Q* Search: Heuristic Search with Deep Q-Networks

Forest Agostinelli [1], Shahaf S. Shperberg [2], Alexander Shmakov [3], Stephen McAleer [4], Roy Fox [3], Pierre Baldi [3]

[1]University of South Carolina, Columbia, South Carolina, USA
[2]Ben-Gurion University of the Negev, Beer Sheva, Israel
[3]University of California, Irvine, California, USA
[4]Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Abstract

Efficiently solving problems with large action spaces using A* search has been of importance to the artificial intelligence community for decades. This is because the computation and memory requirements of A* search grow linearly with the size of the action space. This burden becomes even more apparent when A* search uses a heuristic function learned by computationally expensive function approximators, such as deep neural networks. To address this problem, we introduce Q* search, a search algorithm that uses deep Q-networks to guide search in order to take advantage of the fact that the sum of the transition costs and heuristic values of the children of a node can be computed with a single forward pass through a deep Q-network without explicitly generating those children. This significantly reduces computation time and requires only one node to be generated per iteration. We use Q* search on different domains and action spaces, showing that Q* suffers from only a small runtime overhead as the action size increases. In addition, our empirical results show Q* search is up to 129 times faster and generates up to 1288 times fewer nodes than A* search. Finally, although obtaining admissible heuristic functions from deep neural networks is an ongoing area of research, we prove that Q* search is guaranteed to find a shortest path given a heuristic function does not overestimate the sum of the transition cost and cost-to-go of the state.

Introduction

A* search (Hart, Nilsson, and Raphael 1968) is an algorithm that searches for a sequence of actions that forms a path between a given start state and a give goal, where a goal is a set of goal states. By maintaining a search tree consisting of nodes that represent states and edges that represent transitions between states, search is performed by expanding nodes in this search tree where nodes are prioritized for expansion according to a given cost. The expansion of a node and the computation of the cost of its child nodes is one of the most time-consuming portions of A* search. Node expansion is performed by applying every possible action to the state associated with a given node to generate the child nodes. The cost of a node is computed by adding its path cost plus to its heuristic value, where the path cost is the sum of transition costs from the start node to the given node and the heuristic value is computed by a heuristic function that estimates the cost to go from the state associated with the node to a closest goal state, commonly referred to as the cost-to-go. Since a single iteration of A* search entails removing a node from the priority queue, expanding the node, computing the cost of its children, and pushing its children to the priority queue, 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. This escalating computational load can be substantial, particularly considering that the evaluation of heuristic functions can be computationally intensive. Moreover, in numerous domains, the process of generating states can also be time-consuming, notably in motion-planning and chemical synthesis. However, it is worth noting that much of this computational effort might be redundant, as A* search typically does not expand every single node 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) (Schmidhuber 2015) as heuristic functions. While DNNs are universal function approximators (Hornik, Stinchcombe, and White 1989), they are computationally expensive when compared to heuristic functions based on domain knowledge, human intuition, and pattern databases (Culberson and Schaeffer 1998). Nonetheless, DNNs are able to learn heuristic functions to solve problems ranging from puzzles (Chen and Wei 2011; Arfaee, Zilles, and Holte 2011; McAleer et al. 2019; Agostinelli et al. 2019), to quantum computing (Zhang et al. 2020), to chemical synthesis (Chen et al. 2020), 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 chemical synthesis.

In this paper, we introduce Q* search, a search algorithm guided by a deep Q-network (DQN) (Mnih et al. 2015) that requires only one node to be generated per iteration. 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 allows us to only generate one node per iteration as we can store tuples of nodes and actions in a priority queue whose priority is determined by the DQN. When removing a tuple of a node and action from the queue, we can then generate a new node by applying the action to the state associated with that node. 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. As a result, the only aspect of Q* search that depends on the action space is pushing a node, along with each of the possible actions that can be applied to it, to the priority queue. This is also more memory efficient than explicitly generating all child nodes as, in our implementation, each action is only an integer. Our theoretical results show that Q* search is guaranteed to find a shortest path given a heuristic function that neither overestimates the cost of a shortest path nor underestimates the transition cost. Our experimental results on the Rubik's cube, Lights Out, and 35-Pancake puzzle show that Q* search is orders of magnitude faster than A* search and generates orders of magnitude fewer nodes. While these environments have a fixed action space, the Q* search algorithm is agnostic to whether the action space is fixed or dynamic. In the Discussion section, we discuss avenues for future work that use structured prediction to create DQNs that are applicable to variable action spaces.

Related Work

Partial expansion A* search (PEA*) (Yoshizumi, Miura, and Ishida 2000) 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 removed from the priority queue has to be expanded and the heuristic function has to be applied to all of its children. Notably, PEA* is orthogonal to Q* and can be applied in conjunction with it to further reduce memory consumption. Enhanced partial expansion A* search (EPEA*) (Felner et al. 2012) uses a domain-dependent operator selection function to generate only a subset of children based on their cost. However, when utilizing function approximations such as neural networks as heuristic functions, where the change in heuristic values resulting from action application is not predetermined, EPEA* becomes inapplicable.

Deferred heuristic evaluation (Helmert 2006) has been used to generate only one node per iteration in A* search. This is accomplished by assigning each child node the same heuristic value as its parent node and deferring the evaluation of the child nodes until they are removed from the priority queue. However, this comes at a cost of inaccuracy, especially when the cost-to-go of a child node can be drastically different than that of its parent. We compare Q* to this method in our experiments and show that, in the vast majority of cases, Q* search finds lower-cost solutions and does so much faster than deferred heuristic evaluation. Furthermore, in our experiments, deferred heuristic evaluation sometimes runs out of memory due to its inability to prioritize one child node over another.

Preliminaries

A search problem instance, denoted as $I = (G, c, start, goal, h)$, is comprised of a graph $G = (V, E)$ with states (vertices) in V and edges (transitions) in $E \subseteq V \times V$, and a cost function $c : E \rightarrow \mathbb{R}^+$ that assigns costs to graph edges. The instance specifies a starting state ($start$) and a target state ($goal$) or a predicate $P : V \rightarrow 0, 1$ indicating whether a state satisfies goal conditions. h is a heuristic function which assigns to each state s an estimate of the cost associated with the shortest path leading from s to a nearest goal state, often termed the "cost-to-go". The primary objective is to discover a path within graph G that connects $start$ to $goal$. The quality of the derived path is the cumulative cost of its constituent edges, determined by the cost function. We denote by $d(s, s')$ the shortest (cheapest) path between S and s' in G, and $d(start, goal)$ by C*.

Note that the graph is typically given implicitly, where only the initial state is given alongside a set of transition functions $\mathcal{A}$. These functions represent various transitions such as those between different robot configurations, puzzle permutations, or STRIPS-like states in domain-independent planning problems. In this work, we adhere to this assumption, meaning that we are provided with an *action space* $\mathcal{A}$, and we assume that the set of edges E corresponds to applying each action $a \in \mathcal{A}$ from every state s. We denote the state resulted by applying action a from state s by $A(s, a)$ and the corresponding transition cost by $c^a(s)$.

Deep Approximate Value Iteration

Value iteration (Puterman and Shin 1978) is a dynamic programming algorithm that is central in solving Markov Decision Processes (MDPs) and reinforcement learning problems (Bellman 1957; Bertsekas and Tsitsiklis 1996; Sutton and Barto 1998). It iteratively computes the expected value for each state, initially assuming a zero value for all states and progressively refining these estimations by applying a Bellman update. Value iteration is typically formulated for maximization problems featuring stochastic action effects and a potentially infinite planning horizon. However, in the realm of heuristic search, it can be redefined to address minimization problems, specifically aimed at minimizing costs, with deterministic effects and a finite planning horizon. This redefinition is expressed by the following equation:

$$V'(s) = \min_{a \in \mathcal{A}} (c^a(s) + V(A(s, a))) \tag{1}$$

In this context, V represents a cost-to-go function, estimating the cost of reaching the closest goal state from a given state via the shortest path. This cost-to-go function can seamlessly serve as a heuristic function for A* search.

Nonetheless, representing V 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 (Bertsekas and Tsitsiklis 1996) where V is represented as a parameterized function, v_θ, with parameters θ. We choose to represent

v_θ as a deep neural network (DNN). The parameters θ are learned by using stochastic gradient descent to minimize the following loss function:

$$L(\theta) = (\min_a(c^a(s) + v_{\theta^-}(A(s,a))) - v_\theta(s))^2 \quad (2)$$

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 (Mnih et al. 2015). The parameters θ^- are periodically updated to θ during training. While we cannot guarantee convergence to v_*, approximate value iteration has been shown to approximate v_* (Bertsekas and Tsitsiklis 1996). For the puzzles investigated in this paper, the states used for training v_θ 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. This combination of deep neural networks and approximate value iteration is referred to as deep approximate value iteration (DAVI).

Batch Weighted A* Search

A* search (Hart, Nilsson, and Raphael 1968) is renowned as one of the most widely recognized and influential search algorithms. 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 states that have already been generated to their path costs. The cost of each node is $f(n) = g(n) + h(n.s)$, where $g(n)$ is the path cost, the sum of transition costs along the path from *start* to n, and $h(n.s)$ is the heuristic value, the estimated cost-to-go from the state associated with n to a nearest goal state. After a node is expanded, its children whose states are not already in CLOSED have their states added to CLOSED and then pushed to OPEN. If the state of a child node n is already in CLOSED, but the path cost of n is cheaper than the path cost recorded in CLOSED, then the path cost of the state associated with n is updated in CLOSED and n is added to OPEN. The algorithm starts with only n_{start}, a node corresponding to the *start* state, in OPEN and terminates when the node associated with a goal state is removed from OPEN.

When executing A* with a learned heuristic function (e.g., using 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 B 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 for potentially faster runtimes and less memory usage with a variant of A* search called weighted A* search (Pohl 1970). Weighted A* search computes the cost of each node as $f(n) = \lambda g(n) + h(n.s)$ where $\lambda \in [0, 1]$ is a scalar weighting. This combination of expanding B nodes every iteration and weighting the path cost by λ is referred to as batch-weighted A* search (BWAS). BWAS is a generalization of A* search since A* search can be recovered by setting λ to 1 and B to 1. A* search is guaranteed to find a shortest path if the heuristic function is admissible (Hart, Nilsson, and Raphael 1968) and weighted A* search is guaranteed to find a bounded suboptimal path if the heuristic function is admissible. An admissible heuristic function is a function that never overestimates the cost of a shortest path. Furthermore, it has been shown that BWAS is guaranteed to find a bounded suboptimal path given an admissible heuristic function (Agostinelli et al. 2021). While a DNN is not guaranteed to be admissible, obtaining admissible heuristic functions using DNNs is an ongoing area of research (Ernandes and Gori 2004; Agostinelli et al. 2021). Pseudocode for the BWAS algorithm is given in Algorithm 1.

Algorithm 1 Batch Weighted A* Search (BWAS)

```
Input: $start$, DNN $v_\theta$, batch size $B$, weight $\lambda$
OPEN ← priority queue of nodes based on minimal $f$
CLOSED ← maps states to their shortest discovered path costs
$UB, n_{UB} \leftarrow \infty$, NIL
$LB \leftarrow 0$
$n_{start} \leftarrow$ NODE($s = start, g = 0, p =$ NIL$, f = v_\theta(start)$)
PUSH $n_{start}$ to OPEN
while not IS_EMPTY(OPEN) do
  generated ← []
  while not IS_EMPTY(OPEN) and SIZE(generated) $< B$ do
    $n = (s, g, p, f) \leftarrow$ POP(OPEN)
    if IS_EMPTY (generated) then
      $LB \leftarrow \max(f, LB)$
    if IS_GOAL($s$) then
      if $UB > g$ then
        $UB, n_{UB} \leftarrow g, n$
      continue loop
    for $a$ in $|\mathcal{A}|$ do
      $s' \leftarrow A(s, a)$
      $g(s') \leftarrow g(s) + c^a(s)$
      if $s'$ not in CLOSED or $g(s') <$ CLOSED[$s'$] then
        CLOSED[$s'$] ← $g(s')$
        APPEND(generated, ($s', g(s'), n$))
  if $LB \geqslant \lambda \cdot UB$ then
    return PATH_TO_GOAL($n_{UB}$)
  generated_states ← GET_STATES(generated)
  heuristics ← $v_\theta$(generated_states)
  for $0 \leqslant i \leqslant$ SIZE(generated) do
    $s, g, p \leftarrow$ generated[$i$]
    $h \leftarrow$ heuristics[$i$]
    $n_s \leftarrow$ NODE($s, g, p, f = \lambda \cdot g + h$)
    PUSH $n_s$ to OPEN
return PATH_TO_GOAL($n_{UB}$)          // failure if $n_{UB}$ is NIL
```

Q-learning

Instead of learning a function, v_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 (Bertsekas et al. 1995). In a deterministic, finite-horizon environment, the Q-factor is defined as:

$$Q(s, a) = c^a(s) + \gamma V(s') \quad (3)$$

$V(s')$ can be expressed in terms of Q with $V(s') = \min_{a'} Q(s', a')$. Learning Q by iteratively updating the left-hand side of (3) toward its right-hand side is known as Q-learning (Watkins and Dayan 1992). 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) (Mnih et al. 2015). 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:

$$L(\phi) = \left((c^a(s) + \min_{a'} q_{\phi^-}(A(s,a),a')) - q_\phi(s,a) \right)^2 \tag{4}$$

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 (Watkins and Dayan 1992). 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 is 127 times faster than DAVI.

Methods

Q* Search

We present Q* search, a search algorithm that builds on A* search to take advantage of DQNs. Q* search uses tuples containing a node and an action, which we will refer to as node_action tuples, to search for a path to the goal. The path cost of a node_action tuple, (s,a), is $g(s) + c^a(s)$ and the heuristic value is $h(A(s,a))$. Therefore, the priority of a node_action tuple, $f(s,a)$, is $f(s,a) = g(s) + c^a(s) + h(A(s,a))$. The Q-factor $Q(s,a)$ is equal to $c^a(s) + h(A(s,a))$. Therefore, using DQNs, the transition cost and cost-to-go of all child nodes can be computed using q_ϕ without having to expand s. In various domains, transition costs ($c^a(s)$) for states are often predetermined and readily accessible, eliminating the need for estimation. However, in certain contexts, computing these costs can be resource-intensive. Take motion planning, for instance; calculating the transition cost between states (configurations) often requires executing a local planner, which can be computationally demanding. Moreover, transition costs may vary depending on the subsequent state, adding another layer of complexity. For instance, in grid-like environments, the terrain of both the current and next states can influence the transition cost. Consequently, in the general case, q_ϕ approximates c^a, but this approximation can be substituted with the true transition cost if available.

At every iteration, Q* search pops a node_action tuple, (s,a), from OPEN and generates a new state, $s' = A(s,a)$. Instead of expanding s', Q* search applies the DQN to s' to obtain the sum of the transition cost and the cost-to-go for all of its children. Therefore, we only need a single forward pass through a DNN instead of $|\mathcal{A}|$. Q* search then pushes the new node_action tuples (s', a') to OPEN for all actions $a' \in \mathcal{A}$, where the cost is computed by summing the path cost of s and the output of the DQN corresponding to action a'. In Q* search, the only part that depends on the size of the action space is pushing nodes to OPEN. Unlike A* search, only one node is generated per iteration, regardless of the size of the action space, and the heuristic function only needs to be applied once per iteration.

Algorithm 2 Batch Weighted Q* Search (BWQS)

```
Input: start, DNN $q_\phi$, batch size $B$, weight $\lambda$
OPEN ← priority queue of nodes based on minimal $f$
CLOSED ← maps states to their shortest discovered path costs
$U, n_U \leftarrow \infty$, NIL
$LB \leftarrow 0$
$n_{start} \leftarrow$ NODE($s = start, g = 0, p =$ NIL$, a =$ NO_OP$, f = 0$)
PUSH $n_{start}$ to OPEN
while not IS_EMPTY(OPEN) do
    generated ← []
    while not IS_EMPTY(OPEN) and SIZE(generated) $< B$ do
        $n = (s, a, g, p, f) \leftarrow$ POP(OPEN)
        if IS_EMPTY (generated) then
            $LB \leftarrow \max(f, LB)$
        $s' \leftarrow A(s,a)$
        $g(s') \leftarrow g(s) + c^a(s)$
        if IS_GOAL($s'$) then
            if $U > g + c^a(s)$ then
                $U, n_U \leftarrow g + c^a(s), n$
            continue loop
        if $s'$ not in CLOSED or $g(s') <$ CLOSED[$s'$] then
            CLOSED[$s'$] $\leftarrow g(s')$
            for $a'$ in $|\mathcal{A}|$ do
                APPEND(generated, $(s', g(s'), a', n)$)
    if $LB \geqslant \lambda \cdot U$ then
        return PATH_TO_GOAL($n_U$)
    generated_states_actions ← GET_STATES(generated)
    transition_costs, heuristics ← $q_\phi$(generated_states_actions)
    for $0 \leqslant i \leqslant$ SIZE(generated) do
        $s, a, g, p \leftarrow$ generated[$i$]
        $g' \leftarrow g +$ transition_costs[$i$]
        $h \leftarrow$ heuristics[$i$]
        $n_{(s,a)} \leftarrow$ NODE($s, a, g, p, f = \lambda \cdot g' + h$)
        PUSH $n_{(s,a)}$ to OPEN
return PATH_TO_GOAL($n_U$)                    // failure if $n_U$ is NIL
```

We also perform Q* in batches of size N and with a weight λ, resulting in a variant denoted as BWQS. It is important to note that BWQS serves as a generalization of Q*, where $\lambda = 1$ and $B = 1$. Consequently, the pseudocode for the BWQS algorithm, as outlined in Algorithm 2, inherently encompasses Q*.

Theoretical Analysis

We will show that the BWQS algorithm qualifies as a bounded-suboptimal search approach. This indicates that it is ensured to discover a path with a cost $U \leqslant \frac{C^*}{\lambda}$. This holds true under the condition that all Q-factors $Q(s,a)$ never overestimate $c^a(s) + d(A(s,a), goal)$;[1] we refer to a heuris-

[1] Note that the network can overestimate each component individually, as long as the sum of both components does not result in

tic function meeting this criteria as q-admissible. This proof is an adaptation for the proof that A* search is an admissible search algorithm (Hart, Nilsson, and Raphael 1968).

Lemma 1. *As long as BWQS did not terminate, either there exists a node in OPEN corresponding to a prefix of some shortest path from* $start$ *to* $goal$*, or a shortest path from* $start$ *to* $goal$ *was discovered.*

Proof. At the beginning of the search, $n_{(start,\text{NO\_OP})}$ is in OPEN with $g(n_{(start,\text{NO\_OP})}) = 0$, which is the prefix of any shortest path from $start$ to $goal$. In every search iteration $i > 1$ in a shortest path from $start$ to $goal$ was not discovered, let P be some shortest path from $start$ to $goal$ and Δ be the set of closed nodes in P, that were expanded with optimal g-value. That is, $\Delta = \{n | n \in P \text{ and } , n \in \text{ CLOSED } \text{ and } g(n) = d(start, n)\}$. Δ is not empty, as after the first search iteration, $start \in \Delta$. Let n^* be the element in Δ with the highest index. Since an optimal path from $start$ to $goal$ has not been discovered, $n^* \neq goal$. Let n' be the successor of n' in P. Due the the optimality of P, $g(n') = d(start, n')$. In addition, since $n' \notin \Delta$ and $n^* \in \Delta$, n' is in OPEN. □

Using Lemma 1, we prove our main theorem.

Theorem 1. *Given that all transition costs are greater than zero,* $0 \leqslant \lambda \leqslant 1$*, and a q-admissible heuristic function, BWQS is bounded suboptimal. That is, BWQS returns a solution with a cost bounded by* $\frac{1}{\lambda} \cdot C^*$*, if such a solution exists.*

Proof. First, it is important to recognize that BWQS consistently maintains information about the shortest path discovered so far to the $goal$, identified with a cost denoted as U. Upon termination, BWQS returns this solution. Termination of BWQS occurs under two conditions: either a solution is found with $\lambda \cdot U \leqslant LB$, or the OPEN set becomes empty after exhaustively expanding all nodes in the graph. Consequently, if there exist paths from $start$ to $goal$, BWQS is guaranteed to discover one.

Now, we aim to show that the path returned by BWQS is bounded by $\frac{1}{\lambda} \cdot C^*$. Assume by contradiction that BWQS has terminated and produced a solution with a cost $U > \frac{1}{\lambda} \cdot C^*$. As the algorithm has concluded, we have $LB \geqslant \lambda \cdot U$, indicating that at least one node was expanded with a priority greater than or equal to $\lambda \cdot U$. Let $n_{(s,a)}$ denote the first node expanded during the search with a priority greater than or equal to $\lambda \cdot U$. According to Lemma 1, at the moment $n_{(s,a)}$ was chosen for expansion, there existed another node $n_{(s',a')}$ in OPEN, corresponding to an optimal path (costing C^*).

Since $n_{(s,a)}$ was expanded instead of $n_{(s',a')}$, we infer that at the moment of expansion, $\lambda \cdot g(n_{(s,a)}) + q^c_\theta(s, a) \leqslant \lambda \cdot g(n_{(s',a')}) + q^c_\theta(s', a')$. By virtue of q-admissibility, $g(n_{(s',a')}) + q^c_\theta(s', a') \leqslant C^*$, thus $\lambda \cdot g(n_{(s,a)}) + q^c_\theta(s, a) \leqslant C^*$. However, given that the priority of $n_{(s,a)}$ was greater than or equal to $\lambda \cdot U$, and $U > \frac{1}{\lambda} \cdot C^*$, we reach a contradiction, as $\lambda \cdot g(n_{(s,a)}) + q^c_\theta(s, a) > C^*$.

□

Experimental Evaluation

In this section, we detail our empirical evaluation of Q*.

Settings, Baselines, and Network Architectures

We evaluate Q* across various domains, including the Rubik's Cube (which has 12 actions), the 7 by 7 Lights Out puzzle (Agostinelli et al. 2019) (which has 49 actions), and the 35-Pancake puzzle (which has 49 actions).

We compare Q* to both A* search as well as the deferred version of A* search (Helmert 2006), which we refer to as A$_d$*, where the heuristic value of each child is set to be the same as the heuristic value of the parent. All algorithms expand a batch of nodes N, instead of of a single node at each iteration, and a weight λ (i.e., we evaluated the batch-weighted version for each algorithm). Both A* and A$_d$* employ a state-based cost-to-go function model trained using DVAI. In contrast, Q* utilizes a state-action-based cost-to-go estimation, trained using Q-learning.

We train the state-based cost-to-go function with the same architecture described in Agostinelli et al. (2019), 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 (He et al. 2016), followed by a layer of size 1 representing the cost-to-go. The state-action-based cost-to-go function (DQN, $q_\phi(s, a)$) also has the same architecture with the exception that the output layer is a vector that estimates the cost-to-go for taking every possible action. This implementation of $q_\phi(s, a)$ estimates $c^a(s) + h(A(s, a))$ as a single entity. However, to accommodate the weighted version of Q*, it is essential to multiply the transition cost $c^a(s)$ by λ, as depicted in Algorithm 2. Consequently, our current setup does not facilitate bounded-suboptimal search, for scenarios involving non-uniform transition costs. Nevertheless, in our evaluation, all transition costs are uniform. Thus, this constant offset does not impact the order in which nodes are generated. However, for future endeavors, this limitation could be addressed by training a DQN that segregates the computation of transition costs and cost-to-go (two-head network).

For generating training states, the number of times we scramble the puzzle, K, is set to 30 for the Rubik's cube, 50 for Lights Out, and 70 for the 35-Pancake puzzle.

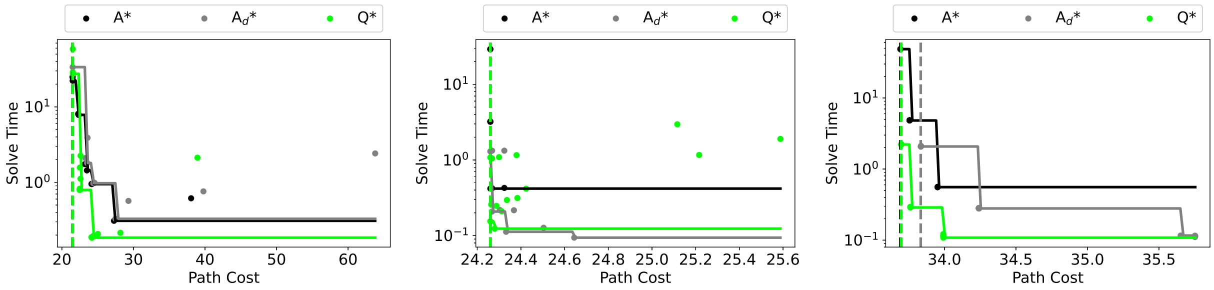
For Q-learning, we select actions according to a Boltzmann distribution where each action a is selected with probability:

$$p_{s,a} = \frac{e^{(-\,q_\phi(s,a)/T)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(-\,q_\phi(s,a')/T)}} \tag{5}$$

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

We train each model with a batch size of 10,000 for 1.2 million iterations using the ADAM optimizer (Kingma and Ba 2014). We update the target networks with the same schedule defined in the DeepCubeA source code (Agostinelli et al. 2020). The machines we use for training and search have 48 2.4 GHz Intel Xeon central processing units (CPUs), 192 GB of random access memory, and two 32GB NVIDIA V100 GPUs.

an overall overestimation.

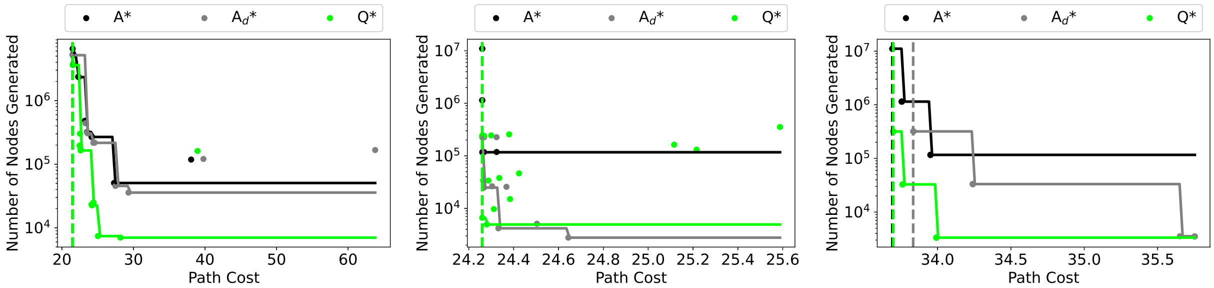


(a) RC

(b) Lights Out

(c) 35-pancake

Figure 1: Relationship between the average path cost and the average time to find a solution.



(a) RC(12)

(b) Lights Out

(c) 35-Pancake

Figure 2: Relationship between the average path cost and the average node generations.

Prior work on solving the Rubik's cube with deep reinforcement learning and A* search used $\lambda = 0.6$ and $N = 10000$ for BWAS (Agostinelli et al. 2019). 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 Q* 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 for the Rubik's cube and 500 test states for Lights Out as used in previous work by Agostinelli et al. (2019). We generated 500 test states for the 35-Pancake puzzle. Each test state was obtained by scrambling the puzzle between 1,000 and 10,000.

Results

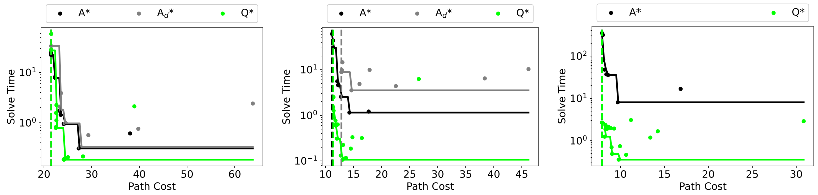
The results are reported in Figures 1 and 2. These figures illustrate the relationship between the average path cost and either the average time taken to find a solution or the average number of generated nodes, respectively. Both figures employ a logarithmic scale on the y-axis, with each data point representing a specific search parameter setting. The dashed line signifies the lowest average path cost identified, while the solid line represents either the fastest solution time or the fewest number of node generations, depending on the context, as determined by a hypothetical threshold for an acceptable average path cost.

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. DAVI is significantly slower than Q-learning, especially when the size of the action space is large.

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

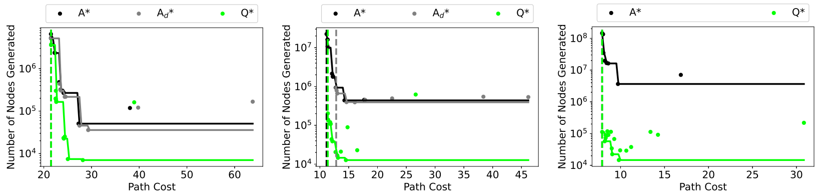
The figures show that, for almost any possible path cost threshold, Q* is significantly faster and generates significantly fewer nodes than both A* and A$_d$*. A$_d$* exhibits overall improvement over A* as it generates only one node per iteration. However, as it assigns the same f-value to all children of a node, it introduces inefficiencies by being unable to prioritize one child node over another.

The lowest average path cost achieved by all algorithms remains comparable, with the maximum difference in aver-



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

Figure 3: Action space size ablation study on Rubik's cube: average path cost vs average time to find a solution.



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

Figure 4: Action space size ablation study on Rubik's cube: average path cost vs average node generations.

age path cost being a mere 0.1%, observed in the RC domain. Overall, For RC, in the best case, A* finds a shortest path 59% of the time while Q* finds a shortest path 56.4% of the time. For Lights Out, in the best case, both A* and Q* find a shortest path 100% of the time.

Ablation Study: Varying Number of Actions

In order to study the performance of Q* as the number of actions increases, we perform an ablation study focusing on the Rubik's cube domain. The standard RC action space includes 12 different actions: each of the six faces can be turned clockwise or counterclockwise. We denote this action space as RC(12). For the ablation study, we add meta-actions to the RC 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 RC(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.

In the experiments reported earlier, we trained the model for 1.2 million iterations. However, as the size of the action space increases, training becomes infeasible for DAVI. Table 1 shows that DAVI 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 DAVI with a batch size of 769, and since RC(1884) has 157 times more actions, we train DAVI with a batch size of 63. Moreover given the escalation in solving time with the expansion of the action space in A* search, we utilize a subset of 100 states for RC(156) and 20 states for RC(1884), instead of the full 500 states employed for RC(12).

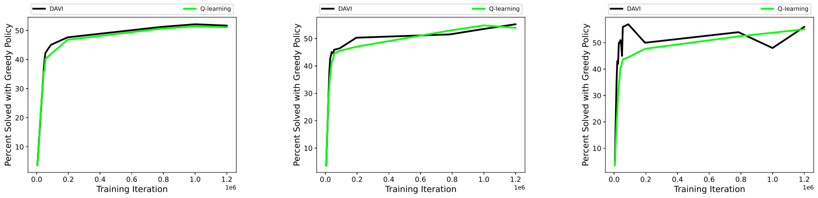
Figures 3 and 4 repeat the experiments of Figures 1 and 2 on the RC environment with the different action-space sizes. The results show that the performance of Q* over A* and A*$_d$ becomes even more pronounced as the action space increases. In fact, for RC(1884), A$_d$* was unable to find a solution due to running out of memory. In the most extreme case, the cheapest average path cost for A* and Q* is identical for RC(1884), however, Q* is 129 times faster and generates 1228 times fewer nodes than A*. The ratios for various desired average path costs are presented in Table 2. It is evident from the table that Q* consistently outperforms A* in all instances except one, often exhibiting orders of magnitude faster speed and greater memory efficiency than A*.

When comparing A* and Q* to themselves for different action spaces, Table 3 shows that, though RC(1884) has 157 times more actions than RC(12), Q* 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, A* takes 37 times as long and generates 62.7 times as many nodes. Overall, Q* has much better performance for both metrics. For RC(156) Q* finds solutions in even less time than it did for RC(12) due to the addition of meta-actions.

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 5 shows this metric as a function of training time. The results show that, in RC(12), DAVI is slightly better than Q-learning. In RC(156) and RC(1884), even though the batch size for DAVI is smaller, the performance is on par with Q-learning. This may be due to the fact that DAVI is only learning the cost-to-go for a single state while Q-learning must learn the sum of the transition cost and cost-to-go for all possible next states.

Table 2: The ratio between A* and Q* search for the solution time and number of nodes generated for hypothetical acceptable path cost thresholds for RC(12), RC(156), and RC(1884).

	Path Cost Threshold								
	RC (12)			**RC (156)**			**RC (1884)**		
	22	25	28	12	14	16	8	9	10
Time	0.8	5.1	1.7	50.8	23.9	10.8	129.7	28.5	22.6
Nodes	1.4	11.9	6.8	92.0	64.0	34.5	1288.4	282.8	249.1



(a) RC(12)

(b) RC(156)

(c) RC(1884)

Figure 5: Percentage of training states solved with a greedy policy as a function of training iteration.

Table 3: Ratio of action space sizes, along with performance ratios (time and nodes generated) compared to RC(12), averaged over all search parameter settings, with standard deviation in parenthesis.

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

Discussion

As the size of the action space increases, Q* becomes significantly more effective than A* in terms of solution time and the number of nodes generated. In the largest action space Q* is orders of magnitude faster and generates orders of magnitude fewer nodes than A* while finding solutions with the same average path cost. For smaller action spaces, while Q* is almost always faster and more memory efficient, A* is capable of finding solutions that are slightly cheaper than Q*. This could be due to the difference in what v_θ and q_ϕ are computing. Since the forward pass performed by the DQN, q_ϕ, is the same as doing a one-step lookahead with v_θ, this could make learning q_ϕ more difficult than learning v_θ. This may explain why, in the case of Lights Out, A$_d$* is, in some cases, faster and generates fewer nodes than Q*. However, Q* becomes better as the path cost threshold decreases. Since training and search are significantly faster for Q-learning and Q*, 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 fixed action spaces, Q* search can readily be applied to a dynamic action space given a DQN capable of computing Q-factors for such an action space. Therefore, it is possible to use Q* to solve problems with dynamic action spaces by choosing a DQN architecture that uses structured prediction. Architectures such as graph convolutional policy networks (You et al. 2018), which were used for molecular optimization, could be modified to estimate Q-factors on problems with a graph structure that corresponds to the action space. In problems involving sequences, Long Short-Term Memory (Hochreiter and Schmidhuber 1997) or Transformer (Vaswani et al. 2017) architectures could be used to compute Q-factors. This would have a direct application to problems with large, but variable, action spaces such as chemical synthesis, theorem proving, program synthesis, and web navigation.

Conclusion

Efficiently solving search problems with large action spaces has been of importance to the artificial intelligence community for decades (Russell 1992; Korf 1993; Yoshizumi, Miura, and Ishida 2000). Q* search uses a DQN to eliminate the majority of the computational and memory burden associated with large action spaces by generating only one node per iteration and requiring only one application of the heuristic function per iteration. When compared to A* search, Q* search is up to 129 times faster and generates up to 1288 times fewer nodes. When increasing the size of the action space by 157 times, Q* search only takes 3.7 times as long and generates only 2.3 times more nodes. The ability that Q* 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

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2020. DeepCubeA. https://github.com/forestagostinelli/DeepCubeA.

Agostinelli, F.; McAleer, S.; Shmakov, A.; Fox, R.; Valtorta, M.; Srivastava, B.; and Baldi, P. 2021. Obtaining Approximately Admissible Heuristic Functions through Deep Reinforcement Learning and A* Search. In *International Conference on Automated Planning and Scheduling - Bridging the Gap Between AI Planning and Reinforcement Learning Workshop*.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17): 2075–2098.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P.; Bertsekas, D. P.; Bertsekas, D. P.; and Bertsekas, D. P. 1995. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific. ISBN 1-886529-10-8.

Chen, B.; Li, C.; Dai, H.; and Song, L. 2020. Retro*: learning retrosynthetic planning with neural guided A* search. In *International Conference on Machine Learning*, 1608–1616. PMLR.

Chen, H.-C.; and Wei, J.-D. 2011. Using neural networks for evaluation in heuristic search algorithm. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

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

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N.; Schaeffer, J.; and Holte, R. 2012. Partial-expansion A* with selective node generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Hornik, K.; Stinchcombe, M.; and White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5): 359–366.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence*, 62(1): 41–78.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's Cube with Approximate Policy Iteration. In *International Conference on Learning Representations*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

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

Puterman, M. L.; and Shin, M. C. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11): 1127–1137.

Russell, S. J. 1992. Efficient Memory-Bounded Search Methods. In *ECAI*, volume 92, 1–5.

Schmidhuber, J. 2015. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8(3-4): 279–292.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with Partial Expansion for Large Branching Factor Problems. In *AAAI/IAAI*, 923–929.

You, J.; Liu, B.; Ying, R.; Pande, V.; and Leskovec, J. 2018. Graph convolutional policy network for goal-directed molecular graph generation. *arXiv preprint arXiv:1806.02473*.

Zhang, Y.-H.; Zheng, P.-L.; Zhang, Y.; and Deng, D.-L. 2020. Topological Quantum Compiling with Reinforcement Learning. *Physical Review Letters*, 125(17): 170501.