A* SEARCH WITHOUT EXPANSIONS: LEARNING HEURISTIC FUNCTIONS WITH DEEP Q-NETWORKS

Forest Agostinelli
Department of Computer Science and Engineering
University of South Carolina
foresta@cse.sc.edu

Alexander Shmakov
Department of Computer Science
University of California, Irvine
ashmakov@uci.edu

Stephen McAleer
Department of Computer Science
University of California, Irvine
smcaleer@uci.edu

Roy Fox
Department of Computer Science
University of California, Irvine
royf@uci.edu

Pierre Baldi
Department of Computer Science
University of California, Irvine
pfbaldi@uci.edu

ABSTRACT

A* search is an informed search algorithm that uses a heuristic function to guide the order in which nodes are expanded. Since the computation required to expand a node and compute the heuristic values for all of its generated children grows linearly with the size of the action space, A* search can become impractical for problems with large action spaces. This computational burden becomes even more apparent when heuristic functions are learned by general, but computationally expensive, deep neural networks. To address this problem, we introduce DeepCubeAQ, a deep reinforcement learning and search algorithm that builds on the DeepCubeA algorithm and deep Q-networks. DeepCubeAQ learns a heuristic function that, with a single forward pass through a deep neural network, computes the sum of the transition cost and the heuristic value of all of the children of a node without explicitly generating any of the children, eliminating the need for node expansions. DeepCubeAQ then uses a novel variant of A* search, called AQ* search, that uses the deep Q-network to guide search. We use DeepCubeAQ to solve the Rubik's cube when formulated with a large action space that includes 1872 meta-actions and show that this 157-fold increase in the size of the action space incurs less than a 4-fold increase in computation time when performing AQ* search and that AQ* search is orders of magnitude faster than A* search.

1 Introduction

While the performance of A* search is often measured in the number of nodes it *expands*, the number of nodes it *generates* through the expansion process has a major impact on performance. Expanding a node requires that every possible action be applied to the state associated with that node, thereby generating new states and, subsequently, new nodes. After a node is generated, the A* search algorithm computes the heuristic value of this newly generated node using the heuristic function. Finally, each newly generated node is then pushed to a priority queue. Therefore, for each iteration of A* search, the number of new nodes generated, the number of applications of the heuristic function, and the number of nodes pushed to the priority queue increases linearly with the size of the action space. Furthermore, much of this computation eventually goes unused as, in general, A* search does not expand every node that it generates.

The need to reduce this linear increase in computational cost has become more relevant with the more frequent use of deep neural networks (DNNs) [24] as heuristic functions. While DNNs are universal function approximators [16], they are computationally expensive when compared to heuristic functions based on domain knowledge, human intuition, and pattern databases [10]. Nonetheless, DNNs are able to learn heuristic functions to solve problems ranging from puzzles [8, 3, 20, 1], to quantum computing [31], to chemical synthesis [7], while making very few assumptions about the structure of the problem. Due to their flexibility and ability to generalize, DNNs offer the promise of learning heuristic functions in a largely domain-independent fashion. Removing the linear increase in computational cost as a function of

the size of the action space would make DNNs practical for a wide range of applications with large action spaces such as multiple sequence alignment, theorem proving, program synthesis, and molecular optimization.

The DeepCubeA algorithm used deep reinforcement learning and A* search to solve puzzles such as the Rubik's cube [1] and has since been used in applications such as quantum computing [31] and cryptography [17]. DeepCubeA uses approximate value iteration [5] to train a heuristic function without relying on domain specific heuristic information. However, the computation required for approximate value iteration also grows linearly with the size of the action space. Therefore, the practicality of both the training phase and the search phase of DeepCubeA reduces as the size of the action space increases.

While previous attempts at reducing the number of generated nodes involve using domain knowledge to partially expand a node by generating only a subset of its children [12], in this work, we introduce the DeepCubeAQ algorithm, which removes the need to expand nodes, entirely. DeepCubeAQ uses Q-learning to train a deep Q-network (DQN) which is then integrated with A* search to create an algorithm we call AQ*. DQNs are DNNs that map a single state to the sum of the transition cost and the heuristic value for each of its successor states. This removes the need for node expansions as we can obtain the information needed to estimate the cost of all children with a single forward pass through a DQN. In addition, the number of times the heuristic function must be applied is constant with respect to the size of the action space instead of linear. To track actions, each node is also associated with an action, which gets applied to its corresponding state when removed from the priority queue. As a result, the only aspect of AQ* search that depends on the action space is pushing a node, along with each of its possible actions, to the priority queue. This is also more memory efficient than explicitly generating all children nodes as, in our implementation, each action is only an integer.

We show that DeepCubeAQ can train a heuristic function and use search to solve problems orders of magnitude faster than DeepCubeA. DeepCubeAQ also generates orders of magnitude fewer nodes than DeepCubeA, thus using less memory. As the size of the action space grows, DeepCubeA becomes impractical while DeepCubeAQ sees only a relatively small increase in runtime and memory usage.

2 Related Work

Partial expansion A* search (PEA*) [29] was proposed for problems with large action spaces. PEA* first expands a node by generating all of its children, however, it only keeps the children whose cost is below a certain threshold. It then adds a bookkeeping structure to remember the highest cost of the discarded nodes. The intention of PEA* is to save memory, however, the computational requirements do not reduce as every node has to be expanded and the heuristic function has to be applied to all of its children. Enhanced partial expansion A* search (EPEA*) [12] uses a domain-dependent operator selection function to only generate a subset of children based on their cost.

DNNs have been used to address large action spaces in adversarial search. In particular, when combined with Monte-Carlo tree search (MCTS) [9], DNNs with multi-dimensional outputs that represent policies have been used to learn to play Go [25], as well as chess and Shogi [26], at superhuman levels. In addition to adversarial search, MCTS combined with DNNs has been applied to combinatorial optimization problems [19] and hard Sokoban problems [13]. In the original DeepCube algorithm, MCTS was combined with DNNs to solve the Rubik's cube [20]. However, DeepCubeA showed that A* search solves the Rubik's cube using shorter solutions in less time than MCTS [1].

3 Preliminaries

3.1 Deep Approximate Value Iteration

Value iteration [23] is a dynamic programming algorithm and a central algorithm in reinforcement learning [4, 5, 27] that iteratively improves a cost-to-go function J, that estimates the cost to go to from a given state to the closest goal state via a shortest path. This cost-to-go function can be readily used as a heuristic function for A* search.

In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored for each possible state s. Value iteration loops through each state s and computes an updated $J'(s)$ using the Bellman equation:

$$J'(s) = \min_a \sum_{s'} P^a(s, s')(g^a(s, s') + \gamma J(s')) \tag{1}$$

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 transition cost, the cost associated with transitioning from state s to s' by taking action a; and

$\gamma \in [0, 1]$ is the discount factor. The action a is an action in a set of actions $\mathcal{A}$. Value iteration has been shown to converge to the optimal cost-to-go, j_* [6]. That is, $j_*(s)$ will return the cost of a shortest path to the closest goal state for every state s.

In the case of the Rubik's cube, all transitions are deterministic. This is also generally the case for problems solved with A* search. Therefore, we can simplify Equation 1 as follows:

$$J'(s) = \min_a (g^a(s, A(s, a)) + \gamma J(A(s, a))) \tag{2}$$

where $A(s, a)$ is the state obtained from taking action a in state s. We set all transition costs to 1 and set γ to 1 for all experiments.

However, representing J as a lookup table is too memory intensive for problems with large state spaces. For instance, the Rubik's cube has 4.3×10^{19} possible states. Therefore, we turn to approximate value iteration [5] where J is represented as a parameterized function, j_θ, with parameters θ. We choose to represent j_θ as a deep neural network (DNN). The parameters θ are learned by using stochastic gradient descent to minimize the following loss function:

$$L(\theta) = (\min_a g^a(s, A(s, a)) + \gamma j_{\theta^-}(A(s, a)) - j_\theta)^2 \tag{3}$$

Where θ^- are the parameters for the "target" DNN that is used to compute the updated cost-to-go. Using a target DNN has been shown to result in a more stable training process. The parameters θ^- are periodically updated to θ during training. While we cannot guarantee convergence to j_*, approximate value iteration has been shown to approximate j_* [5]. This combination of deep neural networks and approximate value iteration is referred to as deep approximate value iteration (DAVI).

3.2 A* Search

A* search [14] is an informed search algorithm which finds a path between the node, n_0, associated with the start state, s_0, and a node, n_g, associated with a goal state, s_g, in a set of goal states. A* search maintains a priority queue, OPEN, from which it iteratively removes and expands the node with the lowest cost and a dictionary, CLOSED, that maps nodes that have already been generated to their path costs. The cost of each node is $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost, the distance between n_0 and n, and $h(n)$ is the heuristic value, the estimated distance between the state associated with n and the nearest goal state. After a node is expanded, its children that are not already in CLOSED are added to CLOSED and then pushed to OPEN. If a child node n is already in CLOSED, but a less costly path from n_0 to n has been encountered, then the path cost of n is updated in CLOSED and n is added to OPEN. The algorithm starts with only n_0 in OPEN and terminates when the node associated with a goal state is removed from OPEN. In this work, the heuristic function is a DNN j_θ. Pseudocode for the A* search algorithm is given in Algorithm 1.

3.3 DeepCubeA

DeepCubeA [1] learns a cost-to-go function j_θ using DAVI and then uses that function as a heuristic function for A* search. The states used for training j_θ with DAVI are generated by randomly scrambling the goal state between 0 and K times. This allows learning to propagate from the goal state to all other states in the training set.

Because the learned heuristic function is a DNN, computing heuristic values can make A* search computationally expensive. To alleviate this issue, one can take advantage of the parallelism provided by graphics processing units (GPUs) by expanding the N lowest cost nodes and computing their heuristic values in parallel. Furthermore, even with a computationally cheap and informative heuristic, A* search can be both time and memory intensive. To address this, one can trade potentially more costly solutions with potentially faster runtimes and less memory usage with a variant of A* search called weighted A* search [22]. Weighted A* search computes the cost of each node as $f(n) = \lambda g(n) + h(n)$ where $\lambda \in [0, 1]$ is a scalar weighting. This combination of expanding N nodes at every iteration and weighting the path cost by λ is referred to as batch-weighted A* search (BWAS).

Algorithm 1 A* Search

Input: starting state s_0, DNN j_θ
OPEN $\leftarrow$ priority queue
CLOSED $\leftarrow$ dictionary that maps nodes to path costs
$n_0 = \text{NODE}(s = s_0, g = 0)$
$c_0 = j_\theta(s_0)$
Push n_0 to OPEN with cost c_0
while not IS_EMPTY(OPEN) **do**
  $n = \text{POP(OPEN)}$
  $s = n.s$
  if IS_GOAL(s) **then**
    return PATH_TO_GOAL(n)
  end if
  for a in 0 to $|\mathcal{A}|$ **do**
    $s' = A(s, a)$
    $g' = g^a(s, s') + n.g$
    $n' = \text{NODE}(s = s', g = g')$
    if n' not in CLOSED or $g' <$ CLOSED$[n']$ **then**
      CLOSED$[n'] = g'$
      $c' = j_\theta(s')$
      Push n' to OPEN with cost c'
    end if
  end for
end while
return failure

4 Methods

4.1 Q-learning

Instead of learning a function, j_θ, that maps a state, s, to its cost-to-go, one can learn a function, q_ϕ, that maps s to its Q-factors, which is a vector containing $Q(s, a)$ for all actions a [6]. In a deterministic environment, the Q-factor is defined as:

$$Q(s, a) = g^a(s, A(s, a)) + \gamma J(A(s, a)) \tag{4}$$

where $J(A(s, a))$ can be expressed in terms of Q with $J(A(s, a)) = \min_{a'} Q(A(s, a), a')$. Learning Q by iteratively updating the left-hand side of (4) toward its right-hand side is known as Q-learning [28]. Like for DAVI, Q is represented as a parameterized function, q_ϕ, and we choose a deep neural network for q_ϕ. This is also known as a deep Q-network (DQN) [21]. The architecture of the DQN is constructed such that the input is the state, s, and the output is a vector that represents $q_\phi(s, a)$ for all actions a. The parameters ϕ are learned using stochastic gradient descent to minimize the loss function:

$$L(\phi) = (g^a(s, A(s, a)) + \gamma \min_{a'} q_{\phi^-}(A(s, a), a') - q_\phi(s, a))^2 \tag{5}$$

Just like in DAVI, the parameters ϕ^- of the target DNN are periodically updated to ϕ during training.

Similar to value iteration, Q-learning has been shown to converge to the optimal Q-factors, q_*, in the tabular case [28]. In the approximate case, Q-learning has a computational advantage over DAVI because, while the number of parameters of the DQN grows with the size of the action space, the number of forward passes needed to compute the loss function stays constant for each update. We will show in our results that, in large action spaces, the training time for Q-learning can be up to 127 times faster than DAVI.

4.2 AQ* Search

We present AQ* search, a search algorithm that builds on A* search to take advantage of DQNs. AQ* search uses tuples containing a node and an action, which we will refer to as node_actions, to search for a path to the goal. The cost

Algorithm 2 AQ* Search

Input: starting state s_0, DQN q_ϕ
OPEN $\leftarrow$ priority queue
CLOSED $\leftarrow$ dictionary that maps nodes to path costs
$n_0 =$ NODE$(s = s_0, g = 0)$
$a =$ NO_OP
$c_0 = \min_{a'} q(s_0, a')$
Push (n_0, a) to OPEN with cost c_0
while not IS_EMPTY(OPEN) **do**
  $(n, a) =$ POP(OPEN)
  $s = n.s$
  $s' = A(s, a)$
  if IS_GOAL(s') **then**
    return PATH_TO_GOAL(n, a)
  end if
  $g' = n.g + g^a(s, s')$
  $n' =$ NODE$(s = s', g = g')$
  if n' not in CLOSED or $g' <$ CLOSED$[n']$ **then**
    CLOSED$[n'] = g'$
    $\mathbf{q} = q_\phi(s', .)$
    for a' in 0 to $|\mathcal{A}|$ **do**
      $c' = g' + \mathbf{q}[i]$
      Push (n', a') to OPEN with cost c'
    end for
  end if
end while
return failure

of a node_action with node n and action a is $g(n) + g^a(s, s') + h(n')$ where s is the state associated with node n and $s' = A(s, a)$ is the successor of s when taking action a, and s' is associated with node n'. The Q-factor $Q(s, a)$ is equal to $g^a(s, s') + h(n')$. Therefore, using DQNs, the transition cost and cost-to-go of all child nodes can be computed using q_ϕ without having to expand n.

At every iteration, AQ* search pops a node_action, (n, a), from OPEN, creates a new state, $s' = A(s, a)$, by applying action a to the state s associated with n, and generates a new node n' with corresponding state s'. Instead of expanding n', AQ* search applies the DQN to s' to obtain the sum of the transition cost and the cost-to-go for all of its children. AQ* search then pushes the new node_actions $(n',\ a')$ to OPEN for all actions a' where the cost is computed by summing the path cost of n and the output of the DQN corresponding to action a'. In AQ* search, the only part that depends on the size of the action space is pushing nodes to OPEN. Unlike A* search, no nodes are expanded and the heuristic function only needs to be applied once per iteration. Pseudocode for the AQ* search algorithm is given in Algorithm 2. In our results, we will show that AQ* search is orders of magnitude faster than A* search.

4.3 DeepCubeAQ

DeepCubeAQ uses the same process as DeepCubeA to generate states for training. Instead of DAVI, DeepCubeAQ uses Q-learning to train a DQN. This DQN is then used in AQ* search. Like DeepCubeA, DeepCubeAQ also performs AQ* in batches of size N and with a weight λ. We also note that the cost of a node_action for the weighted version of AQ* is $\lambda g(n) + g^a(s, s') + h(n')$. However, λ should also be applied to the transition cost $g^a(s, s')$. This does not happen because the transition cost is not computed separately from the cost-to-go. In this work, all the transition costs are the same, therefore, this constant offset does not affect the order in which nodes are expanded. However, in future work, this could be remedied by training a DQN that separates the transition cost and the cost-to-go of the subsequent state.

5 Results

The Rubik's cube action space has 12 different actions: each of the six faces can be turned clockwise or counterclockwise. We refer to this action space as RC(12). To determine how well DeepCubeAQ can find solutions in large action spaces, we add meta-actions to the Rubik's cube action space, creating RC(156) and RC(1884). RC(156) has all the actions

in RC(12) plus all combinations of actions of size two (144). RC(1884) has all the actions in RC(156) plus all combinations of actions of size three (1728). To ensure none of these additional meta-actions are redundant, the cost for all meta-actions is also set to one. The machines we use for training and search have 48 2.4 GHz Intel Xeon central processing units (CPUs), 190 GB of random access memory, and two 32GB NVIDIA V100 GPUs.

We train the cost-to-go function with the same architecture described in [1], which has a fully connected layer of size 5,000, followed by another fully connected layer of size 1,000, followed by four fully connected residual blocks of size 1,000 with two hidden layers per residual block [15]. The DQN also has the same architecture with the exception that the output is a vector that estimates the cost-to-go for taking every possible action. For generating training states, the number of times we scramble the Rubik's cube, K, is set to 30. For Q-learning, we select actions according to a Boltzmann distribution where each action a is selected with probability:

$$p_{s,a} = \frac{e^{(q_\phi(s,a)/T)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(q_\phi(s,a')/T)}} \tag{6}$$

where we set the temperature $T = \frac{1}{3}$.

We train with a batch size of 10,000 for 1.2 million iterations using the ADAM optimizer [18]. We update the target networks with the same schedule defined in the DeepCubeA source code [2]. However, as the size of the action space increases, training with a batch size this large becomes infeasible for DeepCubeA. Table 1 shows that DeepCubeA would take over a month to train on RC(156) and almost a year to train on RC(1884). Therefore, we reduce the batch size in proportion to the differences in the size of the action space with RC(12). Since RC(156) has 13 times more actions, we train DeepCubeA with a batch size of 769 and since RC(1884) has 157 times more actions, we train DeepCubeA with a batch size of 63.

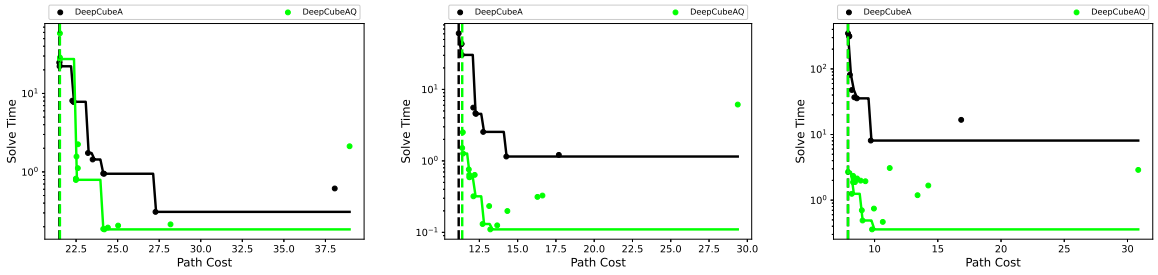
Table 1: The table shows the number of training iterations per second with a batch size of 10,000 and the projected number of days to train for 1.2 million iterations. DeepCubeA (DCA) is significantly slower than DeepCubeAQ (DCQ), especially when the size of the action space is large.

Puzzle	Method	Itrs/Sec	Train Time
RC(12)	DCA	3.96	3.5d
	DCQ	**8.55**	**1.6d**
RC(156)	DCA	0.42	33d
	DCQ	**7.46**	**1.9d**
RC(1884)	DCA	0.04	347d
	DCQ	**5.08**	**2.7d**

5.1 Performance During Search

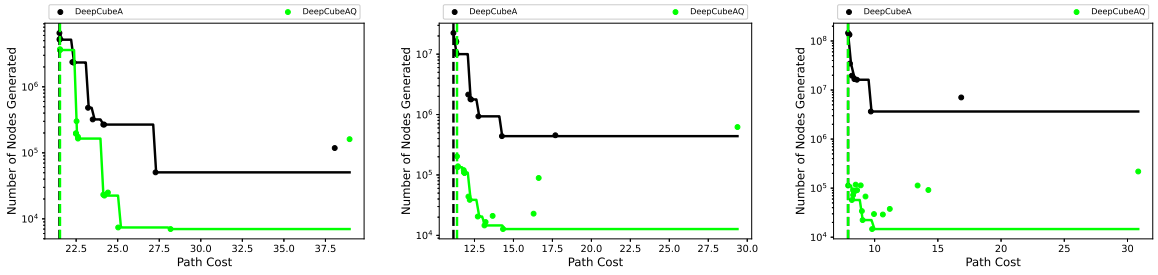
The original work on DeepCubeA used $\lambda = 0.6$ and $N = 10000$ for BWAS. However, since these search parameters create a tradeoff between speed, memory usage, and path cost, we also examine the performance with different parameter settings to understand how A* search and AQ* search compare along these dimensions. Therefore, we try all combinations of $\lambda \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$ and $N \in \{100, 1000, 10000\}$. For each method and each action space, we prune all combinations that cause our machine to run out of memory or that require over 24 hours to complete. We use the same 1,000 test states used in [1], where each test state was obtained by scrambling the Rubik's cube between 1,000 and 10,000. Due to the increase in solving time from DeepCubeA, for RC(156), we use a subset of 100 states and for RC(1884), we use a subset of 20 states. We examine the results both in terms of the lowest path cost averaged over all the test states as well as hypothetical cases where there is some threshold for an acceptable average path cost.

The results show that, as the size of the action space increases, the speed and memory advantages of DeepCubeAQ becomes significant. In the largest action space, RC(1884), the lowest average path cost is the same for DeepCubeA and DeepCubeAQ, however, DeepCubeAQ is orders of magnitude faster and more memory efficient. In the case of RC(12), the lowest average path cost that DeepCubeA finds is 21.493 while for DeepCubeAQ it is 21.529. This is on par with the original DeepCubeA results that obtained an average path cost of 21.50. In the case of RC(156), the lowest average path cost that DeepCubeA finds is 11.15 while for DeepCubeAQ it is 11.38. Finally, in the case of RC(1884), the lowest average path cost for both DeepCubeA and DeepCubeAQ is 7.9. The average cost becomes smaller as the size of the action space increases because the additional meta-actions allow the Rubik's cube to be solved in fewer moves.



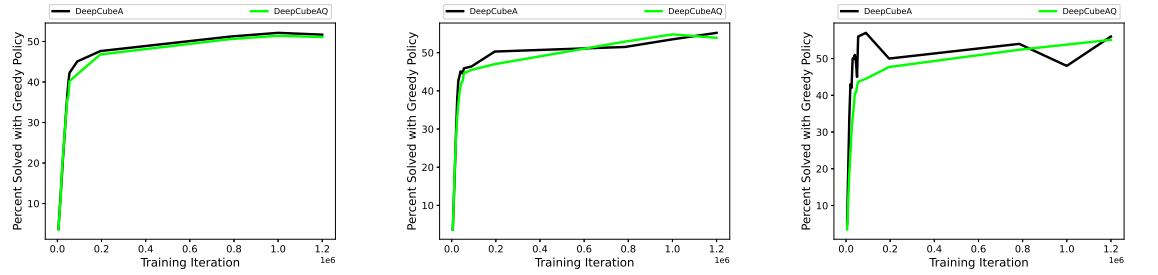
(a) RC(12) (b) RC(156) (c) RC(1884)

Figure 1: The figures shows the relationship between the average path cost and the average time to find a solution. The y-axis has a logarithmic scale. Each dot represents a search parameter setting. The dashed line represents the lowest average path cost found. The solid line represents the fastest solution time as a function of a hypothetical threshold for an acceptable average path cost. For almost all thresholds, DeepCubeAQ is significantly faster than DeepCubeA. For RC(1884), DeepCubeAQ is up to 129 times faster than DeepCubeA.



(a) RC(12) (b) RC(156) (c) RC(1884)

Figure 2: The figures shows the relationship between the average path cost and the average number of nodes generated. The y-axis has a logarithmic scale. Each dot represents a search parameter setting. The dashed line represents the lowest average path cost found. The solid line represents the fewest number of nodes generated as a function of a hypothetical threshold for an acceptable average path cost. For almost all thresholds, DeepCubeAQ generates significantly fewer nodes than DeepCubeA. For RC(1884), DeepCubeAQ generates up to 1288 times fewer nodes than DeepCubeA.



(a) RC(12) (b) RC(156) (c) RC(1884)

Figure 3: Percentage of training states solved with a greedy policy as a function of training iteration.

In Figure 1, we can see the relationship between the average path cost and the time to find a solution for all search parameter settings. In Figure 2, we can see the relationship between the average path cost and the number of nodes generated for all search parameter settings. The figures show that, for almost any possible path cost threshold, DeepCubeAQ is significantly faster and generates significantly fewer nodes than DeepCubeA. In the most extreme case, the cheapest average path cost for DeepCubeA and DeepCubeAQ is identical for RC(1884), however, DeepCubeAQ is 129 times faster and generates 1228 times fewer nodes than DeepCubeA. These ratios for different path costs thresholds are show in Tables 2, 3, and 4.

When comparing DeepCubeA and DeepCubeAQ to themselves for different action spaces, Table 5 shows that, though RC(1884) has 157 times more actions than RC(12), DeepCubeAQ only takes 3.7 times as long to find a solution and generates only 2.3 times as many nodes. On the other hand, in this same scenario, DeepCubeA takes 37 times as long and generates 62.7 times as many nodes.

Table 2: The ratio between DeepCubeA and DeepCubeAQ for the solution time and number of nodes generated for hypothetical acceptable path cost thresholds for RC(12). DeepCubeA only outperforms DeepCubeAQ in one instance.

	Path Cost Thresh			
	22	24	26	30
Time	0.8	1.8	5.1	1.7
Nodes Gen	1.4	1.9	36.2	6.8

Table 3: The ratio between DeepCubeA and DeepCubeAQ for the solution time and number of nodes generated for hypothetical acceptable path cost thresholds for RC(156). DeepCubeAQ outperforms DeepCubeA in all instances.

	Path Cost Thresh			
	22	24	26	30
Time	51.5	23.1	10.5	10.5
Nodes Gen	92.0	64.0	34.5	34.5

Table 4: The ratio between DeepCubeA and DeepCubeAQ for the solution time and number of nodes generated for hypothetical acceptable path cost thresholds for RC(1884). DeepCubeAQ is up to 129 times faster and generates up to 1288 times fewer nodes.

	Path Cost Thresh			
	22	24	26	30
Time	129.7	38.2	28.5	72.7
Nodes Gen	1288.4	342.3	282.8	725.0

Table 5: The table shows each puzzle with an action space augmented with meta-actions along with the ratio between the size of the action space and the size of the action space for RC(12). It then shows the ratio for the time and number of nodes generated for each method when compared to its own performance on RC(12) averaged over all search parameter settings with the standard deviation in parenthesis. DeepCubeAQ has much better performance for both metrics. For RC(156) DeepCubeAQ finds solutions in even less time than it did for RC(12) due to the addition of meta-actions.

Puzzle	Actions	Method	Time	Nodes Gen
RC(156)	x13	DCA	3.5(1.6)	8.7(2.2)
		DCQ	**0.9(0.7)**	**1.4(1.3)**
RC(1884)	x157	DCA	37.0(6.5)	62.7(5.2)
		DCQ	**3.7(4.0)**	**2.3(3.6)**

5.2 Performance During Training

To monitor performance during training, we track the percentage of states that are solved by simply behaving greedily with respect to the cost-to-go function. We generate these states the same way we generate the training states. Figure 3 shows this metric as a function of training time. The results show that, in the case of RC(12), DeepCubeA is slightly better than DeepCubeAQ. In RC(156) and RC(1884), even though the batch size for DeepCubeA is smaller, the performance is on par with DeepCubeAQ. This may be due to the fact that DeepCubeA is only learning the cost-to-go for a single state while DeepCubeAQ must learn the sum of the transition cost and cost-to-go for all possible next states.

6 Discussion

As the size of the action space increases, DeepCubeAQ becomes significantly more effective than DeepCubeA in terms of training time, solution time, and the number of nodes generated. In the largest action space, DeepCubeAQ is orders of magnitude faster and generates orders of magnitude fewer nodes than DeepCubeA while finding solutions with the same average path cost. For smaller action spaces, while DeepCubeAQ is almost always faster and more memory efficient, DeepCubeA is capable of finding solutions that are slightly cheaper than DeepCubeAQ. This could be due to the difference in what j_θ and q_ϕ are being trained to do. Since the forward pass performed by the DQN, q_ϕ, is the same as doing a one-step lookahead with j_θ, perhaps learning is more difficult for the DQN. However, since training and search are significantly faster for DeepCubeAQ, this gap could be closed with longer training times and searching with larger values of λ or N.

While the DQN used in this work was for a fixed action space, modifying architectures such as graph convolutional policy networks [30] to estimate Q-factors would allow DeepCubeAQ to be used on problems with variable action spaces. This would have a direct application to problems with large, but variable, action spaces such as chemical synthesis, molecular optimization, theorem proving, program synthesis, and web navigation.

A* search is guaranteed to find a shortest path if the heuristic function is admissible. An admissible heuristic function is a function that never overestimates the cost of a shortest path. While there is little work on learning admissible heuristic functions using nonlinear function approximators [11], future work can extend these theoretical guarantees to AQ*.

7 Conclusion

The DeepCubeAQ algorithm uses Q-learning to train a DQN to be used as a heuristic function in a novel variant of A* search called AQ* search. AQ* search eliminates the need for node expansions. When increasing the size of the action space by 157 times, AQ* search only takes 3.7 times as long and generates only 2.3 times more nodes. When compared to A* search, AQ* search is up to 129 times faster and generates up to 1288 times fewer nodes. The ability that DeepCubeAQ has to efficiently scale up to large action spaces could play a significant role in finding solutions to many important problems with large action spaces.

References

[1] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. DeepcubeA. `https://github.com/forestagostinelli/DeepCubeA`, 2020.

[3] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[4] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996. ISBN 1-886529-10-8.

[6] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 1995.

[7] Binghong Chen, Chengtao Li, Hanjun Dai, and Le Song. Retro*: learning retrosynthetic planning with neural guided A* search. In *International Conference on Machine Learning*, pages 1608–1616. PMLR, 2020.

[8] Hung-Che Chen and Jyh-Da Wei. Using neural networks for evaluation in heuristic search algorithm. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[9] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[10] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[11] Marco Ernandes and Marco Gori. Likely-admissible and sub-symbolic heuristics. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 613–617. Citeseer, 2004.

[12] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan Sturtevant, Jonathan Schaeffer, and Robert Holte. Partial-expansion A* with selective node generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 2012.

[13] Dieqiao Feng, Carla P Gomes, and Bart Selman. A novel automated curriculum strategy to solve hard sokoban planning instances. *Advances in Neural Information Processing Systems*, 33, 2020.

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[17] Jungha Jin and Keecheon Kim. 3D cube algorithm for the key generation method: Applying deep neural network learning-based. *IEEE Access*, 8:33689–33702, 2020.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:1807.01672*, 2018.

[20] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube without human knowledge. 2019.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4):193–204, 1970.

[23] Martin L Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[24] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[25] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[28] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[29] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with partial expansion for large branching factor problems. In *AAAI/IAAI*, pages 923–929, 2000.

[30] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. *arXiv preprint arXiv:1806.02473*, 2018.

[31] Yuan-Hang Zhang, Pei-Lin Zheng, Yi Zhang, and Dong-Ling Deng. Topological quantum compiling with reinforcement learning. *Physical Review Letters*, 125(17):170501, 2020.