# GPU-Accelerated A* Search with Deep Neural Network Heuristics

**Misagh Soltani, Forest Agostinelli**

Department of Computer Science and Engineering
University of South Carolina, USA
msoltani@email.sc.edu, foresta@cse.sc.edu

## Abstract

Heuristic search is a foundational technique in artificial intelligence. Recent work has demonstrated that powerful heuristic functions represented as deep neural networks (DNNs) can be learned with machine learning. This combination of machine learning and heuristic search has been successfully applied to puzzle solving, quantum circuit synthesis, and chemical reaction mechanism pathfinding. This success has been accompanied by batched heuristic search algorithms, such as batch A* search, which take advantage of parallelism provided by graphics processing units (GPUs) for DNNs. However, the rest of the heuristic search algorithm remains on the central processing unit (CPU). This results in the often costly transfer of data from the CPU to the GPU for heuristic computation and leaves other GPU-based acceleration options unexplored. To address this, we investigate the benefit of performing heuristic search entirely on the GPU. Using the Rubik's Cube as our case study, we systematically profile the speedup achieved by each component of batch A* search when moved to the GPU. Our results show that GPU execution yields about $12.03\times$ speedup over the CPU baseline that only uses the GPU for the heuristic function. These findings suggest that fully GPU-native heuristic search with DNNs is a practical and effective strategy for scaling learned heuristics to demanding combinatorial search tasks.

**Code** — https://github.com/misaghsoltani/GPUAStar

## Introduction

Heuristic search remains one of the most effective approaches for pathfinding on weighted directed graphs. While previous approaches have relied on hand-designed heuristics, pattern databases (Korf 1997; Culberson and Schaeffer 1998), or standardized representations (Helmert 2006) to construct heuristics, deep reinforcement learning (Sutton and Barto 2018) has been shown to learn powerful heuristic functions represented as deep neural networks (DNNs) (Schmidhuber 2015; LeCun, Bengio, and Hinton 2015) with no domain-specific heuristic knowledge. In particular, DeepCubeA showed that a DNN coupled with search can solve Rubik's Cube instances, among other puzzles, without domain-specific heuristic information (Agostinelli et al. 2019). However, when a heuristic function is represented as a DNN, the computational bottleneck often shifts from node expansion to heuristic computation.

Recent work addresses this bottleneck in two complementary ways. One line of work increases the amount of parallel search work that is exposed to the heuristic evaluator by expanding multiple frontier nodes per iteration, so that many successors can be evaluated in a single batched forward pass (Felner, Kraus, and Korf 2003; Agostinelli et al. 2019; Li et al. 2022; Greco et al. 2023). A second line of work accelerates the search procedure itself through parallel frontier management, bulk successor generation, and GPU execution (Burns et al. 2010; Kishimoto, Fukunaga, and Botea 2013; Zhou and Zeng 2015; Futuhi and Sturtevant 2025). Together, these directions suggest a common systems principle: when the heuristic is most efficient on large tensor batches, the surrounding search algorithm must preserve enough ordered parallel work to keep the accelerator occupied. Nevertheless, many other aspects of the aforementioned heuristic search methods are often computed on central processing unit (CPU). This includes adding and removing nodes from the priority queue, goal tests, expanding nodes, and checking and modifying the closed set. Also, since the DNN computation happens on the GPU, data must first be transferred from the CPU to the GPU, which, as we show in this paper, can consume a significant amount of time.

To eliminate CPU-to-GPU transfers during search and to maximize the parallelism provided by GPUs for other aspects of search, we present a GPU-based batch weighted A* search (Pohl 1970) algorithm called $\text{BWAS}_{\text{GPU}}$. $\text{BWAS}_{\text{GPU}}$ uses the GPU to push to and pop from a priority queue, perform goal tests, perform node expansion, and maintain and update a closed dictionary. We compare $\text{BWAS}_{\text{GPU}}$ to BWAS for the $3 \times 3 \times 3$ Rubik's Cube and show that it is approximately $12.03\times$ faster.

## Related Work

### Learned Heuristics and Deep Reinforcement Learning

In recent years, researchers have explored learning heuristics with neural networks to generalize across problem instances without domain-specific heuristic knowledge. Early work demonstrated that neural networks can approximate search estimates through bootstrapping from simpler heuris-

tics (Arfaee, Zilles, and Holte 2011). Approaches that leverage deep reinforcement learning, such as DeepCubeA (Agostinelli et al. 2019), do not assume that a pre-existing solver is given. DeepCubeA trains a DNN to represent a heuristic function using deep reinforcement learning and subsequently employs the learned DNN as an inadmissible heuristic function. This can result in performance that surpasses that of domain-independent heuristics derived from standardized representations (Agostinelli, Panta, and Khandelwal 2024; Muppasani et al. 2024). More recently, DeepCubeAI (Agostinelli and Soltani 2024) and SPAR (Soltani and Agostinelli 2025) learn a world model, use it for learning a heuristic function, and then use the learned world model as the expansion function along with the learned heuristic function to perform heuristic search. However, a limitation of DNN heuristics is their high inference cost.

### Batched and Multi-Node Expansion Strategies

Several works address the slow-heuristic problem by batching or parallelizing node expansions to take advantage of parallelism provided by GPUs for DNNs.

**Weighted and bounded-suboptimal search.** Weighted A* (Pohl 1970), parameterized by an inflation factor $w \geq 1$, has long been used to trade solution optimality for reduced search time (Pearl and Kim 1982). *K*-Focal Search (Greco et al. 2023) extends this idea to a bounded-suboptimal setting: rather than expanding only the single best node, the algorithm maintains a focal list and expands the $K$ most promising nodes per iteration, computing all their heuristic values in parallel on the GPU. In experiments on Rubik's Cube and the 24-puzzle, *K*-Focal Search achieves roughly $6\times$ faster wall-clock runtime and vastly higher node expansion throughput compared with standard focal search (Greco et al. 2023).

**$K$-Best-First Search and BWAS.** Prior work (Felner, Kraus, and Korf 2003) generalized best-first search through $K$-Best-First Search (KBFS), which expands the top $K$ nodes from the open list per iteration, thereby enabling batch heuristic evaluation. Building on this foundation, (Agostinelli et al. 2019) introduced Batch Weighted A* (BWAS), which removes $K$ nodes from open each cycle and issues all neural network heuristic evaluations for their successors in a single batched inference step. By amortizing the fixed overhead of GPU kernel launches and memory transfers over a batch of $K$ successors, BWAS substantially reduces the per-node effective inference cost.

These multi-node expansion strategies are unified by a common insight: evaluating $K$ successor nodes in one batched forward pass exploits the parallelism of modern GPU hardware far more efficiently than $K$ sequential single-node evaluations.

### Parallel and GPU-Accelerated Search

**GA* and GPU-based A*.** (Zhou and Zeng 2015) presented GA*, a GPU-parallelized implementation of A*, which offloads state expansion, successor generation, and open-list management to the GPU to achieve up to approximately $45\times$ speedup over CPU-based A* on large search problems. This established GPU memory management of the search frontier as a viable and highly effective paradigm. GA* gains much of its parallelism by changing frontier semantics: work is distributed across many queues and expansion is governed by local queue minima and a specialized stopping rule. Our approach preserves the ordered batch semantics of BWAS: each iteration removes the global top $K$ nodes under a stable order, generates successors in a fixed parent-major/action-major order, and applies an exact strict-improvement CLOSED update. Our approach is therefore intended to move the same ordered batched computation onto the GPU, rather than to introduce a relaxed GPU search policy.

**Hybrid CPU-GPU frameworks.** More recent frameworks extend GPU parallelism to learned heuristics. Recent work (Futuhi and Sturtevant 2025) observes that batch variants of A* and Weighted A* can be realized by delaying and grouping heuristic calls so that a single GPU kernel evaluates many states simultaneously. In this hybrid CPU-GPU framework for depth-first search variants, the authors demonstrate that GPU acceleration provides large throughput gains when the batch size is sufficiently large to saturate GPU compute resources. Analogously, domain-specific planners such as *K*-Focal (Greco et al. 2023) exploit GPU batching to offset the high per-call cost of neural network heuristics.

## Methods

### Ordered Device-Resident Closed Set

The accelerator-resident closed set is an ordered partial map $D : \mathcal{K} \rightharpoonup \mathcal{V}$. The key domain $\mathcal{K}$ is either the set $I_{64}$ of signed 64-bit integers or the Cartesian product $I_{64}^{\ell}$ of fixed-width integer vectors. The value domain $\mathcal{V}$ is a fixed typed tensor space of shape $s_1 \times \cdots \times s_r$, where $r = 0$ denotes scalar values. Let $M = 2^p$ be the address-table capacity for some integer $p \geq 3$, and let $\mu = M - 1$. The address table $T \in \{-2, -1, 0, \ldots, L-1\}^M$ stores $-1$ for empty slots, $-2$ for tombstones, and nonnegative integers as pointers into an append-only entry log of current length $L$. The entry log contains hash words $H \in W_{64}^L$, a liveness mask $\Lambda \in \{0,1\}^L$, stored keys $K_{\log}$, and stored values $V_{\log}$, where $W_{64} = \{0, \ldots, 2^{64} - 1\}$ is the set of 64-bit words. A previously unseen key appends exactly one new live log entry, while an update to an existing key changes only its stored value. Live entries can therefore be materialized in insertion order by scanning the live log positions from left to right (Python Software Foundation 2019).

All bit-level arithmetic in the hash function is defined on $W_{64}$. We write $\oplus$ for bitwise exclusive-or, $\wedge$ for bitwise and, and $x \gg_u b = \lfloor x/2^b \rfloor$ for logical right shift by $b$ bits. Addition and multiplication are taken modulo $2^{64}$. A table-specific salt $\sigma \in W_{64}$ is sampled once at initialization. For a scalar key $k \in I_{64}$, the hash value is

$$h(k) = m(k \oplus \sigma). \tag{1}$$

Here $m : W_{64} \to W_{64}$ is the SplitMix64 mixer (Steele Jr.,

Lea, and Flood 2014)

$$\begin{aligned} z_1 &= x + \gamma, \\ z_2 &= \big(z_1 \oplus (z_1 \gg_u 30)\big) c_1, \\ z_3 &= \big(z_2 \oplus (z_2 \gg_u 27)\big) c_2, \\ m(x) &= z_3 \oplus (z_3 \gg_u 31), \end{aligned} \tag{2}$$

with constants $\gamma = 0\text{x}9\text{E}3779\text{B}97\text{F}4\text{A}7\text{C}15$, $c_1 = 0\text{xBF}58476\text{D}1\text{CE}4\text{E}5\text{B}9$, and $c_2 = 0\text{x}94\text{D}049\text{BB}133111\text{EB}$. For a vector key $y = (y_1, \ldots, y_\ell) \in I_{64}^{\ell}$, the table precomputes lane offsets $a_j = j\gamma$ and lane weights $b_j = m(a_j \oplus c_1)$, forms

$$u_j = m\big((y_j \oplus \sigma) + a_j\big), \tag{3}$$

and reduces the lanes through

$$h(y) = m\left(\left(\sum_{j=1}^{\ell} u_j b_j\right) \oplus \sigma\right). \tag{4}$$

This reduction is order sensitive, which is required because the keys are fixed coordinate vectors rather than unordered multisets. For wide keys, the sum is evaluated in chunks to bound temporary memory without changing the algebra.

Collision resolution uses perturbation-based open addressing (Python Software Foundation 2019). For a hash word $r \in W_{64}$, the initial probe state is $j_0 = r \wedge \mu$ and $p_0 = r$. At probe step $t$, the table inspects slot $j_t$. Empty slots terminate unsuccessful search. Occupied slots compare the stored hash and then the stored key. Tombstones do not terminate the probe and are remembered as provisional insertion positions. The recurrence is

$$p_{t+1} = p_t \gg_u 5, \qquad j_{t+1} = (5j_t + 1 + p_{t+1}) \wedge \mu. \tag{5}$$

When the table contains no tombstones and the query batch is large enough to amortize the staging overhead, the first $\min\{6, M\}$ probe locations are evaluated in a staged vectorized pass before any unresolved queries continue with the same recurrence. Smaller batches and execution use the same recurrence directly.

A batched insertion processes an ordered sequence $\{(k_i, v_i)\}_{i=1}^{B}$ and matches the semantics of sequential assignment in that same order. If a key appears multiple times in one batch, its final stored value is the value from its last occurrence, while the insertion position of a previously absent key is determined by its first occurrence. Let $u_1, \ldots, u_R$ be the distinct keys ordered by first occurrence, let $\psi(i) \in \{1, \ldots, R\}$ map row $i$ to its distinct key, and let $\ell_j = \max\{i : \psi(i) = j\}$. The batch reduces to $\{(u_j, v_{\ell_j})\}_{j=1}^{R}$. Updates to already present keys overwrite only the stored value. Previously absent keys append new log entries. If several new keys initially target the same address slot, placement is resolved iteratively by admitting the earliest proposal for each slot, reprobing the rejected proposals against the updated table, and repeating until every new key is placed. This stable conflict rule preserves insertion order exactly.

Let $n$ denote the number of live entries, let $f$ denote the number of nonempty address slots, and let $d$ denote the number of tombstones. Before processing $R$ distinct candidate keys, the table first rebuilds at the current capacity whenever $d > \max\{32, n\}$. It then doubles the capacity until

$$3(f + R) < 2M \tag{6}$$

holds. A rebuild compacts the live log, resets the address table to the empty state, and reinserts the live entries in increasing insertion order. This removes tombstones, restores the guarantee that at least one empty slot exists, and preserves ordered materialization.

Lookup, membership, and ordered materialization are fully batched. The closed-set primitive used during search is a fused strict-minimum update for floating-point scalar values. For an ordered batch $\{(k_i, c_i)\}_{i=1}^{B}$ with $c_i \in \mathbb{R}$, row $i$ is accepted if and only if $c_i$ is smaller than both the previously stored value for $k_i$ and every earlier batch value associated with the same key. If $u_1, \ldots, u_R$ and $\psi(i)$ denote the distinct keys and their inverse map, and if $\rho_j \in \mathbb{R} \cup \{+\infty\}$ is the value currently stored for $u_j$, then the acceptance rule is

$$c_i < \min\big(\rho_{\psi(i)}, \min\{c_j : j < i,\ k_j = k_i\}\big). \tag{7}$$

After the update, each distinct key stores

$$\rho'_j = \min\big(\rho_j, \min\{c_i : \psi(i) = j\}\big). \tag{8}$$

The accelerator-resident variant computes this rule through a stable grouping of equal keys followed by a shifted prefix-minimum reduction, so the acceptance mask and the updated stored values are produced in one device-resident pass.

## Batch Weighted Search

A search node $n$ stores a state $s(n) \in \mathcal{S}$, a path cost $g(n)$, a heuristic value $h(n) = h_\theta(s(n))$, a goal indicator $z(n) = G(s(n))$, a parent reference, and the action that generated the node. We reserve the term classical Weighted A* for the priority

$$f_{\mathrm{WA}}(n) = g(n) + \omega h(n), \qquad \omega \geq 1, \tag{9}$$

which is standard in the heuristic-search literature (Pohl 1970; Pearl and Kim 1982).

We instead use the weighted score

$$f_w(n) = w\, g(n) + (1 - z(n))\, h(n), \tag{10}$$

where $0 \leq w \leq 1$ is a path-cost weight. The factor $(1 - z(n))$ ensures that the heuristic contribution of a goal node is zero. Let $K$ be the maximum number of nodes removed from OPEN in one search iteration, and let $B_h$ be the maximum number of states evaluated in a single heuristic forward pass. When $B_h$ is unspecified, the entire surviving child batch is scored at once.

Both variants use the same score and goal semantics. Goal testing is applied to every generated child before heuristic evaluation so that goal children receive zero heuristic contribution in the ranking function. Termination is based on goals removed from OPEN rather than on merely generated successors: the search stops only after completing an iteration whose popped batch contains at least one goal node. Algorithm 1 summarizes this procedure for the accelerator-resident variant.

Algorithm 1: Accelerator-Resident BWAS

**Input:** root state $s_0$; ordered actions $\mathcal{A}$; encoded transition $E$; transition costs $c$; goal test $G$; heuristic $h_\theta$; path-cost weight $w$; batch limits $K$ and $B_h$

**Output:** state trajectory and action sequence to a popped goal, or failure if OPEN is exhausted

Initialize $\mathcal{N}$, OPEN, CLOSED with missing values $+\infty$, and $\Gamma \leftarrow \emptyset$
Encode $s_0$ as $x_0$; create $n_0 = (x_0, 0, h_\theta(x_0), G(x_0), -1, -1)$; append $n_0$ to $\mathcal{N}$, insert it into OPEN with key $f_w(n_0)$, and set $\mathrm{CLOSED}[x_0] \leftarrow 0$
**while** OPEN $\neq \emptyset$ and $\Gamma = \emptyset$ **do**
    Remove up to $K$ node indices $P$ from OPEN in stable nondecreasing $f_w$ order, and append $\{i \in P : z_i = 1\}$ to $\Gamma$
    Gather states and path costs for $P$, then generate $Y \leftarrow \langle (E(x_i, a),\ g_i + c(x_i, a),\ i,\ a) : i \in P,\ a \in \mathcal{A} \rangle$ in parent-major then action-major order
    Evaluate $z_j \leftarrow G(x_j)$ for every candidate $j \in Y$
    Compute $\chi_j$ and update CLOSED by the fused strict-minimum update on ordered pairs $(x_j, g_j)$; retain $R \leftarrow \langle j \in Y : \chi_j = 1 \rangle$
    Evaluate $h_\theta(x_j)$ for $j \in R$ in mini-batches of size at most $B_h$; materialize $R$ in $\mathcal{N}$ and append their new indices to OPEN with keys $w\, g_j + (1 - z_j) h_j$, preserving generation order
**end while**
**if** $\Gamma = \emptyset$ **then return** failure
**return** trajectory and actions for $i^\star = \arg\min_{i \in \Gamma} g_i$, obtained by following parent references to the root and reversing

**Host-Resident Baseline** The host-resident baseline stores each node as a record containing $s(n)$, $g(n)$, $h(n)$, $f_w(n)$, a parent pointer, and the generating action. OPEN is a binary priority queue keyed lexicographically by $(f_w, \tau)$, where $\tau$ is a monotonically increasing insertion timestamp. CLOSED is an associative map $C : \mathcal{S} \rightharpoonup \mathbb{R}_{\geq 0}$ that stores the best path cost seen so far for each state. The root node is initialized with $g = 0$, its goal indicator is evaluated immediately, its heuristic is evaluated once before the search loop, and the resulting node is inserted into OPEN. The same root state is also recorded in CLOSED with value 0, so immediate revisits to the root are suppressed.

One iteration pops up to $K$ nodes of minimum key from OPEN. If fewer than $K$ nodes remain, the iteration uses the available prefix. Every popped node is expanded under the full ordered action set $\mathcal{A}$, which yields an ordered child sequence of length at most $|\mathcal{A}|K$. Child ordering is parent major and then action major. The children of the first popped node therefore appear first, followed by the children of the second popped node, and so on. Each child receives path cost $g(\mathrm{parent}) + c(\mathrm{parent}, a)$, where $c(\mathrm{parent}, a)$ is the transition cost when taking action $a$ from state $s(\mathrm{parent})$, a parent pointer to the generating node, and the index of the generating action. Goal testing is applied to the full child sequence before any heuristic evaluations are issued.

Closed-set filtering then scans the generated children in this fixed order. If the ordered child sequence is $\{(\widetilde{s}_i, \widetilde{g}_i)\}_{i=1}^{R}$, child $i$ is retained exactly when

$$\widetilde{g}_i < C(\widetilde{s}_i), \tag{11}$$

with the convention that missing states have value $+\infty$. Whenever a child is retained, the map is updated immediately to $C(\widetilde{s}_i) \leftarrow \widetilde{g}_i$. This suppresses candidates that fail to improve the historical closed-set value and also suppresses later within-batch candidates that fail to improve the best earlier cost for the same state; a later lower-cost duplicate is retained. Only the retained children are passed to the heuristic model. Their heuristic values are evaluated in mini-batches of size $B_h$, their scores $f_w$ are computed, and the resulting nodes are inserted into OPEN in generation order. Search terminates after the first iteration whose popped batch contains at least one goal. If several goals are popped in the terminal batch, the returned solution is the popped goal of minimum path cost. The final action sequence and state trajectory are recovered by following parent pointers from that goal node back to the root and then reversing the collected chain.

**Accelerator-Resident Variant** The accelerator-resident variant preserves the same search logic but replaces object-based storage by dense tensors. Let $d$ be the encoded state dimension and let $N$ be the number of materialized nodes. The node store contains a state matrix $X \in I_{64}^{N \times d}$, path costs $g \in \mathbb{R}^N$, heuristic values $h \in \mathbb{R}^N$, ranking scores $f \in \mathbb{R}^N$, solved flags $z \in \{0, 1\}^N$, parent indices $p \in \{-1, 0, \ldots, N-1\}^N$, and parent actions $a \in \{-1, 0, \ldots, |\mathcal{A}|-1\}^N$. By default, state and parent metadata use signed 64-bit integers and cost-like quantities use 32-bit floating-point storage. Each search instance also stores OPEN as parallel tensors of node indices and their scores, maintained in insertion order, together with a CLOSED map from encoded states to best known path costs. Popping removes the lowest-score entries using a stable ordering, so equal-score ties follow insertion order. If a positive search budget $B_{\max}$ is specified, the initial CLOSED capacity is chosen as $\max\{1024, \min\{|\mathcal{A}|K, B_{\max}\}\}$. Otherwise it is chosen as $\max\{1024, |\mathcal{A}|K\}$. The root state is encoded once, inserted into the node store, inserted into OPEN, and recorded in CLOSED with value 0.

An iteration first selects the active search instances that have not yet popped a goal. For each active instance, up to $K$ node indices are removed from OPEN in stable nondecreasing order of $f$. The corresponding parent states and parent path costs are gathered from the node store. If the popped-parent batch has size $B$, we write the gathered parent matrix as $P \in I_{64}^{B \times d}$. Expansion applies all actions in $\mathcal{A}$ simultaneously through an encoded transition operator $E : I_{64}^d \times \mathcal{A} \to I_{64}^d$ and produces a child tensor $Y \in I_{64}^{B \times |\mathcal{A}| \times d}$ together with a transition-cost matrix in $\mathbb{R}^{B \times |\mathcal{A}|}$. In index notation,

$$Y_{b,m,:} = E(P_{b,:}, a_m), \tag{12}$$

where $a_m$ is the $m$-th action in the ordered action set. The accompanying transition-cost matrix stores the corresponding edge costs. Flattening along the first two axes yields

$|\mathcal{A}|B$ candidate children in the same parent-major, action-major order as the host-resident baseline. Parent indices are generated by repeating each popped node index $|\mathcal{A}|$ times, and action labels are generated by repeating the ordered action indices for every parent.

Goal testing is evaluated on the full child batch before heuristic evaluation. Closed-set filtering is then executed independently for each search instance through the fused strict-minimum update from the preceding subsection. For the candidate sequence $\{(\widetilde{s}_i, \widetilde{g}_i)\}_{i=1}^{R}$, the filter returns a Boolean mask $\chi_i$ that is true exactly when $\widetilde{g}_i$ improves both the previously stored CLOSED value for $\widetilde{s}_i$ and every earlier candidate cost in the same batch for that same state. Simultaneously, CLOSED is updated so that each key stores the smallest path cost seen so far. This single operation therefore combines historical non-improvement filtering, within-batch strict-improvement filtering, and closed-set maintenance without transferring states off the accelerator.

Only the entries with $\chi_i = 1$ are passed to the heuristic model, and the model is evaluated in mini-batches of size $B_h$ when necessary. After clipping negative predictions to zero, the retained children receive scores through $f_w$, are appended to the node store, and are reinserted into OPEN by concatenating their new node indices and scores onto the corresponding frontier tensors. As in the host-resident baseline, the search stops after the first iteration whose popped batch contains a goal. Among the goals popped in that terminal batch, the solver chooses the one with minimum path cost. It then reconstructs the solution by following parent indices until the sentinel value $-1$ is reached, collecting the associated action labels, reversing that action sequence, and decoding the stored state rows back into environment states.

## Experiments

We instantiate the study on the $3 \times 3 \times 3$ Rubik's Cube. Search is performed on one problem instance at a time, so all batching arises within a single search tree rather than across independent roots. The transition function, action ordering, and learned heuristic are identical for the host-resident and accelerator-resident variants, and both variants use the same trained network weights. The reported end-to-end runs use $w = 0.6$, $K = 10^4$, and $B_h = 10^4$ for both variants. All accelerator-based measurements are collected on a single NVIDIA H200 GPU. In the host-resident baseline, heuristic evaluation uses that same GPU, while the remaining search logic stays on the CPU. This isolates the effect of moving the rest of the search procedure onto the accelerator.

Let $\mathcal{S}$ denote the state space, let $\mathcal{A}$ denote a finite ordered action set, let $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ be the deterministic transition function, let $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{>0}$ be the transition-cost function, and let $G : \mathcal{S} \to \{0, 1\}$ be the goal test. In the Rubik's Cube domain, each state $s \in \mathcal{S}$ is represented by a sticker-permutation vector $x(s) \in \{0, \dots, 53\}^{54}$, where coordinate $i$ stores the identifier of the sticker occupying position $i$. The solved state is

$$x^\star = (0, 1, \dots, 53). \tag{13}$$

The ordered action set is

$$\mathcal{A} = (U^{-1}, U, D^{-1}, D, L^{-1}, L, R^{-1}, R, B^{-1}, B, F^{-1}, F). \tag{14}$$

Each action $a \in \mathcal{A}$ is realized by a fixed permutation $\pi_a$ of the 54 coordinates, and every move has unit cost. The transition function is therefore

$$x(T(s, a)) = x(s)_{\pi_a}, \qquad c(s, a) = 1. \tag{15}$$

Here $x(s)_{\pi_a}$ denotes the vector obtained by permuting the coordinates of $x(s)$ according to $\pi_a$.

The learned heuristic operates on face labels rather than on raw sticker identifiers. The implementation first divides each encoded sticker identifier by 9. The network then casts this value to an integer category before applying its internal one-hot encoding. Equivalently, for coordinate $i \in \{1, \dots, 54\}$, define the face label

$$q_i(s) = \left\lfloor \frac{x_i(s)}{9} \right\rfloor \in \{0, \dots, 5\}. \tag{16}$$

Let $q(s) \in \{0, \dots, 5\}^{54}$ denote the resulting face-label vector, and let $\varphi(q(s)) \in \{0, 1\}^{54 \times 6}$ be its component-wise one-hot encoding. The network first flattens $\varphi(q(s))$ into $54 \cdot 6 = 324$ binary features, then applies two fully connected layers of widths 5000 and 1000. Each of these two layers is followed by batch normalization and a rectified linear unit. The hidden representation is then processed by four residual blocks of width 1000. Each residual block contains two fully connected layers with batch normalization, inserts a rectified linear unit after the first layer, adds the block input through an identity skip connection, and applies a final rectified linear unit after the addition. A final scalar output head produces a raw value estimate $\widehat{h}_\theta(s)$. The heuristic used by search is the clipped output

$$h_\theta(s) = \max\{0, \widehat{h}_\theta(s)\}. \tag{17}$$

The parameter vector $\theta$ is fixed during search.

We evaluate the system in two complementary ways. First, we perform end-to-end search runs on the first 100 instances from the publicly released Rubik's Cube test set used in DeepCubeA (Agostinelli et al. 2019) and DeepCubeAI (Agostinelli and Soltani 2024). For each of the two search variants, we record whether each state is solved, its wall-clock search time, its generated-node count, its solution length, and the accumulated time spent in successor generation, solved-state evaluation, duplicate filtering, heuristic evaluation, frontier insertion, frontier removal, and the full search iteration. The reported end-to-end summary therefore reflects the complete search workload over the same 100 states for both variants.

Second, we isolate the principal search stages in a real-search-backed ablation study. For this purpose, rather than benchmarking synthetic or independently sampled inputs, we capture one exact search snapshot from a real search process: iteration 250 from test state 76 of the 100-state test set. We then reuse the captured states, OPEN entries, CLOSED entries, candidate pairs, and path-cost values for both variants. For each matched condition, the CPU and

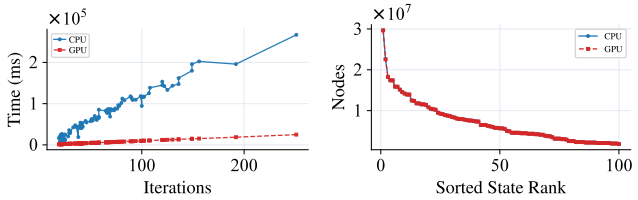
GPU measurements therefore operate on the same input sequence; larger workload points are formed by using prefixes of the captured sequence or by cyclically repeating that sequence when the requested sweep point exceeds the observed batch. We benchmark host-to-device transfer, successor generation, solved-state evaluation, duplicate filtering, frontier insertion, and frontier removal. For the batched stages, we sweep batch size over 1, 10, $10^2$, $10^3$, $10^4$, $10^5$, and $10^6$. For the structure-sensitive stages, we sweep structure size and workload over the same values, reporting both fixed-structure and fixed-workload views. Frontier removal is benchmarked over its supported workload range in this captured snapshot, which extends through $10^4$. Each ablation configuration is executed with five timed repetitions after an untimed warmup. We report mean, standard deviation, minimum, and maximum runtime. All component plots in Figures 4–6 use a logarithmic $y$-axis.

## Timing Protocol

All operation-level measurements use the same timing categories for the host-resident and accelerator-resident variants. In addition to complete wall-clock solve time, the end-to-end runs accumulate timers for successor generation (expand), solved-state evaluation (is-solved), duplicate filtering (check), heuristic evaluation (heuristic), frontier insertion (push), frontier removal (pop), and the full search-iteration boundary. For each named operation timer, we break it down into two parts: *core* and *misc*. The *core* portion is the timed primitive that implements the named search operation, and the *misc* portion is the surrounding data preparation and bookkeeping required by that primitive, including batching, conversion, materialization, device movement, and profiling-wrapper overhead. The full-iteration timer is measured independently over the solver step. Root solved-state and heuristic evaluations are included in the corresponding operation totals, whereas root node/store construction, CLOSED seeding, and initial OPEN insertion are treated as initialization outside the loop buckets. The reported runs use CUDA-aware operation timing with synchronization for both variants. Timing calls synchronize the selected CUDA device around timed regions while still preserving host wall-clock cost. In the host-resident baseline, the non-heuristic search primitives remain CPU operations, and synchronization prevents outstanding heuristic-device work from leaking across operation boundaries. We provide detailed coverage information for these boundaries in supplementary material.

# Results

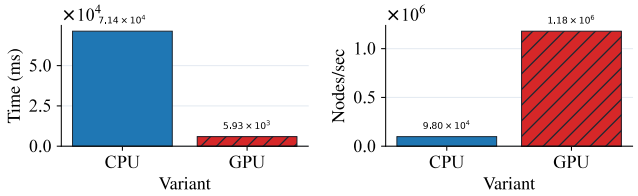
The end-to-end experiment shows that moving the full search procedure onto the accelerator substantially changes wall-clock performance without changing observed search outcomes. As Table 1 summarizes, both variants solve all 100 test states, generate approximately $6.99 \times 10^8$ nodes, use the same $6,216$ total iterations, and return the same mean solution length of 21.28. Mean solve time drops from 71.37 s to 5.931 s, a $12.03\times$ speedup, while throughput rises from $9.80 \times 10^4$ to $1.18 \times 10^6$ nodes per second. Figures 1 and 2 show that this advantage persists across the 100-state test set with the same search effort per state.




(a) Per-instance runtime after sorting the 100 test states by iteration count.

(b) Per-instance nodes generated after sorting the 100 test states by iteration count.

Figure 1: End-to-end results on the 100-state Rubik's Cube test set. GPU search is faster across the problem difficulty range without reducing search effort.



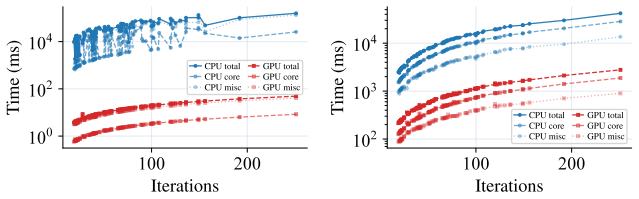

(a) Average solve time over the test states.

(b) Average node generation throughput over the test states.

Figure 2: Aggregate end-to-end comparisons on the 100-state test set. The accelerator-resident variant improves both runtime and throughput rather than merely shifting work between metrics.

The operation breakdown explains the speedup. In the host-resident baseline, successor generation accounts for $53.52\%$ of measured iteration time, followed by frontier insertion, duplicate filtering, heuristic evaluation, solved-state evaluation, and frontier removal at $13.95\%$, $13.60\%$, $12.70\%$, $3.96\%$, and $2.21\%$ respectively. In the accelerator-resident variant, successor generation falls from 38.07 s to 12.2 ms per test state, solved-state evaluation from 2.81 s to 7.1 ms, frontier insertion from 9.92 s to 10.8 ms, and frontier removal from 1.57 s to 37.8 ms. Duplicate filtering remains the largest non-heuristic residual at 702.7 ms, but total non-heuristic time drops by $80.5\times$, from 62.05 s to 770.6 ms. The accelerator-resident profile is therefore dominated by heuristic evaluation, which accounts for $86.97\%$ of iteration time, with duplicate filtering contributing $11.86\%$, and other reported stage contributing below $1\%$. Figure 3 shows that this reduction holds across the test set.

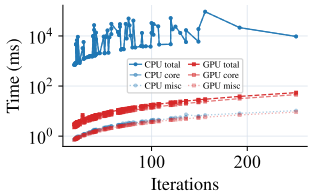
The real-search-backed component benchmarks make this separation explicit. Table 2 highlights representative points, while Figure 4 shows batch-size sweeps for regular stages. Successor generation gives the largest scaling difference: CPU time grows from $45.5\,\mu$s at batch size 1 to 9.879 s at $10^6$, whereas accelerator time stays near 15–16 $\mu$s through $10^3$, rises to $182.7\,\mu$s at $10^5$, and reaches 1.685 ms at $10^6$. The resulting speedup grows from $2.93\times$ to $5.86 \times 10^3$. Solved-state evaluation follows the same pattern with a



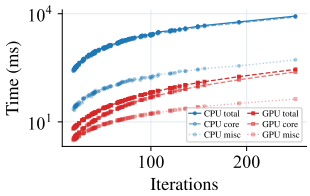

(a) Successor generation (expand).



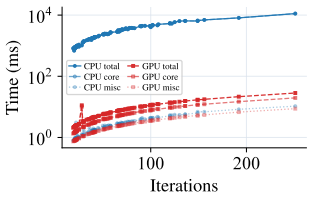

(b) Duplicate filtering (CLOSED check).




(c) Frontier insertion (OPEN push).

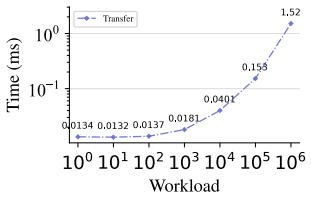



(d) Frontier removal (OPEN pop).

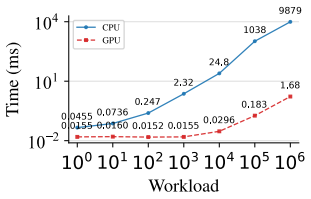



(e) Solved-state evaluation.

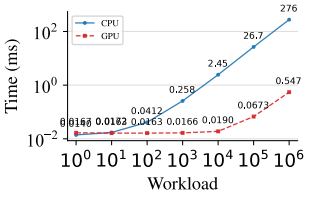
Figure 3: Per-state operation timing after sorting the 100 test states by iteration count. Each panel shows total, core, and miscellaneous timing series for the CPU and GPU variants.




(a) Host-to-device transfer.




(b) Successor generation.




(c) Solved-state evaluation.

Figure 4: Batch-size sweeps for regular stages on one real-search snapshot. CPU time grows faster than accelerator time as batch size increases.

| Var. | Mean (s) | Std. (s) | Min (s) | Max (s) | Solved | Soln. len. | Nodes/sec |
|---|---|---|---|---|---|---|---|
| CPU | 71.37 | 51.02 | 8.43 | 267.0 | 100/100 | 21.28 | 97,99 |
| GPU | **5.931** | 4.261 | 1.53 | 24.86 | 100/100 | 21.28 | 1,179,067 |

Table 1. End-to-end search summary on the 100-state Rubik's Cube test set. Both variants solve the same instances with the same mean solution length, while the accelerator-resident variant is substantially faster.

smaller absolute gain: CPU time increases from $14.0\,\mu$s to 276.3 ms, while the accelerator remains near 16–19 $\mu$s through $10^4$ and reaches $546.6\,\mu$s at $10^6$. The CPU is slightly faster at batch size 1, and the accelerator becomes faster at batch size 10, reaching $505\times$ speedup at the largest batch. The transfer panel reports the separate host-to-device copy cost.

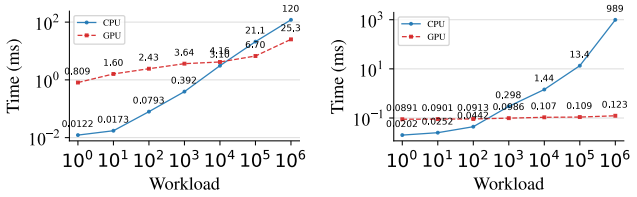
The structure-sensitive stages show threshold behavior rather than uniform improvement. Figures 5 and 6 show workload and structure-size sweeps. Duplicate filtering is the least accelerator-favorable surface: at structure size $10^6$, CPU time grows from $12.2\,\mu$s at workload 1 to 120.1 ms at workload $10^6$, while accelerator time grows from $808.7\,\mu$s to 25.3 ms. The accelerator is therefore slower at small and moderate workloads and becomes faster only between workloads $10^4$ and $10^5$, reaching $4.75\times$ speedup at workload $10^6$. This crossover shifts outward as the structure becomes smaller: at structure size $10^5$, the accelerator becomes faster only in the high-query regime. For structure sizes $10^4$ and smaller, it remains slower over the entire measured range. Frontier insertion exhibits the opposite pattern. For fixed structure size $10^6$, CPU time rises from $20.2\,\mu$s at workload 1 to 989.2 ms at workload $10^6$, whereas accelerator time stays between $89.1\,\mu$s and $123.4\,\mu$s. The accelerator is slower at workloads 1, 10, and $10^2$, becomes faster at workload $10^3$, and reaches $13.5\times$, $123\times$, and $8.02\times10^3$ speedup at workloads $10^4$, $10^5$, and $10^6$. Frontier removal, measured over its supported workload range through $10^4$, lies between these two cases. At structure size $10^6$, CPU time rises from $23.6\,\mu$s to 18.5 ms, while accelerator time remains near 0.20–0.21 ms; the accelerator becomes faster between workloads 10 and $10^2$ and reaches $88.6\times$ speedup at workload $10^4$. At smaller structures, the crossover occurs later or does not occur within this range.

Overall, results support a consistent empirical pattern. Dense and regular search stages benefit most from accelerator execution. Structure-management stages do not improve uniformly: insertion and removal become strongly accelerator-favorable after crossing workload-dependent thresholds, while duplicate filtering remains CPU-favorable over most of the practical surface and becomes accelerator-favorable only in the largest combined structure-and-workload regimes. The end-to-end measurements are consistent with this component-level picture: after the regular stages and OPEN operations are accelerated, the remaining non-heuristic residual is primarily CLOSED filtering, and the overall accelerator-resident iteration time is dominated by heuristic evaluation.

## Discussion, Limitations, and Future Work

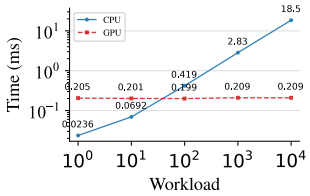
### Discussion

Moving the full batched-search pipeline onto the accelerator yields a large end-to-end gain without changing the observed



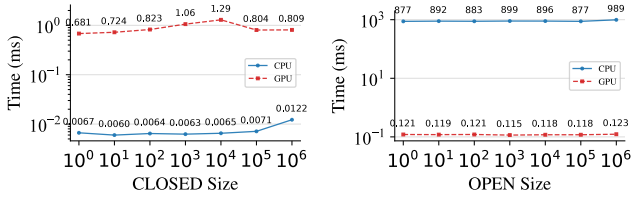

(a) Duplicate filtering (CLOSED check) at structure size $10^6$.

(b) Frontier insertion (OPEN push) at structure size $10^6$.



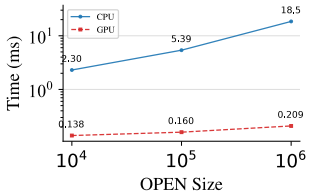

(c) Frontier removal (OPEN pop) at structure size $10^6$.

Figure 5: Workload sweeps for structure-sensitive stages at the largest measured structure size. Duplicate filtering remains CPU-favorable over much of the workload range, whereas insertion and removal become strongly GPU-favorable once the workload is large enough.




(a) Duplicate filtering (CLOSED check) at workload 1.

(b) Frontier insertion (OPEN push) at workload $10^6$.




(c) Frontier removal (OPEN pop) at workload $10^4$.

Figure 6: Structure-size sweeps for workload regimes. Frontier insertion remains nearly flat across structure sizes, whereas duplicate filtering and frontier removal retain stronger structure dependence.

| Stage | Point | CPU (ms) | GPU (ms) | Speedup |
|---|---|---|---|---|
| Succ. gen. | batch $= 10^6$ | 9879.3 | 1.685 | 5864.6× |
| Goal test | batch $= 10^6$ | 276.3 | 0.547 | 505.5× |
| Dup. filter | sz $= 10^6$, wl $= 10^6$ | 120.1 | 25.30 | 4.75× |
| Frontier ins. | sz $= 10^6$, wl $= 10^6$ | 989.2 | 0.123 | 8018× |
| Frontier rem. | sz $= 10^6$, wl $= 10^4$ | 18.51 | 0.209 | 88.6× |

Table 2. Component timings from the real-search-backed ablation. The largest gains occur in regular batched stages and large-workload OPEN operations, while CLOSED duplicate filtering improves only in the largest combined structure-and-workload regime.

search behavior on the evaluation set. As Table 1 shows, both variants solve the same 100 test instances, generate the same number of nodes, and return the same mean solution length, while the accelerator-resident variant reduces mean solve time by 12.03×. The improvement therefore comes from where the same search computation is performed.

The component results clarify which search stages benefit most. Figure 4 shows that regular batched stages such as successor generation and solved-state evaluation scale strongly on the accelerator, whereas Figures 5 and 6 show less uniform gains for structure-sensitive stages. Duplicate filtering improves only in the largest regimes, while frontier insertion and removal become accelerator-favorable after workload-dependent thresholds. Thus, after regular stages and OPEN operations are accelerated, the remaining non-heuristic runtime is primarily CLOSED filtering, and the full accelerator-resident profile is dominated by heuristic evaluation.

These results frame accelerator-native search as a whole-pipeline systems problem rather than heuristic acceleration alone. Future gains are likely to require both faster heuristic inference and better dynamic management of CLOSED under the ordering constraints required by the search algorithm. The framework can transfer to other discrete domains that provide accelerator-compatible successor generation, goal testing, and heuristic evaluation, but the realized speedup will depend on state footprint, operation regularity, heuristic cost, and batch size.

## Limitations

The host-resident baseline already evaluates the heuristic on the same accelerator, so the reported gain isolates the effect of moving the remaining search logic onto the accelerator. This is the comparison studied here, but it is not exhaustive across hybrid designs, data structures, or hardware stacks. The component ablations are also snapshot-based and should be read as controlled operation-surface measurements rather than as a full characterization of every search-iteration distribution. The evaluation focuses on runtime and throughput, and it does not characterize energy consumption, memory headroom, multi-accelerator execution, or interactions with other search algorithms.

## Future Work

Future work should study compiler-assisted optimization of the accelerator-resident pipeline, including graph capture, backend choice, full-graph execution, and dynamic-shape handling for both heuristic evaluation and tensorized search operators. A second direction is multi-accelerator scaling through Distributed Tensor (DTensor) and Device Mesh, or sharded-parameter execution. The central question is when additional parallelism outweighs the communication and synchronization costs required to preserve OPEN ordering, CLOSED updates, and the returned solution path.

## Conclusion

In this work, we introduced a framework for moving batch weighted A* search from a host-resident design to an accelerator-resident design with substantial end-to-end benefit. On the Rubik's Cube test states, our accelerator-resident implementation achieved a $12.03\times$ speedup in mean solve time while preserving the observed search outcomes of the host-resident baseline. Our ablation study explains why: regular batched operators and large-workload OPEN operations become highly accelerator-favorable, whereas CLOSED duplicate filtering remains the largest bottleneck besides the neural network heuristic evaluation in the end-to-end accelerator runs. Taken together, these results show that fully accelerator-resident search with learned heuristics is both practical and effective, but that its next frontier lies in the joint optimization of learned evaluation and dynamic data structures. The same framework can be adapted to other domains or to learned transition and world-model components when their successor, goal-test, and heuristic operations can be expressed as accelerator-compatible tensor computations.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Award No. 2426622. The authors gratefully acknowledge the computational resources provided by the Theia high performance computing cluster at the University of South Carolina which is supported by National Science Foundation Grant No. 2320292. We also acknowledge the technical assistance and resources provided by Research Computing at the University of South Carolina (RRID:SCR_027488).

## 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.; Panta, R.; and Khandelwal, V. 2024. Specifying goals to deep neural networks with answer set programming. In *34th International Conference on Automated Planning and Scheduling*.

Agostinelli, F.; and Soltani, M. 2024. Learning discrete world models for heuristic search. In *Reinforcement Learning Conference*.

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

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39: 689–743.

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

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39(1): 19–39.

Futuhi, E.; and Sturtevant, N. R. 2025. A Parallel CPU-GPU Framework for Batching Heuristic Operations in Depth-First Heuristic Search. *arXiv preprint arXiv:2507.11916*.

Greco, M.; Toro, J.; Hernández, C.; and Baier, J. A. 2023. K-focal search for slow learned heuristics. *IEEE Access*, 12: 1599–1607.

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

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195: 222–248.

Korf, R. E. 1997. 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*, AAAI'97/IAAI'97, 700–705. AAAI Press. ISBN 0-262-51095-2.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature*, 521(7553): 436.

Li, T.; Chen, R.; Mavrin, B.; Sturtevant, N. R.; Nadav, D.; and Felner, A. 2022. Optimal search with neural networks: Challenges and approaches. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 109–117.

Muppasani, B.; Pallagani, V.; Srivastava, B.; and Agostinelli, F. 2024. Comparing Rubik's Cube Solvability in Domain-Independent Planners Using Standard Planning Representations for Insights and Synergy with Upcoming Learning Methods. In *International Conference on Automated Planning and Scheduling - Heuristics and Search for Domain-Independent Planning Workshop*.

Pearl, J.; and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE transactions on pattern analysis and machine intelligence*, (4): 392–399.

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

Python Software Foundation. 2019. Dictionary object implementation using a hash table. https://github.com/python/cpython/blob/v3.8.1/Objects/dictobject.c. CPython source file Objects/dictobject.c; accessed 2026-03-23.

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

Soltani, M.; and Agostinelli, F. 2025. Stable Planning through Aligned Representations in Model-Based Reinforcement Learning. In *NeurIPS 2025 Workshop on Embodied World Models for Decision Making*.

Steele Jr., G. L.; Lea, D.; and Flood, C. H. 2014. Fast Splittable Pseudorandom Number Generators. In *Proceedings of the 29th Conference on Object-Oriented Programming, Systems, Languages and Applications*, 453–472. ACM. ISBN 978-1-4503-2585-1.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Zhou, Y.; and Zeng, J. 2015. Massively parallel A* search on a GPU. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29.