while finding an optimal solution to the 15 puzzle takes less than a second on a modern-day desktop, finding an optimal solution to the 24 puzzle can take days, and finding an optimal solution to the 35 puzzle is generally intractable⁹. Not only are the aforementioned puzzles relevant as mathematical games, but they can also be used to test planning algorithms¹⁰ and to assess how well a machine learning approach may generalize to different environments. Furthermore, because the operation of the Rubik's cube and other combinatorial puzzles are deeply rooted in group theory, these puzzles also raise broader questions about the application of machine learning methods to complex symbolic systems, including mathematics. In short, for all these reasons, the Rubik's cube poses interesting challenges for machine learning.

To address these challenges, we have developed DeepCubeA, which combines deep learning^{11,12} with classical reinforcement learning¹³ (approximate value iteration^{14–16}) and path finding methods (weighted A* search^{17,18}). DeepCubeA is able to solve combinatorial puzzles such as the Rubik's cube, 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban (Fig. 1). DeepCubeA works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function). Given that random play is unlikely to end in the goal state, DeepCubeA trains on states obtained by starting from the goal state and randomly taking moves in reverse. After training, the learned cost-to-go function is used as a heuristic to solve the puzzles using a weighted A* search^{17–19}.

DeepCubeA builds on DeepCube²⁰, a deep reinforcement learning algorithm that solves the Rubik's cube using a policy and value function combined with Monte Carlo tree search (MCTS). MCTS, combined with a policy and value function, is also used by AlphaZero, which learns to beat the best existing programs in chess, Go and shogi²¹. In practice, we find that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path. In contrast, DeepCubeA finds a shortest path to the goal for puzzles for which a shortest path is computationally verifiable: 60.3% of the time for the Rubik's cube and over 90% of the time for the 15 puzzle, 24 puzzle and Lights Out.

Deep approximate value iteration

Value iteration¹⁵ is a dynamic programming algorithm^{14,16} that iteratively improves a cost-to-go function J . In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s . However, this lookup table representation becomes infeasible for combinatorial puzzles with large state spaces like the Rubik's cube. Therefore, we turn to approximate value iteration¹⁶, where J is represented by a parameterized function implemented by a DNN. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go of state s , $J(s)$, and the updated cost-to-go estimation $J'(s)$:

¹Department of Computer Science, University of California Irvine, Irvine, CA, USA. ²Department of Statistics, University of California Irvine, Irvine, CA, USA. ³These authors contributed equally: Forest Agostinelli, Stephen McAleer, Alexander Shmakov. *e-mail: pfbaldi@uci.edu

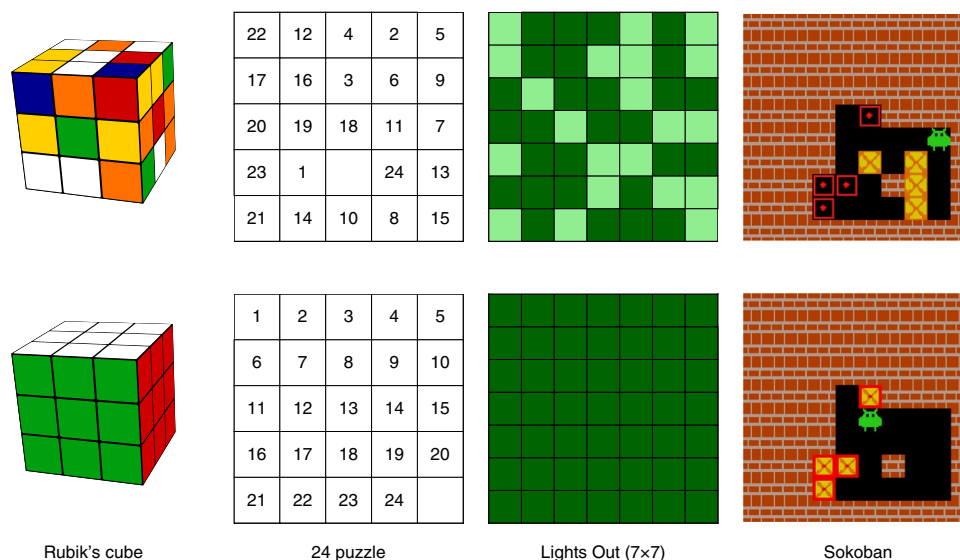


Fig. 1 | Visualization of scrambled states and goal states. Visualization of a scrambled state (top) and the goal state (bottom) for four puzzles investigated here.

$$J'(s) = \min_a (g^a(s, A(s, a)) + J(A(s, a))) \quad (1)$$

where $A(s, a)$ is the state obtained from taking action a in state s and $g^a(s, s')$ is the cost to transition from state s to state s' taking action a . For the puzzles investigated in this Article, $g^a(s, s')$ is always 1. We call the resulting algorithm 'deep approximate value iteration' (DAVI).

We train on a state distribution that allows information to propagate from the goal state to all the other states seen during training. Our method of achieving this is simple: each training state x_i is obtained by randomly scrambling the goal state k_i times, where k_i is uniformly distributed between 1 and K .

Although the update in equation (1) is only a one-step lookahead, it has been shown that, as training progresses, J approximates the optimal cost-to-go function J^* (ref. 16). This optimal cost-to-go function computes the total cost incurred when taking a shortest path to the goal. Instead of equation (1), multi-step lookaheads such as a depth- N search or Monte Carlo tree search can be used. We experimented with different multi-step lookaheads and found that multi-step lookahead strategies resulted in, at best, similar performance to the one-step lookahead used by DAVI (see Methods for more details).

Batch weighted A* search

After learning a cost-to-go function, we can then use it as a heuristic to search for a path between a starting state and the goal state. The search algorithm that we use is a variant of A* search¹⁷, a best-first search algorithm that iteratively expands the node with the lowest cost until the node associated with the goal state is selected for expansion. The cost of each node x in the search tree is

determined by the function $f(x) = g(x) + h(x)$, where $g(x)$ is the path cost, which is the distance between the starting state and x , and $h(x)$ is the heuristic function, which estimates the distance between x and the goal state. The heuristic function $h(x)$ is obtained from the learned cost-to-go function:

$$h(x) = \begin{cases} 0 & \text{if } x \text{ is associated with the goal state} \\ J(x) & \text{otherwise} \end{cases} \quad (2)$$

We use a variant of A* search called weighted A* search¹⁸. Weighted A* search trades potentially longer solutions for potentially less memory usage by using, instead, the function $f(x) = \lambda g(x) + h(x)$, where λ is a weighting factor between zero and one. Furthermore, using a computationally expensive model for the heuristic function $h(x)$, such as a DNN, could result in an intractably slow solver. However, $h(x)$ can be computed for many nodes in parallel by expanding the N lowest cost nodes at each iteration. We call the combination of A* search with a path-cost coefficient λ and batch size N 'batch weighted A* search' (BWAS).

In summary, the algorithm presented in this Article uses DAVI to train a DNN as the cost-to-go function on states whose difficulty ranges from easy to hard. The trained cost-to-go function is then used as a heuristic for BWAS to find a path from any given state to the goal state. We call the resulting algorithm DeepCubeA.

Results

To test the approach, we generate a test set of 1,000 states by randomly scrambling the goal state between 1,000 and 10,000 times. Additionally, we test the performance of DeepCubeA on the three

Table 1 | Comparison of DeepCubeA with optimal solvers based on PDBs along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem and number of nodes generated per second on the Rubik's cube states that are furthest away from the goal

| Puzzle | Solver | Length | Percentage of optimal solutions | No. of nodes | Time taken (s) | Nodes per second |
|---------------------------|---------------------|--------|---------------------------------|-----------------------|----------------|--------------------|
| Rubik's cube _n | PDBs ⁷ | – | – | – | – | – |
| | PDBs+ ²⁴ | 26.00 | 100.0 | 2.41×10^{10} | 13,561.27 | 1.78×10^6 |
| | DeepCubeA | 26.00 | 100.0 | 5.33×10^6 | 18.77 | 2.96×10^5 |

PDBs+ refer to Rokicki's optimal solver, which uses PDB combined with knowledge of group theory^{24,25}. DeepCubeA finds a shortest path to the goal for all of the states furthest away from the goal.

Table 2 | Comparison of the size (in GB) of the lookup tables for pattern PDBs and the size of the DNN used by DeepCubeA

| | Rubik's cube | 15 puzzle | 24 puzzle | 35 puzzle | 48 puzzle | Lights Out | Sokoban |
|-------------------|--------------|-----------|-----------|-----------|-----------|------------|---------|
| PDBs | 4.67 | 8.51 | 1.86 | 0.64 | 4.86 | – | – |
| PDBs ⁺ | 182.00 | – | – | – | – | – | – |
| DeepCubeA | 0.06 | 0.06 | 0.08 | 0.08 | 0.10 | 0.05 | 0.06 |

PDBs⁺ refers to Rokicki's PDB combined with knowledge of group theory^{24,25}. The table shows that DeepCubeA always uses memory that is orders of magnitude less than PDBs.

known states that are the furthest possible distance away from the goal (26 moves)²². To assess how often DeepCubeA finds a shortest path to the goal, we need to compare our results to a shortest path solver. We can obtain a shortest path solver by using an **iterative deepening A* search (IDA*)**²³ with an admissible heuristic computed from a **PDB**. Initially, we used the PDB described in Korf's work on finding optimal solutions to the Rubik's cube⁷; however, this solver only solves a few states a day. Therefore, we use the optimal solver that was used to find the maximum of the minimum number of moves required to solve the Rubik's cube from any given state (so-called 'God's number')^{24,25}. This human-engineered solver relies on large PDBs²⁶ (requiring 182 GB of memory) and sophisticated knowledge of group theory to find a shortest path to the goal state. Comparisons between DeepCubeA and shortest path solvers are shown later in Table 5.

The DNN architecture consists of **two fully connected hidden layers**, followed by **four residual blocks**²⁷, followed by a **linear output unit** that represents the cost-to-go estimate. The hyperparameters of BWAS were chosen by doing a grid search over λ and N on data generated separately from the test set (see Methods for more details). When performing BWAS, the heuristic function is computed in parallel across four NVIDIA Titan V graphics processing units (GPUs).

Performance. DeepCubeA finds a solution to 100% of all test states. DeepCubeA finds a shortest path to the goal 60.3% of the time. Aside from the optimal solutions, 36.4% of the solutions are only two moves longer than the optimal solution, while the remaining 3.3% are four moves longer than the optimal solution. For the three states that are furthest away from the goal, DeepCubeA finds a shortest path to the goal for all three states (Table 1). Although we relate the performance of DeepCubeA to the performance of shortest path solvers based on **PDBs**, a direct comparison cannot be made because shortest path solvers guarantee an optimal solution while DeepCubeA does not.

Although PDBs can be used in shortest path solvers, they can also be used in BWAS in place of the heuristic learned by

DeepCubeA. We use Korf's PDB heuristic for BWAS and compare to DeepCubeA. We perform BWAS with $N=10,000$ and $\lambda=0.0, 0.1$ and 0.2 . We compute the PDB heuristic in parallel across 32 central processing units (CPUs). Note that at $\lambda=0.3$ BWAS runs out of memory when using PDBs. Figure 2 shows that performing BWAS with DeepCubeA's learned heuristic consistently produces shorter solutions, generates fewer nodes and is overall much faster than **Korf's PDB heuristic**.

We also compare the memory footprint and speed of pattern databases to DeepCubeA. In terms of memory, for pattern databases, it is necessary to load lookup tables into memory. For DeepCubeA, it is necessary to load the DNN into memory. Table 2 shows that DeepCubeA uses significantly less memory than PDBs. In terms of speed, we measure how quickly PDBs and DeepCubeA compute a heuristic for a single state, averaging over 1,000 states. Given that DeepCubeA uses neural networks, which benefit from GPUs and batch processing, we measure the speed of DeepCubeA with both a single CPU and a single GPU, and with both sequential and batch processing of the states. Table 3 shows that, as expected, PDBs on a single CPU are faster than DeepCubeA on a single CPU; however, the speed of PDBs on a single CPU is comparable to the speed of DeepCubeA on a single GPU with batch processing.

During training we monitor how well the DNN is able to solve the Rubik's cube using a greedy best-first search; we also monitor how well the DNN is able to estimate the optimal cost-to-go function (computed with Rokicki's shortest path solver²⁵). How these performance metrics change as a function of training iteration is shown in Fig. 3. The results show that DeepCubeA first learns to solve states closer to the goal before it learns to solve states further away from the goal. Cost-to-go estimation is less accurate for states further away from the goal; however, the cost-to-go function still correctly orders the states according to difficulty. In addition, we found that DeepCubeA frequently used the conjugate patterns of moves of the form aba^{-1} in its solutions and often found symmetric solutions to symmetric states. An example of this is shown in Fig. 4 (see Methods for more details).

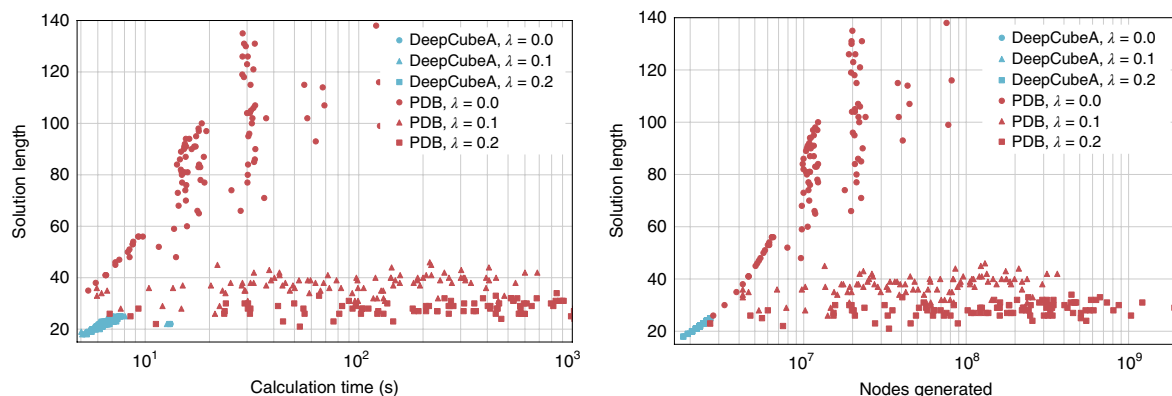


Fig. 2 | The performance of DeepCubeA versus PDBs when solving the Rubik's cube with BWAS. $N=10,000$ and λ is either 0.0, 0.1 or 0.2. Each dot represents the result on a single state. DeepCubeA is both faster and produces shorter solutions.

Table 3 | A suggestive comparison of the speed (in seconds) of the lookup tables for PDBs and the speed of the DNN used by DeepCubeA when computing the heuristic for a single state

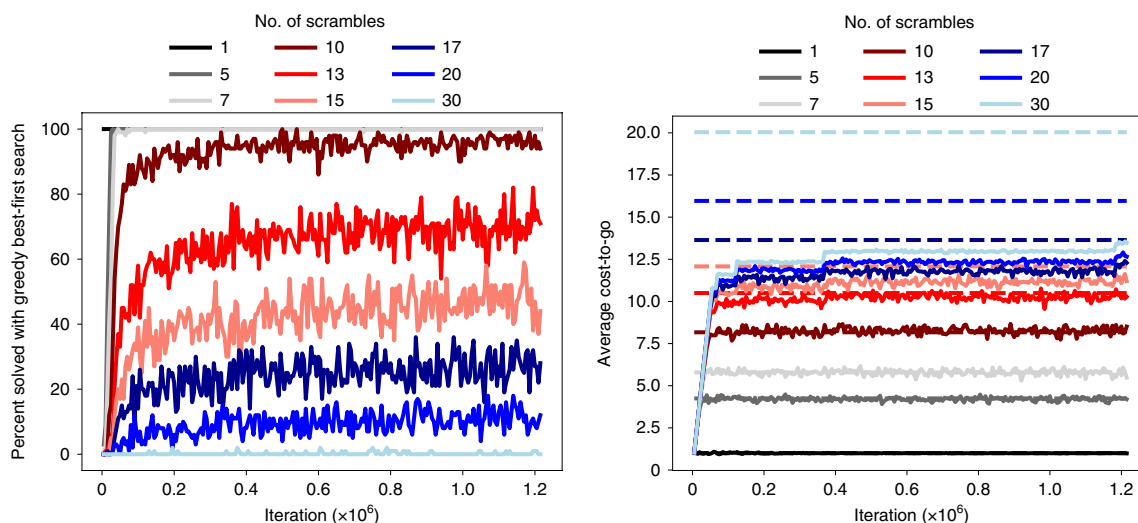
| | Rubik's cube | 15 puzzle | 24 puzzle | 35 puzzle | 48 puzzle | Lights Out | Sokoban |
|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| PDBs | 2×10^6 | 1×10^6 | 2×10^6 | 3×10^6 | 4×10^6 | - | - |
| PDBs ⁺ | 6×10^7 | - | - | - | - | - | - |
| DeepCubeA (GPU-B) | 6×10^6 | 6×10^6 | 7×10^6 | 8×10^6 | 9×10^6 | 7×10^6 | 6×10^6 |
| DeepCubeA (GPU) | 3×10^3 | 3×10^3 | 3×10^3 | 2×10^3 | 3×10^3 | 4×10^3 | 3×10^3 |
| DeepCubeA (CPU-B) | 7×10^4 | 6×10^4 | 9×10^4 | 9×10^4 | 1×10^3 | 1×10^3 | 7×10^4 |
| DeepCubeA (CPU) | 6×10^3 | 6×10^3 | 8×10^3 | 8×10^3 | 1×10^2 | 2×10^1 | 6×10^3 |

Results were averaged over 1,000 states. DeepCubeA was timed on a single CPU and on a single GPU when doing sequential processing of the states and batch processing of the states (batch processing is denoted by the '-B' suffix). PDBs⁺ refers to Rokicki's PDB combined with knowledge of group theory^{24,25}. On a GPU, DeepCubeA is comparable to PDBs.

Table 4 | Comparison of DeepCubeA with optimal solvers based on PDBs along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem and number of nodes generated per second for the 24 puzzle and 35 puzzle

| Puzzle | Solver | Length | Percentage of optimal solutions | No. of nodes | Time taken (s) | Nodes per second |
|-----------|-------------------|--------|---------------------------------|-----------------------|----------------|--------------------|
| 24 puzzle | PDBs ⁹ | 89.41 | 100.0 | 8.19×10^{10} | 4,239.54 | 1.91×10^7 |
| | DeepCubeA | 89.49 | 96.98 | 6.44×10^6 | 19.33 | 3.34×10^5 |
| 35 puzzle | PDBs ⁹ | - | - | - | - | - |
| | DeepCubeA | 124.64 | - | 9.26×10^6 | 28.45 | 3.25×10^5 |

For the 24 puzzle, DeepCubeA finds a shortest path to the goal the overwhelming majority of the time. For the 35 puzzle, no tractable optimal solver exists.

**Fig. 3 | The performance of DeepCubeA.** The plots show that DeepCubeA first learns how to solve cubes closer to the goal and then learns to solve increasingly difficult cubes. Dashed lines represent the true average cost-to-go.

Generalization to other combinatorial puzzles. The Rubik's cube is only one combinatorial puzzle among many others. To demonstrate the ability of DeepCubeA to generalize to other puzzles, we applied DeepCubeA to four popular sliding tile puzzles: the 15 puzzle, the 24 puzzle, 35 puzzle and 48 puzzle. Additionally, we applied DeepCubeA to Lights Out and Sokoban. Sokoban posed a unique challenge for DeepCubeA because actions taken in its environment are not always reversible.

Sliding tile puzzles. The 15 puzzle has 1.0×10^{13} possible combinations, the 24 puzzle has 7.7×10^{24} possible combinations, the 35 puzzle has 1.8×10^{41} possible combinations and the 48 puzzle has 3.0×10^{62} possible combinations. The objective is to move the puzzle

into its goal configuration shown in Fig. 1. For these sliding tile puzzles, we generated a test set of 500 states randomly scrambled between 1,000 and 10,000 times. The same DNN architecture and hyperparameters that are used for the Rubik's cube are also used for the n puzzles with the exception of the addition of two more residual layers. We implemented an optimal solver using additive pattern databases⁹. DeepCubeA not only solved every test puzzle, but also found a shortest path to the goal 99.4% of the time for the 15 puzzle and 96.98% of the time for the 24 puzzle. We also test on the 17 states that are furthest away from the goal for the 15 puzzle (these states are not known for the 24 puzzle)²⁸. Solutions produced by DeepCubeA are, on average, 2.8 moves longer than the length of a shortest path and DeepCubeA finds a shortest path to the goal

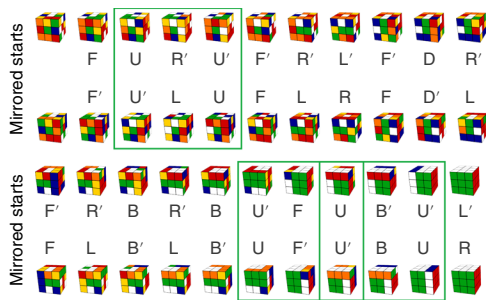


Fig. 4 | An example of symmetric solutions that DeepCubeA finds to symmetric states. Conjugate triplets are indicated by green boxes. Note that the last two conjugate triplets are overlapping.

for 17.6% of these states. For the 24 puzzle, on average, PDBs take 4,239 s and DeepCubeA takes 19.3 s, over 200 times faster. Moreover, in the worst case we observed that the longest time needed to solve the 24 puzzle is 5 days for PDBs and 2 min for DeepCubeA. The average solution length is 124.76 for the 35 puzzle and 253.53 for the 48 puzzle; however, we do not know how many of them are optimal due to the optimal solver being prohibitively slow for the 35 puzzle and 48 puzzle. The performances of DeepCubeA on the 24 puzzle and 35 puzzle are summarized in Table 4.

Although the shortest path solver for the 35 puzzle and 48 puzzle was prohibitively slow, we compare DeepCubeA to PDBs using BWAS. The results show that, compared to PDBs, DeepCubeA produces shorter solutions and generates fewer nodes, as shown in Supplementary Figs. 6 and 7. In combination, these results suggest

that, as the size of the n -puzzle increases, DeepCubeA scales favourably compared to PDBs.

Lights Out. Lights Out is a grid-based puzzle consisting of an $N \times N$ board of lights that may be either active or inactive. The goal is to convert all active lights to inactive from a random starting position, as seen in Fig. 1. Pressing any light in the grid will switch the state of that light and its immediate horizontal and vertical neighbours. At any given state, a player may click on any of the N^2 lights. However, one difference of Lights Out compared to the other environments is that the moves are commutative. We tested DeepCubeA on the 7×7 Lights Out puzzle. A theorem by Scherphuis²⁹ shows that, for the 7×7 Lights Out puzzle, any solution that does not contain any duplicate moves is an optimal solution. Using this theorem, we found that DeepCubeA found a shortest path to the goal for all test cases.

Sokoban. Sokoban³⁰ is a planning problem that requires an agent to move boxes onto target locations. Boxes can only be pushed, not pulled. Note that training states are generated by pulling boxes instead of pushing them (see Methods for more details). To test our method on Sokoban, we train on the 900,000 training examples and test on the 1,000 testing examples used by previous research on a single-agent policy tree search applied to Sokoban³¹. DeepCubeA successfully solves 100% of all test examples. We compare the solution length and number of nodes expanded to this same previous research³². Although the goals of the aforementioned paper are slightly different from ours, DeepCubeA finds shorter paths than previously reported methods and also expands, at least, three times fewer nodes (Table 5).

Table 5 | Comparison of DeepCubeA with optimal solvers based on PDBs along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem and number of nodes generated per second

| Puzzle | Solver | Length | Percentage of optimal solutions | No. of nodes | Time taken (s) | Nodes per second |
|---------------------------|--------------------------|--------|---------------------------------|-----------------------|----------------|--------------------|
| Rubik's cube | PDBs ⁷ | – | – | – | – | – |
| | PDBs ⁺²⁴ | 20.67 | 100.0 | 2.05×10^6 | 2.20 | 1.79×10^6 |
| | DeepCubeA | 21.50 | 60.3 | 6.62×10^6 | 24.22 | 2.90×10^5 |
| Rubik's cube _n | PDBs ⁷ | – | – | – | – | – |
| | PDBs ⁺²⁴ | 26.00 | 100.0 | 2.41×10^{10} | 13,561.27 | 1.78×10^6 |
| | DeepCubeA | 26.00 | 100.0 | 5.33×10^6 | 18.77 | 2.96×10^5 |
| 15 puzzle | PDBs ⁹ | 52.02 | 100.0 | 3.22×10^4 | 0.002 | 1.45×10^7 |
| | DeepCubeA | 52.03 | 99.4 | 3.85×10^6 | 10.28 | 3.93×10^5 |
| 15 puzzle _n | PDBs ⁹ | 80.00 | 100.0 | 1.53×10^7 | 0.997 | 1.56×10^7 |
| | DeepCubeA | 82.82 | 17.65 | 2.76×10^7 | 69.36 | 3.98×10^5 |
| 24 puzzle | PDBs ⁹ | 89.41 | 100.0 | 8.19×10^{10} | 4,239.54 | 1.91×10^7 |
| | DeepCubeA | 89.49 | 96.98 | 6.44×10^6 | 19.33 | 3.34×10^5 |
| 35 puzzle | PDBs ⁹ | – | – | – | – | – |
| | DeepCubeA | 124.64 | – | 9.26×10^6 | 28.45 | 3.25×10^5 |
| 48 puzzle | PDBs | – | – | – | – | – |
| | DeepCubeA | 253.35 | – | 1.96×10^7 | 74.46 | 2.63×10^5 |
| Lights Out | DeepCubeA | 24.26 | 100.0 | 1.14×10^6 | 3.27 | 3.51×10^5 |
| Sokoban | LevinTS ³² | 39.80 | – | 6.60×10^3 | – | – |
| | LevinTS(*) ³² | 39.50 | – | 5.03×10^3 | – | – |
| | LAMA ³² | 51.60 | – | 3.15×10^3 | – | – |
| | DeepCubeA | 32.88 | – | 1.05×10^3 | 2.35 | 5.60×10^1 |

The datasets with an 'h' subscript are datasets containing the states that are furthest away from the goal state. PDBs⁺ refers to Rokicki's PDB combined with knowledge of group theory^{24,25}. For Sokoban, we compare nodes expanded instead of nodes generated to allow for a direct comparison to previous work. DeepCubeA often finds a shortest path to the goal. For the states that are furthest away from the goal, DeepCubeA either finds a shortest path or a path close in length to a shortest path.

Discussion

DeepCubeA is able to solve planning problems with large state spaces and few goal states by learning a cost-to-go function, parameterized by a DNN, which is then used as a heuristic function for weighted A* search. The cost-to-go function is learned by using approximate value iteration on states generated by starting from the goal state and taking moves in reverse. DeepCubeA's success in solving the problems investigated in this Article suggests that DeepCubeA can be readily applied to new problems given an input representation, a state transition model, a goal state and a reverse state transition model that can be used to adequately explore the state space.

With a heuristic function that never overestimates the cost of a shortest path (that is, an admissible heuristic function), weighed A* search comes with known bounds on how much the length of a solution can deviate from the length of an optimal solution. Although DeepCubeA's heuristic function is not guaranteed to be admissible, and thus does not satisfy the requirement for these theoretical bounds, DeepCubeA nevertheless finds a shortest path to the goal in the majority of cases (see Methods for more details).

The generality of the core algorithm suggests that it may have applications beyond combinatorial puzzles, as problems with large state spaces and few goal states are not rare in planning, robotics and the natural sciences.

Methods

The Rubik's cube. The $3 \times 3 \times 3$ Rubik's cube consists of smaller cubes called cubelets. These are classified by their sticker count: centre, edge and corner cubelets have 1, 2 and 3 stickers, respectively. The Rubik's cube has 26 cubelets with 54 stickers in total. The stickers have colours and there are six colours, one per face. In the solved state, all stickers on each face of the cube are the same colour. Given that the set of stickers on each cubelet is unique (that is, there is only one cubelet with white, red and green stickers), the 54 stickers themselves can be uniquely identified in any legal configuration of the Rubik's cube. The representation given to the DNN encodes the colour of each sticker at each location using a one-hot encoding. As there are six possible colours and 54 stickers in total, this results in a state representation of size 324.

Moves are represented using face notation: a move is a letter stating which face to rotate. **F**, **B**, **L**, **R**, **U** and **D** correspond to turning the front, back, left, right, up and down faces, respectively. Each face name is in reference to a fixed front face. A clockwise rotation is represented with a single letter, whereas a letter followed by an apostrophe represents an anticlockwise rotation. For example: R rotates the right face by 90° clockwise, while R' rotates it by 90° anticlockwise.

The Rubik's cube state space has 4.3×10^{19} possible states. Any valid Rubik's cube state can be optimally solved with at most 26 moves in the quarter-turn metric, or 20 moves in the half-turn metric^{22,25}. The quarter-turn metric treats 180° rotations as two moves, whereas the half-turn metric treats 180° rotations as one move. We use the quarter-turn metric.

Additional combinatorial puzzles. *Sliding puzzles.* Another combinatorial puzzle we use to test DeepCubeA is the n -piece sliding puzzle. In the n puzzle, n square sliding tiles, numbered from 1 to n , are positioned in a square of length $\sqrt{n+1}$, with one empty tile position. Thus, the 15 puzzle consists of 15 tiles in a 4×4 grid, the 24 puzzle consists of 24 tiles in a 5×5 grid, the 35 puzzle consists of 35 tiles in a 6×6 grid and the 48 puzzle consists of 48 tiles in a 7×7 grid. Moves are made by swapping the empty position with any tile that is horizontally or vertically adjacent to it. For both puzzles, the representation given to the neural network uses one-hot encoding to specify which piece (tile or blank position) is in each position. For example, the dimension of the input to the neural network for the 15 puzzle would be $16 \times 16 = 256$. The 15 puzzle has $16!/2 \approx 1.0 \times 10^{13}$ possible states, the 24 puzzle has $25!/2 \approx 7.7 \times 10^{24}$ possible states, the 35 puzzle has $36!/2 \approx 1.8 \times 10^{41}$ possible states and the 48 puzzle has $49!/2 \approx 3.0 \times 10^{62}$ possible states. Any valid 15 puzzle configuration can be solved with at most 80 moves^{33,34}. The largest minimal numbers of moves required to solve the 24 puzzle, 35 puzzle and 48 puzzle are not known.

Lights Out. Lights Out contains N^2 lights on an $N \times N$ board. The lights can either be on or off. The representation given to the DNN is a vector of size N^2 . Each element is 1 if the corresponding light is on and 0 if the corresponding light is off.

Sokoban. The Sokoban environment we use is a 10×10 grid that contains four boxes that an agent needs to push onto four targets. In addition to the agent, boxes and targets, Sokoban also contains walls. The representation given to the DNN contains four binary vectors of size 10^2 that represent the position on the agent, boxes, targets and walls. Given that boxes can only be pushed, not pulled, some actions are irreversible. For example, a box pushed into a corner can no longer be

moved, creating a sampling problem because some states are unreachable when starting from the goal state. To address this, for each training state, we start from the goal state and allow boxes to be pulled instead of pushed.

Deep approximate value iteration. Value iteration¹⁵ is a dynamic programming algorithm^{14,16} that iteratively improves a cost-to-go function J . In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s . Value iteration loops through each state s and updates $J(s)$ until convergence:

$$J(s) \leftarrow \min_a \sum_{s'} P^a(s, s') (g^a(s, s') + \gamma J(s')) \quad (3)$$

Here $P^a(s, s')$ is the transition matrix representing the probability of transitioning from state s to state s' by taking action a ; $g^a(s, s')$ is the cost associated with transitioning from state s to s' by taking action a ; γ is the discount factor. In principle, this update equation can also be applied to the puzzles investigated in this Article. However, as these puzzles are deterministic, the transition function is a degenerate probability mass function for each action, simplifying equation (3). Furthermore, because we wish to assign equal importance to all costs, $\gamma = 1$. Therefore, we can update $J(s)$ using equation (1).

However, given the size of the state space of the Rubik's cube, maintaining a table to store the cost-to-go of each state is not feasible. Therefore, we resort to approximate value iteration¹⁶. Instead of representing the cost-to-go function as a lookup table, we approximate the cost-to-go function using a parameterized function j_θ with parameters θ . This function is implemented using a DNN.

Therefore, we call the resulting algorithm DAVI:

Algorithm 1: DAVI.

Input:

- B:** Batch size
- K:** Maximum number of scrambles
- M:** Training iterations
- C:** How often to check for convergence
- ϵ :** Error threshold

Output:

- θ :** Trained neural network parameters

$\theta \leftarrow \text{initialize_parameters}()$

$\theta_e \leftarrow \theta$

for $m = 1$ **to** M **do**

$X \leftarrow \text{get_scrambled_states}(B, K)$

for $x_i \in X$ **do**

$y_i \leftarrow \min_a (g^a(x_i, A(x_i, a)) + j_{\theta_e}(A(x_i, a)))$

$\theta, \text{loss} \leftarrow \text{train}(j_\theta, X, y)$

if $(M \bmod C = 0)$ **and** $(\text{loss} < \epsilon)$ **then**

$\theta_e \leftarrow \theta$

Return θ

To train the DNN, we have two sets of parameters: the parameters being trained, θ , and the parameters used to obtain an improved estimate of the cost-to-go, θ_e . The output of $j_{\theta_e}(s)$ is set to 0 if s is the goal state. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go and the estimation obtained from equation (1). Every C iterations, the algorithm checks if the error falls below a certain threshold ϵ ; if so, then θ_e is set to θ . The entire DAVI process is shown in Algorithm 1. Although we tried updating θ_e at each iteration, we found that the performance saturated after a certain point and sometimes became unstable. Updating θ_e only after the error falls below a threshold ϵ yields better, more stable, performance.

Training set state distribution. For learning to occur, we must train on a state distribution that allows information to propagate from the goal state to all the other states seen during training. Our approach for achieving this is simple: each training state x_i is obtained by randomly scrambling the goal state k_i times, where k_i is uniformly distributed between 1 and K . During training, the cost-to-go function first improves for states that are only one move away from the goal state. The cost-to-go function then improves for states further away as the reward signal is propagated from the goal state to other states through the cost-to-go function. This can be seen as a simplified version of prioritized sweeping³⁵. Exploring in reverse from the goal state is a well-known technique and has been used in means-end analysis³⁶ and STRIPS³⁷. In future work we will explore different ways of generating a training set distribution.

Distributed training. In the Rubik's cube environment, there are 12 possible actions that can be applied to every state. Using equation (1) to update the cost-to-go estimate of a single state thus requires applying the DNN to 12 states. As a result, equation (1) takes up the majority of the computational time. However, as is the case with methods such as ExIt³⁸, this is a trivially parallelizable task that can easily be distributed across multiple GPUs.

BWAS. A* search¹⁷ is a heuristic-based search algorithm that finds a path between a starting node x_s and a goal node x_g . A* search maintains a set, OPEN, from which it iteratively removes and expands the node with the lowest cost. The cost of each node x is determined by the function $f(x) = g(x) + h(x)$, where $g(x)$ is the path cost

(the distance between x_i and x) and $h(x)$ is the heuristic function, which estimates the distance between x and x_g . After a node is expanded, that node is then added to another set, CLOSED, and its children that are not already in CLOSED are added to OPEN. The algorithm starts with only the starting node in OPEN and terminates when the goal node is removed from OPEN.

In this application, each node corresponds to a state of the Rubik's cube and the goal node corresponds to the goal state shown in Fig. 1. The path cost of every child of a node x is set to $g(x) + 1$. The path cost of x_g is 0. The heuristic function $h(x)$ is obtained from the learned cost-to-go function shown in equation (2).

A variant of A* search, called weighted A* search³⁹, trades potentially longer solutions for potentially less memory usage. In this case, the function $f(x)$ is modified to $f(x) = \lambda g(x) + h(x)$, with weight $\lambda \in [0, 1]$. While decreasing the weight λ will not necessarily decrease the number of nodes generated³⁹, in practice our experiments show that decreasing λ generally reduces the number of nodes generated and increases the length of the solutions found. In our implementation, if we encounter a node x that is already in CLOSED, and if x has a lower path cost than the node that is already in CLOSED, we remove that node from CLOSED and add x to OPEN.

The most time-consuming aspect of the algorithm is the computation of the heuristic $h(x)$. The heuristic of many nodes can be computed in parallel across multiple GPUs by expanding the N best nodes from OPEN at each iteration. Our experiments show that larger values of N generally lead to shorter solutions and evaluate more nodes per second than searches with smaller N . We call the combination of A* search with a path-cost weight λ and a batch size of N 'BWAS'.

To satisfy the theoretical bounds on how much the length of a solution will deviate from the length of an optimal solution, the heuristic used in the weighted A* search must be admissible. That is to say that the heuristic can never overestimate the cost to reach the goal. Although DeepCubeA's value function is not admissible, we empirically evaluate by how much DeepCubeA overestimates the cost to reach the goal. To do this, we obtain the length of a shortest path to the goal for 100,000 Rubik's cube states scrambled between 1 and 30 times. We then evaluate those same states with DeepCubeA's heuristic function j_θ . We find that DeepCubeA's heuristic function does not overestimate the cost to reach the goal 66.8% of the time and 97.4% of the time it does not overestimate it by more than one. The average overestimation of the cost is 0.24.

Neural network architecture. The first two hidden layers of the DNNs have sizes of 5,000 and 1,000, respectively, with full connectivity. These are then followed by four residual blocks²⁷, where each residual block has two hidden layers of size 1,000. Finally, the output layer consists of a single linear unit representing the cost-to-go estimate (Supplementary Fig. 3). We used batch normalization⁴⁰ and rectified linear activation functions⁴¹ in all hidden layers. The DNN was trained with a batch size of 10,000, optimized with ADAM⁴², and did not use any regularization. The maximum number of random moves applied to any training state K was set to 30. The error threshold ε was set to 0.05. We checked if the loss fell below the error threshold every 5,000 iterations. Training was carried out for 1 million iterations on two NVIDIA Titan V GPUs, with six other GPUs used in parallel for data generation. In total, the DNN saw 10 billion examples during training. Training was completed in 36 h. When solving scrambled cubes from the test set, we use four NVIDIA X Pascal GPUs in parallel to compute the cost-to-go estimate. For the 15 puzzle, 24 puzzle and Lights Out we set K to 500. For the 35 puzzle, 48 puzzle and Sokoban we set K to 1,000. For the 24 puzzle we use six residual blocks instead of four.

Comparison to multi-step lookahead update strategies. Instead of using equation (1), which may be seen as a depth-1 breadth-first search (BFS), to update the estimated cost-to-go function we experimented with a depth-2 BFS. To obtain a better perspective on how well DeepCubeA's learning procedure trains the given DNN, we also implemented an update strategy of trying to directly imitate the optimal cost-to-go function calculated using the handmade optimal solver²⁵ by minimizing the mean squared error between the output of the DNN and the oracle value provided by the optimal solver. We demonstrate that the DNN trained with DAVI achieves the same performance as a DNN with the same architecture trained with these update strategies. The performance obtained from a depth-2 BFS update strategy is shown in Supplementary Fig. 1. Although the final performance obtained with depth-2 BFS is similar to the performance obtained with depth-1 BFS, its computational cost is significantly higher. Even when using 20 GPUs in parallel for data generation (instead of six), the training time is five times longer for the same number of iterations. Supplementary Fig. 2 shows that the DNN trained to imitate the optimal cost-to-go function predicts the optimal cost-to-go more accurately than DeepCubeA for states scrambled 20 or more times. The figure also shows the performance on solving puzzles using a greedy best-first search with this imitated cost-to-go function suffers for states scrambled fewer than 20 times. We speculate that this is because imitating the optimal cost-to-go function causes the DNN to overestimate the cost to reach the goal for states scrambled fewer than 20 times.

Hyperparameter selection for BWAS. To choose the hyperparameters of BWAS, we carried out a grid search over λ and N . Values of λ were 0.0, 0.2, 0.4, 0.6, 0.8 and

1.0 and values of N were 1, 100, 1,000 and 10,000. The grid search was performed on 100 cubes that were generated separately from the test set. The GPU machines available to us had 64 GB of RAM. Hyperparameter configurations that reached this limit were stopped early and thus not included in the results. Supplementary Fig. 4 shows how λ and N affect performance in terms of average solution length, average number of nodes generated, average solve time and average number of nodes generated per second. The figure shows that as λ increases, the average solution length decreases; however, the time to find a solution typically increases as well. The results also show that larger values of N lead to shorter solution lengths, but generally also require more time to find a solution; however, the number of nodes generated per second also increases due to the parallelism provided by the GPUs. Because $\lambda = 0.6$ and $N = 10,000$ resulted in the shortest solution lengths, we use these hyperparameters for the Rubik's cube. For the 15 puzzle, 24 puzzle and 35 puzzle we use $\lambda = 0.8$ and $N = 20,000$. For the 48 puzzle we use $\lambda = 0.6$ and $N = 20,000$. We increased N from 10,000 to 20,000 because we saw a reduction in solution length. For Lights Out we use $\lambda = 0.2$ and $N = 1,000$. For Sokoban we use $\lambda = 0.8$ and $N = 1$.

PDBs. PDBs²⁶ are used to obtain a heuristic using lookup tables. Each lookup table contains the number of moves required to solve all possible combinations of a certain subgoal. For example, we can obtain a lookup table by enumerating all possible combinations of the edge cubelets on the Rubik's cube using a BFS. These lookup tables are then combined through either a max operator or a sum operator (depending on independence between subgoals)^{7,8} to produce a lower bound on the number of steps required to solve the problem. Features from different PDBs can be combined with neural networks for improved performance⁴³.

For the Rubik's cube, we implemented the PDB that Korf uses to find optimal solutions to the Rubik's cube⁷. For the 15 puzzle, 24 puzzle and 35 puzzle, we implement the PDBs described in Felner and other's work on additive PDBs⁹. To the best of our knowledge, no-one has created a PDB for the 48 puzzle. We create our own by partitioning the puzzle into nine subgoals of size 5 and one subgoal of size 3. For all the n puzzles, we also save the mirror of each PDB to improve the heuristic and map each lookup table to a representation of size p^k where p is the total number of puzzle pieces and k is the size of the subgoal. Although this uses more memory, this is done to increase the speed of the lookup table⁹. For the n puzzle, the optimal solver algorithm (IDA*⁴²) adds an additional optimization by only computing the location of the beginning state in the lookup table and then only computing offsets for each subsequently generated state.

Web server. We have created a web server, located at <http://deepcube.igb.uci.edu/>, to allow anyone to use DeepCubeA to solve the Rubik's cube. In the interest of speed, the hyperparameters for BWAS are set to $\lambda = 0.2$ and $N = 100$ in the server. The user can initiate a request to scramble the cube randomly or use the keyboard keys to scramble the cube as they wish. The user can then use the 'solve' button to have DeepCubeA compute and post a solution, and execute the corresponding moves. The basic web server's interface is displayed in Supplementary Fig. 5.

Conjugate patterns and symmetric states. Because the operation of the Rubik's cube is deeply rooted in group theory, solutions produced by an algorithm that learns how to solve this puzzle should contain group theory properties. In particular, conjugate patterns of moves of the form aba^{-1} should appear relatively often when solving the Rubik's cube. These patterns are necessary for manipulating specific cubelets while not affecting the positions of other cubelets. Using a sliding window, we gathered all triplets in all solutions to the Rubik's cube and found that aba^{-1} accounted for 13.11% of all triplets (significantly above random), while aba accounted for 8.86%, aab accounted for 4.96% and abb accounted for 4.92%. To put this into perspective, for the optimal solver, aba^{-1} , aba , aab and abb accounted for 9.15, 9.63, 5.30 and 5.35% of all triplets, respectively.

In addition, we found that DeepCubeA often found symmetric solutions to symmetric states. One can produce a symmetric state for the Rubik's cube by mirroring the cube from left to right, as shown in Fig. 4. The optimal solutions for two symmetric states have the same length; furthermore, one can use the mirrored solution of one state to solve the other. To see if this property was present in DeepCubeA, we created mirrored states of the Rubik's cube test set and solved them using DeepCubeA. The results showed that 58.30% of the solutions to the mirrored test set were symmetric to those of the original test set. Of the solutions that were not symmetric, 69.54% had the same solution length as the solution length obtained on the original test set. To put this into perspective, for the handmade optimal solver, the results showed that 74.50% of the solutions to the mirrored test set were symmetric to those of the original test set.

Data availability

The environments for all puzzles presented in this paper, code to generate labelled training data and initial states used to test DeepCubeA are available through a Code Ocean compute capsule (<https://doi.org/10.24437/CO.4958495.v1>)⁴⁴.

Received: 23 January 2019; Accepted: 7 June 2019;
Published online: 15 July 2019

References

1. Lichodziejewski, P. & Heywood, M. in *Genetic Programming Theory and Practice VIII* (eds Riolo, R., McConaghy, T. & Vladislavleva, E.) 35–54 (Springer, 2011).
2. Smith, R. J., Kelly, S. & Heywood, M. I. Discovering Rubik's cube subgroups using coevolutionary GP: a five twist experiment. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* 789–796 (ACM, 2016).
3. Brunetto, R. & Trunda, O. Deep heuristic-learning in the Rubik's cube domain: an experimental evaluation. *Proc. ITAT* **1885**, 57–64 (2017).
4. Johnson, C. G. Solving the Rubik's cube with learned guidance functions. In *Proceedings of 2018 IEEE Symposium Series on Computational Intelligence (SSCI)* 2082–2089 (IEEE, 2018).
5. Korf, R. E. Macro-operators: a weak method for learning. *Artif. Intell.* **26**, 35–77 (1985).
6. Arfae, S. J., Zilles, S. & Holte, R. C. Learning heuristic functions for large state spaces. *Artif. Intell.* **175**, 2075–2098 (2011).
7. Korf, R. E. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence* 700–705 (AAAI Press, 1997); <http://dl.acm.org/citation.cfm?id=1867406.1867515>
8. Korf, R. E. & Felner, A. Disjoint pattern database heuristics. *Artif. Intell.* **134**, 9–22 (2002).
9. Felner, A., Korf, R. E. & Hanan, S. Additive pattern database heuristics. *J. Artif. Intell. Res.* **22**, 279–318 (2004).
10. Bonet, B. & Geffner, H. Planning as heuristic search. *Artif. Intell.* **129**, 5–33 (2001).
11. Schmidhuber, J. Deep learning in neural networks: an overview. *Neural Netw.* **61**, 85–117 (2015).
12. Goodfellow, I., Bengio, Y., Courville, A. & Bengio, Y. *Deep Learning* Vol. 1 (MIT Press, 2016).
13. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* Vol. 1 (MIT Press, 1998).
14. Bellman, R. *Dynamic Programming* (Princeton Univ. Press, 1957).
15. Puterman, M. L. & Shin, M. C. Modified policy iteration algorithms for discounted Markov decision problems. *Manage. Sci.* **24**, 1127–1137 (1978).
16. Bertsekas, D. P. & Tsitsiklis, J. N. *Neuro-dynamic Programming* (Athena Scientific, 1996).
17. Hart, P. E., Nilsson, N. J. & Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* **4**, 100–107 (1968).
18. Pohl, I. Heuristic search viewed as path finding in a graph. *Artif. Intell.* **1**, 193–204 (1970).
19. Ebdend, R. & Drechsler, R. Weighted A* search—unifying view and application. *Artif. Intell.* **173**, 1310–1342 (2009).
20. McAleer, S., Agostinelli, F., Shmakov, A. & Baldi, P. Solving the Rubik's cube with approximate policy iteration. *Proceedings of International Conference on Learning Representations (ICLR)* (PMLR, 2019).
21. Silver, D. et al. A general reinforcement learning algorithm that masters chess, shogi and Go through self-play. *Science* **362**, 1140–1144 (2018).
22. Rokicki, T. *God's Number is 26 in the Quarter-turn Metric* <http://www.cube20.org/qtm/> (2014).
23. Korf, R. E. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.* **27**, 97–109 (1985).
24. Rokicki, T. *cube20* <https://github.com/rokicki/cube20src> (2016).
25. Rokicki, T., Kociemba, H., Davidson, M. & Dethridge, J. The diameter of the Rubik's cube group is twenty. *SIAM Rev.* **56**, 645–670 (2014).
26. Culberson, J. C. & Schaeffer, J. Pattern databases. *Comput. Intell.* **14**, 318–334 (1998).
27. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* 770–778 (IEEE, 2016).
28. Kociemba, H. *15-Puzzle Optimal Solver* <http://kociemba.org/themen/fifteen/fifteensolver.html> (2018).
29. Scherphuis, J. *The Mathematics of Lights Out* <https://www.jaapsch.net/puzzles/lomath.htm> (2015).
30. Dor, D. & Zwick, U. Sokoban and other motion planning problems. *Comput. Geom.* **13**, 215–228 (1999).
31. Guez, A. et al. *An Investigation of Model-free Planning: Boxoban Levels* <https://github.com/deepmind/boxoban-levels/> (2018).
32. Orseau, L., Lelis, L., Lattimore, T. & Weber, T. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems* (eds Bengio, S. et al.) 3201–3211 (Curran Associates, 2018).
33. Brügger, A., Marzetta, A., Fukuda, K. & Nievergelt, J. The parallel search bench ZRAM and its applications. *Ann. Oper. Res.* **90**, 45–63 (1999).
34. Korf, R. E. Linear-time disk-based implicit graph search. *JACM* **55**, 26 (2008).
35. Moore, A. W. & Atkeson, C. G. Prioritized sweeping: reinforcement learning with less data and less time. *Mach. Learn.* **13**, 103–130 (1993).
36. Newell, A. & Simon, H. A. *GPS, a Program that Simulates Human Thought* Technical Report (Rand Corporation, 1961).
37. Fikes, R. E. & Nilsson, N. J. STRIPS: a new approach to the application of theorem proving to problem solving. *Artif. Intell.* **2**, 189–208 (1971).
38. Anthony, T., Tian, Z. & Barber, D. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems* (eds Guyon, I. et al.) 5360–5370 (Curran Associates, 2017).
39. Wilt, C. M. & Rumel, W. When does weighted A* fail? In *Proc. SOCS* (eds Borrajo, D. et al.) 137–144 (AAAI Press, 2012).
40. Ioffe, S. & Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of International Conference on Machine Learning* (eds Bach, F. & Blei, D.) 448–456 (PMLR, 2015).
41. Glorot, X., Bordes, A. & Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (eds Gordon, G., Dunson, D. & Dudík, M.) 315–323 (PMLR, 2011).
42. Kingma, D. P. & Ba, J. Adam: a method for stochastic optimization. In *Proceedings of International Conference on Learning Representations (ICLR)* (eds Bach, F. & Blei, D.) (PMLR, 2015).
43. Samadi, M., Felner, A. & Schaeffer, J. Learning from multiple heuristics. In *Proceedings of the 23rd National Conference on Artificial Intelligence* (ed. Cohn, A.) (AAAI Press, 2008).
44. Agostinelli, F., McAleer, S., Shmakov, A. & Baldi, P. *Learning to Solve the Rubik's Cube* (Code Ocean, 2019); <https://doi.org/10.24433/CO.4958495.v1>

Acknowledgements

The authors thank D.L. Flores for useful suggestions regarding the DeepCubeA server and T. Rokicki for useful suggestions and help with the optimal Rubik's cube solver.

Author contributions

P.B. designed and directed the project. F.A., S.M. and A.S. contributed equally to the development and testing of DeepCubeA. All authors contributed to writing and editing the paper.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information is available for this paper at <https://doi.org/10.1038/s42256-019-0070-z>.

Reprints and permissions information is available at www.nature.com/reprints.

Correspondence and requests for materials should be addressed to P.B.

Publisher's note: Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature Limited 2019