Towards Automated Discovery of God-like Folk Algorithms for Rubik's Cube

Garrett E. Katz, Naveed Tahir

Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY, 13244
{gkatz01,ntahir}@syr.edu

Abstract

We present a multi-objective meta-search procedure that constructs candidate algorithms for state-space search puzzles like Rubik's cube. The candidate algorithms take the form of macro databases, i.e., rule tables that specify sequences of actions to perform in different states. Rules are repeatedly applied until the puzzle is solved. The objectives favor candidates that are god-like (solving the puzzle in fewer steps) and folk-like (having fewer rules in the macro database). We build each candidate with a non-deterministic rule table construction, and then optimize over the non-deterministic choice points to find candidates near the Pareto-optimal trades-offs between godliness and folksiness. We prove that the rule table construction is correct: it always terminates and solves every state at termination. This is verified empirically on the full $2\times2\times2$ "pocket" cube, where correct (but unoptimized) constructions take under one hour and the total number of rules is less than 10% the number of possible states. We also empirically assess the multi-objective optimization on restricted variants of the cube with up to 29K possible states, showing relative improvements in the objectives between 14-20%. Avenues for scaling up the method in future work are discussed.

Introduction

A "god algorithm" for a puzzle like Rubik's cube is one that always transforms the puzzle into its solved state using as few actions as possible. With unlimited time and memory, a god algorithm could be implemented by a giant rule table that maps each possible state to an optimal solution path. In contrast, humans often memorize smaller rule tables that map states to leading portions of sub-optimal solution paths, and repeatedly execute rules until a given problem instance is solved. These are termed "folk" algorithms because they are feasible for humans to learn and execute.

Optimally solving the general $n \times n \times n$ Rubik's cube is known to be NP-hard (Demaine, Eisenstat, and Rudoy 2018). A classic AI approach to the $3\times3\times3$ cube used iterative deepening best-first search, guided by a pattern database heuristic that mapped states to solution path lengths (Korf 1997). This algorithm was godly (found optimal solution paths), but not folksy, because the memory requirements of the pattern database and best-first search render it infeasible for a human. In a more recent approach, the pattern database was replaced by deep reinforcement learning to approximate the heuristic function used by best-first search (Agostinelli et al. 2019; McAleer et al. 2019). This approach was fairly god-like (finding optimal solution paths 60.3% of the time), and has some relevance to human solving insofar as the heuristic function is approximated with neuro-inspired techniques. However, the best-first search component is not folk-like, and the learned weights of the function approximator are not easily distilled into algorithmic instructions that a human could understand.

Instead of mapping states to heuristic values, an alternative approach maps each state to a useful action sequence, also called a "macro" (Fikes and Nilsson 1971), that should be performed in that state (Korf 1982). Multiple states that share some common features (e.g., a subset of matching cubies) are mapped to the same macro. We refer to this approach as a "macro database." Macro databases are akin to folk algorithms, particularly if the number of records is reasonably small. They are god-like to whatever extent the macros are leading portions of optimal solution paths.

Automated discovery of macros has proven effective in several AI planning domains (Botea et al. 2005; Chrpa and Vallati 2019). This suggests a question which is the focus of this paper: whether we can automate discovery of macro databases that are both god-like and folk-like. Automated macro database discovery is a meta-search, in that it does not search for *solutions* for specific states, but rather it searches for *solvers* (in the form of macro databases) that generate solutions for any state. The meta-search is a multi-objective optimization problem (Marler and Arora 2004; Deb et al. 2000) with two objectives, because it seeks macro database algorithms that are both god-like (generating short solution paths) and folk-like (using small macro databases). The Pareto optima are those algorithms that cannot be made more godly without becoming less folksy, and vice versa.

Considerations of Pareto optimality arise naturally in multi-objective problems, where the focus shifts from the decision variable space to the objective space (Miettinen 2012). The solutions obtained are seldom optimal for all the given objectives and usually characterized by tradeoffs between competing objectives. Multi-objective optimization problems can also be transformed into one or more single-objective problems using so-called "scalarization" tech-

niques (Eichfelder 2008) - for example, taking a weighted average of multiple objective values - especially when the relative priorities of those objectives are known. Under certain conditions, solutions to multiple single-objective problems may also lie in the Pareto optimal set of the multi-objective problem. Although various scalarizations exist and some are preferable to others under certain assumptions, they all extend the methods and theories of single-objective optimization to multi-objective optimization.

In this work, we employ a recent scalarization technique with strong theoretical guarantees, based on so-called "hypervolume scalarization" (Zhang and Golovin 2020). This approach repeatedly performs single-objective optimization on randomly sampled scalarizations, drawn from a carefully crafted function family. The function family is related to the notion of dominated hypervolume in objective space. With sufficiently many repetitions, the aggregated set of solutions from the individual single-objective problems converges to the multi-objective Pareto-optimal set.

The main novelties in our approach are our method for producing candidate macro databases, and our method for optimizing scalarization functions over those candidates. Our naive first attempt, based on uniform random sampling of initial candidates, was ineffective for generating macro databases that were even correct (finding solution paths of any length), let alone god-like. In response, we designed a more sophisticated procedure, which is provably guaranteed to construct correct macro databases. The godliness and folksiness of the result (but not its correctness) depend on the non-deterministic choices made during the construction. We use a Monte-Carlo back-tracking search that compares different random instantiations of the non-deterministic choices to optimize a given scalarization. An empirical evaluation was performed on a Linux workstation with 8-core Intel i7 CPU, 32GB of RAM, Python 3.7.3, and NumPy 1.20.0 (Harris et al. 2020). All random choices use NumPy's default generator and seed, and the code to reproduce all experiments is freely available online.[1]

The focus of this paper is developing a strong theoretical foundation for our approach, rather than a high performance computing platform or speed-optimized implementation. We prove several statements regarding the correctness of our macro database construction, but our current implementation does not yet scale to the full $3 \times 3 \times 3$ Rubik's cube. In this paper's empirical results, correct (but unoptimized) constructions are limited to the smaller $2 \times 2 \times 2$ "pocket" cube with 3.7M possible states, and optimized constructions are limited to restricted variants of the pocket cube with at most 29K possible states. We conclude by discussing future work to scale up the approach.

Macro Database Algorithms

A macro database implements a partial function from states to macros. For our state representation, we number the individual "facies" (i.e., faces of individual cubies) from 1 to K, and the possible colors from 1 to 6. For an $n \times n \times n$ cube, there are n^2 facies on each side, and $K = 6n^2$ facies total. For example, the $2 \times 2 \times 2$ pocket cube has $K = 24$ facies total, 4 per side. A state s is represented by a length-K vector, with entry $s_k \in \{1, ..., 6\}$ specifying the color of the k^{th} facie. We let s^* denote the solved state, and $\mathcal{S}$ denote the set of possible states.

A macro m is a sequence of actions $\langle a_i \rangle_{i=1}^{T_m}$. The new state s' after performing action a on state s is denoted $s' = a(s)$. Similarly, the result of performing a macro m is denoted

$$s' = m(s) = a_{T_m}(a_{T_m-1}(...(a_1(s))...)). \quad (1)$$

A rule r specifies a set of states that share some common features (i.e., some matching facie colors) and a macro m_r that should be performed when any of those states are encountered. The set of states is determined by one "prototype" state S_r, and a wildcard mask $W_r \in \{0, 1\}^K$ that indicates which facies must match S_r. A value of $W_{r,k} = 1$ indicates that facie k is a "wildcard" and need not be matched for the rule to apply. Formally, the rule applies to a state s only if $(s_k = S_{r,k}) \vee (W_{r,k} = 1)$ for all k.

To guarantee correctness of our construction procedure, it is also important to store a number $\ell_r \in \mathbb{N}$ with each rule r, which specifies the length of a certain (potentially sub-optimal) solution path from S_r to s^*. Therefore, a formal macro database is a list of rules, each of which is a 4-tuple: $\mathcal{R} = \langle (S_r, W_r, m_r, \ell_r) \rangle_{r=1}^{R}$. Given any query state s, the set of all rules applicable to s is given by:

$$\big\{ r \in \{1, ..., R\} \,|\, \forall k \, (s_k = S_{r,k}) \vee (W_{r,k} = 1) \big\}. \quad (2)$$

A query function $r = \text{QUERY}(\mathcal{R}, s)$ returns one such rule r in this set, if the set is non-empty. Otherwise, QUERY returns $r =$ False, to indicate that no existing rules in $\mathcal{R}$ apply to s.

The working memory requirements of a full best-first search are not feasible for human solvers. However, it may be reasonable for a human to look one or two steps ahead (i.e., to perform a shallow breadth-first search) in order to find an applicable rule in the neighborhood of their current state s. We model this with a function

$$r', s', \mathbf{p}' = \text{RULE-SEARCH}(\mathcal{R}, s) \quad (3)$$

which performs a depth-limited breadth-first search rooted at s. The return value s' is the first nearby state found for which $\text{QUERY}(\mathcal{R}, s') \neq$ False, whereas r' is the rule returned by $\text{QUERY}(\mathcal{R}, s')$, and $\mathbf{p}'$ is the sequence of actions from s to s' in the breadth-first search tree. If the depth-limit D (fixed at 1 in this paper) is reached before any such r' is found, the rule search has failed and the returned value for r' is False.

Given a maximum solution length M, macro database $\mathcal{R}$, and initial state s, we consider a cube solving algorithm $\mathcal{A}(M, \mathcal{R}, s)$ that repeatedly calls RULE-SEARCH to find the next macro, and then applies the macro, until the cube is solved. Each action along the way is appended to a running list which is eventually returned as the solution path. The algorithm terminates with failure if at any point RULE-SEARCH fails to find an applicable rule, or if the sequence of actions performed so far exceeds M.

This solving algorithm is illustrated in Figure 1 and codified in Algorithm 1. $\langle\rangle$ denotes an empty list, $\langle x \rangle$ denotes a list with a single element x, $|x|$ denotes the length of a list

[1] https://www.github.com/garrettkatz/cubbies

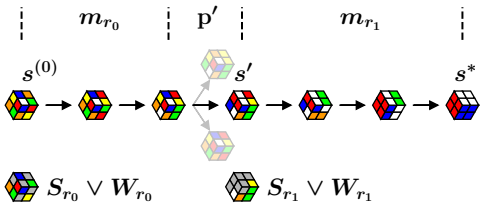

Figure 1: Illustration of Algorithm 1. Wildcard facies are shown in gray. Initial state $s^{(0)}$ matches rule r_0, triggering macro m_{r_0}. The result $m_{r_0}(s^{(0)})$ does not match any rules, but a depth-1 breadth-first rule search identifies a path $\mathbf{p}'$ to state s' that matches rule $r' = r_1$. Applying macro m_{r_1} to s' reaches the solved state s^*.

x, and $\oplus$ denotes list concatenation. The first return value v is a status flag indicating whether the algorithm terminated successfully or not. The next value $\mathbf{p} = \langle a^{(t)} \rangle_{t=1}^{T}$ with $T \leq M$ is the sequence of actions performed up until termination, successful or not. The intermediate state after performing action $a^{(t)}$ is denoted $s^{(t)}$, with initial state $s^{(0)} = s$, and final state $s^{(T)} = s^*$ if the algorithm was successful. For our macro database construction procedure, it will also prove useful to return the sequence of rules that were applied along the way, $\mathbf{r} = \langle r_n \rangle_{n=1}^{N}$, as well as each corresponding intermediate state $s^{(t_n)}$ that matched rule r_n, with $\{t_1, ..., t_n\} \subseteq \{0, 1, ..., T\}$.

Macro Database Queries

A brute-force implementation of QUERY$(\mathcal{R}, s)$ loops over every rule in $\mathcal{R}$, checking for a match with s. This takes time linear in the number of rules R, which does not scale well. We can achieve constant-time in R (and linear in K, the dimensionality of the state vectors) by instead using a prefix tree (or "trie") data structure (De La Briandais 1959; Fredkin 1960). Each edge in the trie is labeled with one of the possible facie colors $\{1, ..., 6\}$, and a trie node at depth $k \leq K$ corresponds to a prefix $(S_{r,1}, ..., S_{r,k})$ of at least one prototype state in $\mathcal{R}$. Leaf nodes correspond to a complete rule prototype and also store its associated rule index. Given this data structure, the query procedure loops over k, following the trie edge with label s_k at depth k (Algorithm 2). If at any point no such edge exists, the query state does not match the macro database and the query returns False. Otherwise, once a trie leaf is reached at depth $k = K$, the corresponding rule index r is returned.

Algorithm 2 can accommodate wildcards if we allow multiple edges in the trie between a parent and its single child, as shown in Figure 2(a). However, for simplicity, we restrict any parent with *multiple* children to have only one edge per child. Parents with multiple children must therefore be rendered "tame" by disabling the wildcard at the corresponding position in all associated rules, as shown in Figure 2(b). This taming occurs when a parent's second child is introduced by the addition of a new rule. All remaining nodes introduced by the new rule can be initially wild. We formalize this procedure as a function

$$\mathcal{R}' = \text{ADD\_RULE}(\mathcal{R}, S, m, \ell),$$

which adds a new rule with prototype state S, macro m, and path cost ℓ to an existing macro database $\mathcal{R}$. As few nodes are tamed as necessary to incorporate the new rule. The function returns $\mathcal{R}'$, the modified macro database including the new rule and modified wildcards. Note that by constraining trie edges and wildcards this way, query states will never match more than one rule. In other words, the set in Formula (2) will either be empty or a singleton, so there is no ambiguity as to which rule QUERY should return.

Algorithm 1: $\mathcal{A}(M, \mathcal{R}, s)$

Input:
M: A maximum solution path length
$\mathcal{R}$: A macro database
s: A scrambled initial state
Output:
v: True if s^* was found, False otherwise
p: The sequence of performed actions
r: The sequence of applied rules
s: The sequence of states that matched the rules

r, **s**, **p** $\leftarrow \langle\rangle, \langle\rangle, \langle\rangle$
loop
  $r', s', \mathbf{p}' \leftarrow$ RULE-SEARCH$(\mathcal{R}, s)$
  if $r' =$ False **then**
    Return False, **p**, **r**, **s**
  end if
  $s, \mathbf{r}, \mathbf{s}, \mathbf{p} \leftarrow m_r(s'), \mathbf{r} \oplus \langle r' \rangle, \mathbf{s} \oplus \langle s' \rangle, \mathbf{p} \oplus \mathbf{p}' \oplus m_r$
  if $|\mathbf{p}| > M$ **then**
    Return False, **p**, **r**, **s**
  end if
  if $s = s^*$ **then**
    Return True, **p**, **r**, **s**
  end if
end loop

Algorithm 2: QUERY$(\mathcal{R}, s)$

Input:
$\mathcal{R}$: A macro database
s: Query state
Output:
r: A rule applicable to s if one exists, False otherwise

node $\leftarrow$ root of $\mathcal{R}$'s prefix tree
for $k \in \{1, ..., K\}$ **do**
  if node.child[s_k] = None **then**
    Return False
  end if
  node $\leftarrow$ node.child[s_k]
end for
Return node.rule

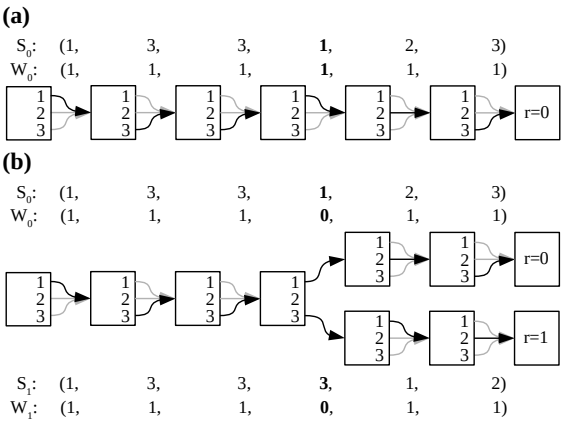

Figure 2: Prefix trees with wildcards. Edges corresponding to prototype facies are black and always persist, while gray edges support wildcards and get removed when wildcards are disabled. **(a)**: A prefix tree with a single rule and all nodes wild. **(b)** The new prefix tree after a second rule was added where $S_{0,4} \neq S_{1,4}$, which causes the branching node to be tamed and corresponding wildcards to be disabled (bold-faced numbers in the prototype and wildcard vectors). The figure depicts state vectors in $\{1,2,3\}^6$ for simplicity, as opposed to the actual pocket cube state space in $\{1,...,6\}^{24}$.

Macro Database Construction

Our initial approach created and mutated prototypes, wildcards, and macros in a rule table uniformly at random. Rule table fitness was evaluated by running $\mathcal{A}$ on a random sample of states, and a candidate population was guided towards the Pareto-optimal set with a meta-heuristic approach. While some degree of improvement was observed, we found that the best candidates were far from godly, with $\mathcal{A}$ failing on most of the random sample.

This motivated us to design a more sophisticated method for constructing macro databases. We arrived at an iterative, non-deterministic construction that provably converges to a correct macro database. The macro database is correct in that $\mathcal{A}$ will always find a solution path (though it may be suboptimal) from any scrambled state to s^* within the maximum solution length M. The total number of rules and the optimality of the solutions depend on the non-deterministic choices made during construction. Hence, this construction serves as a more solid foundation upon which the non-deterministic choices may be optimized to improve folksiness and godliness. The construction works by iteratively incorporating possible states, one at a time, into a growing macro database. If the state is not already solved properly, the macro database is modified to accommodate it, either by disabling certain wildcards (setting them to 0) or by adding new rules. States are repeatedly incorporated until every state is solved properly. Non-deterministic choices include the order in which states are incorporated, the wildcards that are disabled, and the macros used for new rules.

The sub-routine to incorporate a given state s into a partial macro database $\mathcal{R}$ is codified in Algorithm 3. The incorporation routine also requires a path from s to s^* with length at most $M - D$, although it need not be optimal. Formally, the actions and intermediate states along this path are the inputs $\langle a^{(t)} \rangle_{t=1}^{T}$ and $\langle s^{(t)} \rangle_{t=0}^{T}$, with $s^{(0)} = s$ and $s^{(T)} = s^*$. Here, the maximum solution length M is treated as hyperparameter (note that some invocations of $\mathcal{A}$ during incorporation require a smaller limit $M' < M$, line 6).

Algorithm 3: INCORPORATE($\mathcal{R}, \langle s^{(t)} \rangle_{t=0}^{T}, \langle a^{(t)} \rangle_{t=1}^{T}$)

Input:
$\mathcal{R}$: A (potentially incomplete) macro database
$\langle s^{(t)} \rangle_{t=0}^{T}, \langle a^{(t)} \rangle_{t=1}^{T}$: A path with $s^{(T)} = s^*$ and $T \leq M - D$
Output:
$\mathcal{R}$: A (potentially modified) copy of the macro database
ϕ: True if $\mathcal{R}$ was unmodified, False otherwise

1: $\langle (S_r, W_r, m_r, \ell_r) \rangle_{r=1}^{R} \leftarrow \mathcal{R}$
2: $\phi \leftarrow$ True
3: $r \leftarrow$ QUERY$(\mathcal{R}, s^{(0)})$
4: **if** $r \neq$ False **then**
5: $\quad M' \leftarrow M - (D + |m_r|)$
6: $\quad v, \mathbf{p}, \langle \overline{r}_n \rangle_{n=1}^{N}, \langle \overline{s}^{(t_n)} \rangle_{n=1}^{N} \leftarrow \mathcal{A}(M', \mathcal{R}, m_r(s^{(0)}))$
7: $\quad$ **if** $v =$ False **then**
8: $\quad\quad \phi \leftarrow$ False
9: $\quad\quad \overline{s}^{(t_0)}, \overline{r}_0 \leftarrow s^{(0)}, r$
10: $\quad\quad \omega \leftarrow \{(n,k) \mid (0 \leq n \leq N) \wedge (S_{\overline{r}_n,k} \neq \overline{s}_k^{(t_n)})\}$
11: $\quad\quad$ Choose one $(\hat{n}, \hat{k})$ from ω
12: $\quad\quad W_{\overline{r}_{\hat{n}},\hat{k}} \leftarrow 0$
13: $\quad\quad \mathcal{R} \leftarrow \langle (S_r, W_r, m_r, \ell_r) \rangle_{r=1}^{R}$
14: $\quad$ **end if**
15: **else**
16: $\quad r', s', \mathbf{p}' \leftarrow$ RULE-SEARCH$(\mathcal{R}, s^{(0)})$
17: $\quad$ **if** $r' =$ False **then**
18: $\quad\quad \phi \leftarrow$ False
19: $\quad\quad \tau \leftarrow \{(t,r) \mid (s^{(t)} = S_r) \wedge (D + t + \ell_r \leq M)\}$
20: $\quad\quad$ Choose one $(\hat{t}, \hat{r})$ from τ
21: $\quad\quad \mathcal{R} \leftarrow$ ADD_RULE$(\mathcal{R}, s^{(0)}, \langle a^{(t)} \rangle_{t=1}^{\hat{t}}, \hat{t} + \ell_{\hat{r}})$
22: $\quad$ **end if**
23: **end if**
24: Return $\mathcal{R}$, ϕ

Algorithm 3 can be conceptually split into two checks. If either check fails, $\mathcal{R}$ is modified accordingly, and ϕ is set to False to indicate that the construction is unfinished. First, lines 4-14 check for soundness, i.e., that any matched rule (line 4) initiates a successful solve by $\mathcal{A}$ (line 6). If this is not the case (line 7), one wildcard in the failing rule sequence (line 10) is disabled (line 12) so that the offending rule will no longer be matched. Next, lines 15-23 check for completeness, i.e., that any state matches at least one rule within the RULE-SEARCH depth-limit (line 16). If this is not the case (line 17), one suitable intermediate state along the path to s^* (line 19) is chosen as the endpoint for a new macro starting at $s^{(0)}$. A new rule using this macro is added to $\mathcal{R}$ (line 21). In our implementation, every choice from ω or τ is made independently and uniformly at random.

Correctness Guarantees

A sound and complete macro database can be constructed by iteratively incorporating every state in multiple passes over the state space. This is codified in Algorithm 4, which we refer to as "RCONS" for "run construction." To support Monte-Carlo back-tracking optimization (described later), we must track the history of partially complete macro databases in a list $\mathcal{H} = \langle \mathcal{R}_i \rangle_{i=1}^{I}$, where $\mathcal{R}_i$ is the partially complete database after the i^{th} modification. RCONS is initialized with a partial incomplete history and then appends new modifications to the history (line 12) until all states are solved correctly (line 15). The initial history could come from a previous, back-tracked invocation of RCONS, or it could be initialized from scratch with an empty history $\mathcal{H} = \langle\rangle$. In the latter case, the macro database is initialized with a single rule (line 2) for the solved state: its prototype is the solved state, its macro is the empty sequence, its solution length is 0, and its wildcards are all disabled ($\mathbf{0}$ denotes a vector of all 0's). States are then incorporated in multiple passes over the state space (lines 6-15). In a given pass, if any incorporations modify $\mathcal{R}$, then ϕ is False (line 9) and φ becomes False (line 10) to indicate that the construction has not yet converged. The construction must be provided with SCRAMBLES$(M - D)$, an iterator that yields pairs $(\mathbf{s}, \mathbf{p})$ suitable for INCORPORATE. That is, $(\mathbf{s}, \mathbf{p}) = (\langle s^{(t)} \rangle_{t=0}^{T}, \langle a^{(t)} \rangle_{t=1}^{T})$ is a solution path from a scrambled state $s^{(0)}$ to the solved state $s^{(T)} = s^*$, including all intermediate steps ($s^{(t+1)} = a^{(t+1)}(s^{(t)})$). To guarantee correctness, every path must have length $T \leq M - D$, and every possible state must appear as $s^{(0)}$ in one pair $(\mathbf{s}, \mathbf{p})$, but the order of iteration can be arbitrary.

The correctness proof hinges on the fact that as long as the macro database is incorrect, the construction always has choices that improve it (by disabling wildcards or adding rules). Formally, the sets on lines 10 and 19 of Algorithm 3 are always non-empty when those lines are executed. Furthermore, rule prototype states are always unique and hence the total number of rules is bounded above by the total number of possible states. Since rules are never deleted, and wildcards are only disabled (never enabled) after their rule is added, both processes are bounded and eventually monotonic, so they converge in finite time. In an extreme case (the impractical god algorithm described at the beginning of this paper), the number of rules could converge to the number of possible states, and all wildcards could eventually be disabled. However, we observe (next section) that the construction typically converges to much smaller rule sets.

Full proofs of the foregoing claims are available in the supplementary material for this paper. Here, we provide a proof sketch and state the main steps as lemmas. The first step is a lemma that new rules can always be added when needed. Intuitively, in the worst case, a macro can always be added that transforms a scrambled state all the way to the solved state. More formally:

Lemma 1. *τ is always non-empty when line 19 of Algorithm 3 is executed.*

Proving that wildcards can always be disabled when needed is more involved. For that, it is useful to first prove two auxiliary lemmas about rule *prototype* states in particular. Prototype states have special theoretical utility because they always match their own rule, regardless of the wildcard values. The first auxiliary lemma states that distinct rules have distinct prototype states:

Algorithm 4: RCONS$(\mathcal{H})$

Input:
$\mathcal{H} = \langle \mathcal{R}_i \rangle_{i=1}^{I}$: Initial modification history
Output:
$\mathcal{H}$: Updated modification history

```
1: if H = ⟨⟩ then
2:     R ← ⟨(s*, 0, ⟨⟩, 0)⟩
3: else
4:     R ← R_I
5: end if
6: repeat
7:     φ ← True
8:     for s, p ∈ SCRAMBLES(M − D) do
9:         R, φ ← INCORPORATE(R, s, p)
10:        φ ← φ ∧ φ
11:        if ¬φ then
12:            H ← H ⊕ ⟨R⟩
13:        end if
14:    end for
15: until φ
16: Return H
```

Lemma 2. *Rules created during Algorithm 4 have distinct prototype states, i.e., $S_r = S_{r'}$ implies $r = r'$.*

This is because new rules are only added with prototypes that do not already have a match. The second auxiliary lemma guarantees that each rule's macro maps its own prototype state to another rule's prototype state. This induces rule chains from prototypes to other prototypes, and eventually to s^*, within the maximum solution length and without any intermediate RULE-SEARCH. Formally:

Lemma 3. *For any rule r, there exists a sequence of rules $\langle r_n^* \rangle_{n=0}^{N}$ with $r_0^* = r$ and the following properties:*

- $S_{r_{n+1}^*} = m_{r_n^*}(S_{r_n^*})$ *for* $n < N$
- $S_{r_N^*} = s^*$, *the solved state*
- $\ell_{r_0^*} = \sum_{n=0}^{N} |m_{r_n^*}| < M - D$

This lemma holds because whenever a new rule is added, the set τ of options for that rule is properly constrained. Specifically, the endpoint of the new macro is constrained to be an existing prototype state sufficiently close to s^*.

Once all wildcards for a rule have been disabled, it will only be triggered by its prototype. Hence, the prototype rule chains above ensure that once all rules in a chain have no wildcards left to disable, the chain successfully solves its initiating state. By contrapositive, if a rule chain fails, there must be some wildcards left to disable. Therefore, we obtain the result that wildcards can always be disabled if needed:

Lemma 4. *ω is always non-empty when line 10 of Algorithm 3 is executed.*

Finally, using Lemmas 1-4 we prove that construction terminates in finite time (Proposition 1) and returns a sound and complete macro database (Proposition 2).

Proposition 1. *Construction can always proceed and terminates in finite time.*

Construction always proceeds because of Lemmas 1 and 4, which guarantee the set of choices at any decision point to be non-empty. To show termination, we first note that rules are only added, never removed, and bounded above by the total number of states (due to lemma 2). Therefore the total number of rules converges in finite time. After that point, the total number of non-zero wildcards is monotonically non-increasing (and bounded below by 0), since existing wildcards are only disabled, never enabled. Therefore the total number of non-zero wildcards also converges in finite time. After that point, the macro database never changes, ϕ is never set to False, and hence the construction terminates.

Once the construction no longer needs to disable wildcards, every matched rule initiates a successful solve, so the macro database is sound. Similarly, once there is no more need to add rules, every state neighbors at least one applicable rule, so the macro database is complete. Therefore:

Proposition 2. *When construction terminates, the returned rule table $\mathcal{R}$ is correct: i.e., for any state s, $\mathcal{A}(M, \mathcal{R}, s)$ returns a path from s to s^* in at most M actions.*

Empirical Validation

Correctness of RCONS was validated empirically on the full pocket cube, as well as several restricted variants of the pocket cube with fewer allowed actions and smaller state-spaces. For the computer experiments in this paper, we implemented SCRAMBLES by sampling possible states uniformly at random without replacement, paired with their optimal solution paths. Calculating optimal solutions for the full state space and storing it in memory is feasible for the pocket cube on modern hardware (we discuss scaling to the full $3 \times 3 \times 3$ cube in the discussion section). Note that even though we provide optimal solution paths for each scramble, only leading portions are used as macros, and the construction is still non-trivial because we aim to vastly compress the full state space into a much smaller rule table. Providing optimal paths is not theoretically required for correctness, but we found empirically that it promotes more rapid convergence and more godly results.

For empirical validation, we measure ground truth correctness at regular intervals during construction by exhaustively calling $\mathcal{A}(M, \mathcal{R}_i, s)$ on every possible state s with the current macro database $\mathcal{R}_i$, and logging the fraction of possible states that are solved successfully. Figure 3 depicts one representative run of RCONS, showing the success rate (fraction of all possible states that are solved correctly) and modification rate (fraction of incorporations where $\phi =$ False) at multiple points during construction. Modification rate was calculated within a sliding window of length 30 along the x-axis. These results confirm that the construction eventually achieves a correct macro database with 100% success rate, and we can also observe that modifications become gradually less frequent over time.

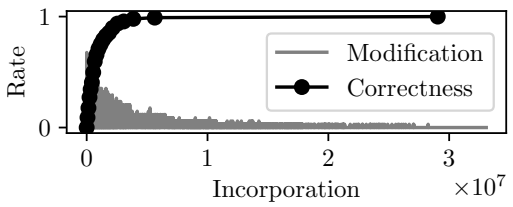

Figure 3: One representative run of RCONS on the full pocket cube. The horizontal axis indicates how many times INCORPORATE has been called. The vertical axis indicates success rate (black) and modification rate (gray) at different points during the reconstruction.

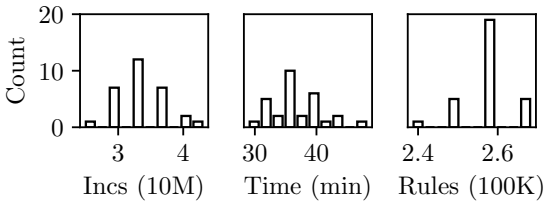

Figure 4: Histograms of total incorporations (left), total run time in minutes (middle), and total number of rules at convergence (right), aggregated from 30 independent runs of RCONS on the full pocket cube.

To gauge reproducibility, we performed 30 independent repetitions of the construction with different random instantiations of the non-deterministic choice points. All 30 repetitions eventually achieved 100% correctness. For each repetition, we also measured the total number of incorporations before convergence, the total running time in minutes, and the final number of rules in the macro database at convergence. The empirical distributions of these metrics, aggregated over the 30 repetitions, are shown in Figure 4. All runs finished within one hour, requiring at most 42 million incorporations. The final number of rules in the macro database had an average and standard deviation of 259,120 ($\pm$ 6,284), which is approximately 7% of the total pocket cube state space size (which has roughly 3.7 million reachable states).

Optimizing Godliness and Folksiness

We now turn to multi-objective optimization of godliness and folksiness. Our macro database construction involves randomness in new rule creation, wildcard restriction, and states sampled for incorporation. Different random instantiations produce different degrees of godliness and folksiness. We can optimize over these instantiations to improve the godliness or folksiness of the resulting macro database.

We formally define folksiness $F(\mathcal{R})$ based on the ratio of total rule count to the number of possible states,

$$F(\mathcal{R}) = 1 - |\mathcal{R}|/|\mathcal{S}|, \tag{4}$$

so that folksiness is 0 in the extreme case of one rule per state, and approaches 1 as the number of rules decreases.

Given a state s, we define godliness $g(\mathcal{R}, s)$ similarly, based on the ratio of actual solution length to maximum allowed solution length:

$$g(\mathcal{R}, s) = \begin{cases} 0 & \text{if } v^{(\mathcal{R},s)} = \text{False}, \\ 1 - |\mathbf{p}^{(\mathcal{R},s)}|/M & \text{otherwise}, \end{cases} \quad (5)$$

where $v^{(\mathcal{R},s)}$ and $\mathbf{p}^{(\mathcal{R},s)}$ are the success flag and solution path returned by $\mathcal{A}(M, \mathcal{R}, s)$, so that shorter solutions are more godly. The state-independent godliness $G(\mathcal{R})$ is then the expected value $\mathbb{E}_s[g(\mathcal{R}, s)]$, where s is drawn uniformly at random from the set of possible states. For computational expediency, in practice we use a Monte-Carlo estimate of G from a sample of 120 states drawn uniformly at random.

We adopt hypervolume scalarization (Zhang and Golovin 2020) as our optimization technique. The hypervolume dominated by a set of points $\hat{Y} = \{\hat{y}^{(i)}\}_{i=1}^{n}$ in $\mathbb{R}^K$ is

$$\mathcal{HV}(Y) = \text{vol}\left(\bigcup_{i=1}^{n} \{y \in \mathbb{R}^K \mid \forall k\, 0 \le y_k \le \hat{y}_k^{(i)}\}\right), \quad (6)$$

assuming each $\hat{y}_k^{(i)}$ is non-negative. Dominated hypervolume is maximized when $\hat{Y}$ are the Pareto-optimal multi-objective values. Hypervolume scalarizations take the form

$$\sigma_\lambda(y) = \min_k(\max(0, y_k/\lambda_k))^K \quad (7)$$

where $y, \lambda \in \mathbb{R}^K$. Zhang and Golovin (2020) show that

$$\mathcal{HV}(Y) \propto \mathbb{E}_{\lambda \sim S_+^{K-1}}\left[\max_{y \in Y} \sigma_\lambda(y)\right], \quad (8)$$

where S_+^{K-1} is a uniform distribution over the positive orthant of the unit hyper-sphere. This leads to a simple multi-objective algorithm with strong theoretical guarantees: one repeatedly samples a random λ and optimizes the single-valued objective $\max_x \sigma_\lambda(y(x))$. With sufficiently many samples, the union of all x encountered during optimization converges to (a superset of) the Pareto-optimal set. In our case, $x = \mathcal{R}$ and $y(x) = (F(x), G(x))$.

To use this approach, we devised a single-objective optimization routine over possible $\mathcal{R}$ via Monte-Carlo back-tracking, codified in Algorithm 5. The method is Monte-Carlo in that it sub-samples the set of all possible RCONS traces, i.e., all possible histories of random choices that could be made during construction. Each sampled trace runs to completion (line 4) and is then evaluated using the scalarization σ_λ. The maximal scalarization value found so far is saved in the variable z. The best trace so far is repeatedly back-tracked n modifications into the past and forked into another trace. If the fork performs worse than the best trace, the number of back-track steps n is increased by an integer amount Δn (line 7). Otherwise, the fork becomes the new best so far and n is reset to 1 (lines 9 and 10). Intuitively, this looks for local improvements to the trace first, and gradually looks more globally when local improvements are not found. The method stops when the trace has been back-tracked as far as possible (line 12). Similarly to RCONS, it is initialized with an incomplete macro database containing one rule for the solved state (line 1), such that line 4 in the first iteration will construct the first complete trace. The increase amount Δn is a hyperparameter: smaller values search more thoroughly but result in longer search time. We set $\Delta n = 32$ in our experiments, and stop the optimization early once 256 forks have been constructed.

Algorithm 5: Back-tracking Monte-Carlo Optimization

Input:
σ_λ: A scalarization function
Output:
$\mathcal{R}$: An optimized ruled set

1: $\mathcal{R}_1 \leftarrow \langle s^*, \mathbf{0}, \langle\rangle, 0\rangle$
2: $R, n, z \leftarrow 1, 0, -\infty$
3: **repeat**
4: $\quad \langle \mathcal{R}'_i \rangle_{i=1}^{R'} \leftarrow \text{RCONS}(\langle \mathcal{R}_i \rangle_{i=1}^{R-n})$
5: $\quad z' \leftarrow \sigma_\lambda(\mathcal{R}'_{R'})$
6: $\quad$ **if** $z' \le z$ **then**
7: $\quad\quad n \leftarrow n + \Delta n$
8: $\quad$ **else**
9: $\quad\quad \langle \mathcal{R}_i \rangle_{i=1}^{R} \leftarrow \langle \mathcal{R}'_i \rangle_{i=1}^{R'}$
10: $\quad\quad z, n \leftarrow z', 1$
11: $\quad$ **end if**
12: **until** $n \ge R$
13: Return $\mathcal{R}_R$

Algorithm 5 calls RCONS in every iteration, and potentially runs many iterations before n reaches R and the search terminates. Since each run of RCONS on the pocket cube approaches one hour, it was not practical to run Algorithm 5 on the full pocket cube using our current implementation and computing resource constraints. Instead, we empirically assessed the backtracking search on two restricted variants of the pocket cube. In the first, two of three rotational cube axes were limited to half-turns, resulting in 5040 possible states. In the second, only two of the three rotational axes were allowed to turn, resulting in roughly 29K possible states. We set $M = 30$ for the 5040 variant and $M = 50$ for the 29K variant, the idea being that larger M allows the meta-search to trade more godliness for folksiness.

Figure 5 shows the results of repeatedly running Algorithm 5 on the two restricted pocket cube variants, using different independent random scalarization weights λ in each run. The best traces found do not coincide perfectly with the empirical Pareto-optimal front, but this is not surprising given the stochasticity of the method. It is also consistent with the method of Zhang and Golovin (2020), according to which all points generated during single-objective optimization (not just the final optima) should be saved to encompass the Pareto optimal set. Comparing the best and average values for each objective over all forks, we found relative percentage improvements of roughly 18-20% for folksiness and 14-16% for godliness.

Discussion

We have presented an approach towards automated discovery of god-like folk algorithms for Rubik's cube. The

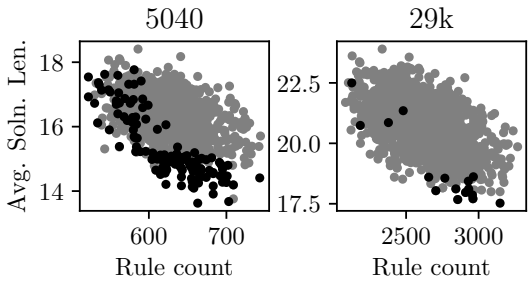

Figure 5: Final objective values for all forks in all repetitions of Monte-Carlo backtracking: 128 repetitions for the 5040 state variant (left) and 16 for the 29K state variant (right). The optimal fork in each repetition is shown in black; all other forks are shown in gray. Average solution length (i.e., godliness) is shown on the vertical axis and rule count in the final macro database (i.e., folksiness) is shown on the horizontal axis.

method converges to sound and complete algorithms which can be optimized to improve their folksiness and godliness. Currently, the method does not scale beyond restricted variants of the pocket cube. It relies on multiple passes over the full state space, uniform state sampling without replacement, and availability of optimal solutions for each state. Future work will focus on addressing these issues, scaling to the $3 \times 3 \times 3$ Rubik's cube, and assessing the methodology in other application domains.

One potential route to reducing the run-time of RCONS is visible in our experimental results (Figure 3). We observe that *almost* perfect success rate is achieved much sooner than true convergence, and that modification rate (easily measured) may provide information useful for estimating success rate (expensive to measure). With more careful statistical modeling, this could potentially be exploited to justify early stopping of RCONS and incorporate branch-and-bound techniques into the Monte-Carlo backtracking search. To deal with uniform state sampling and optimal solutions in the $3 \times 3 \times 3$ cube, it may be possible to leverage techniques for uniform sampling on large graphs (Katzir, Liberty, and Somekh 2011; Chiericetti et al. 2016), and existing optimal (but unfolksy) Rubik's cube solvers (Korf 1997). A more systematic and theoretically well-grounded hyperparameter selection than done here could also lead to performance improvements in our method.

Lastly, although our present focus is Rubik's cube, the method might also be applicable to other domains that admit similar state representations. For example, blocks-world problems (Nilsson 1980) with K blocks can be represented by state vectors $s \in \{0, ..., K\}^K$, where $s_k = j$ when the k^{th} block is stacked on top of the j^{th} block ($j = 0$ for the table). Future work should explore the applicability of our method to this and other benchmark planning domains, such as those from the international planning competitions (Pommerening, Torralba, and Balyo 2018).

Acknowledgements

Supported by ONR award N00014-19-1-2044.

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.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24: 581–621.

Chiericetti, F.; Dasgupta, A.; Kumar, R.; Lattanzi, S.; and Sarlós, T. 2016. On sampling nodes in a network. In *Proceedings of the 25th International Conference on World Wide Web*, 471–481.

Chrpa, L.; and Vallati, M. 2019. Improving domain-independent planning via critical section macro-operators. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7546–7553.

De La Briandais, R. 1959. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, 295–298.

Deb, K.; Agrawal, S.; Pratap, A.; and Meyarivan, T. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*, 849–858. Springer.

Demaine, E. D.; Eisenstat, S.; and Rudoy, M. 2018. Solving the Rubik's Cube Optimally is NP-complete. In *35th Symposium on Theoretical Aspects of Computer Science*.

Eichfelder, G. 2008. *Adaptive Scalarization Methods in Multiobjective Optimization*, 21–66. Vector Optimization. Springer-Verlag Berlin Heidelberg.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Fredkin, E. 1960. Trie Memory. *Commun. ACM*, 3(9): 490–499.

Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; and Oliphant, T. E. 2020. Array programming with NumPy. *Nature*, 585(7825): 357–362.

Katzir, L.; Liberty, E.; and Somekh, O. 2011. Estimating sizes of social networks via biased sampling. In *Proceedings of the 20th international conference on World wide web*, 597–606.

Korf, R. E. 1982. A Program That Learns to Solve Rubik's Cube. In *AAAI*, 164–167.

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

Marler, R. T.; and Arora, J. S. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6): 369–395.

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*.

Miettinen, K. 2012. *Nonlinear Multiobjective Optimization*, volume 12 of *International Series in Operations Research & Management Science*, 4–36. Springer Science & Business Media.

Nilsson, N. J. 1980. Principles of artificial intelligence. *Tioga, Palo Alto, CA*.

Pommerening, F.; Torralba, A.; and Balyo, T. 2018. The International Planning Competition.

Zhang, R.; and Golovin, D. 2020. Random Hypervolume Scalarizations for Provable Multi-Objective Black Box Optimization. In *International Conference on Machine Learning*, 11096–11105. PMLR.