Discovering Rubik's Cube Subgroups using Coevolutionary GP – A Five Twist Experiment

Robert J. Smith
Dalhousie University,
6050 University Av., Halifax,
NS. Canada
rsmith@cs.dal.ca

Stephen Kelly
Dalhousie University,
6050 University Av., Halifax,
NS. Canada
skelly@cs.dal.ca

Malcolm I. Heywood
Dalhousie University,
6050 University Av., Halifax,
NS. Canada
mheywood@cs.dal.ca

ABSTRACT

This work reports on an approach to direct policy discovery (a form of reinforcement learning) using genetic programming (GP) for the $3 \times 3 \times 3$ Rubik's Cube. Specifically, a synthesis of two approaches is proposed: 1) a previous group theoretic formulation is used to suggest a sequence of objectives for developing solutions to different stages of the overall task; and 2) a hierarchical formulation of GP policy search is utilized in which policies adapted for an earlier objective are explicitly transferred to aid the construction of policies for the next objective. The resulting hierarchical organization of policies explicitly demonstrates task decomposition and policy reuse. Algorithmically, the process makes use of a recursive call to a common approach for maintaining a diverse population of GP individuals and then learns how to reuse subsets of programs (policies) developed against the earlier objective. Other than the two objectives, we do not explicitly identify how to decompose the task or mark specific policies for reuse. Moreover, at the end of evolution we return a population solving 100% of 17,675,698 different initial Cubes for the two objectives currently in use.

CCS Concepts

•Computing methodologies → Sequential decision making; Genetic programming;

Keywords

Genetic Programming; Coevolution; Games; Reinforcement Learning; Task Transfer

1. INTRODUCTION

The 'classic' $3 \times 3 \times 3$ Rubik's Cube (hereafter, the Rubik's Cube or Cube) represents a game of complete information consisting of a discrete characterization of states and actions. Actions typically take the form of a clockwise or counter clockwise twist (quarter turn) relative to each of the 6 cube faces, i.e. a total of 12 *atomic* actions. A Cube consists of

26 cubies of which there are 8 corner, 12 edge and 6 centre cubies; the latter never changing their position, thus defining the colour for each face. Each face consists of 9 facelets that, depending on whether they are edges or corners are explicitly connected to 1 or 2 neighbouring facelets. The total number of states is in the order of 4.3×10^{19} [16] and, unlike many continuous domains, even single actions result in a third of facelets changing position. Thus, as more cubies appear in their correct position, applying actions is more likely to increase the entropy of the Cube's state. Conversely, the Cube possesses many symmetries, thus sequences of moves can potentially define operations that move (subsets of) cubies around the Cube without displacing other subsets of cubies; or, from a group theoretic perspective, 'invariances' are identified that provide transforms between subgroups.

In short, the Rubik's Cube task has several properties that make the task an interesting candidate for solving using reinforcement learning (RL) techniques. The Cube is described by a 54 dimensional vector, or large enough to potentially result in the curse of dimensionality [26], but small enough to warrant direct application of a machine learning algorithm without requiring specialized hardware support. Moreover, the number of possible actions (12) is also higher than typically encountered in RL benchmarks, further contributing to the curse of dimensionality. Finally, given that it is already known that invariances exist for transforming the Cube between different subgroups, it seems reasonable that a learning algorithm should be capable of discovering such invariances. It is currently unknown whether RL algorithms can address these issues for the Rubik's Cube task domain. Moreover, we are not interested in adopting a solution that assumes the availability of task specific actions and/or instructions.

We investigate these questions under a coevolutionary genetic programming (GP) framework for policy search that has the capacity to incrementally construct 'policy trees' from multiple (previously evolved) programs [5, 15, 13, 12]. Thus, the term *policy tree* has nothing to do with the representation assumed for each program, but refers to the ability to construct solutions through an explicitly hierarchical organization of previously evolved code. Moreover, each 'individual' (or policy) is composed from multiple programs that learn to decompose the original task through a bidding metaphor or cooperative coevolution [18].

This study will develop an approach to transferring solutions between two consecutive subgroups representing successive fitness objectives for solving the Rubik's Cube. The resulting two level policy tree is demonstrated to produce

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. *GECCO '16 July 20-24, 2016, Denver, CO, USA*

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4206-3/16/07.

DOI: <http://dx.doi.org/10.1145/2908812.2908887>

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

a single *individual* that solves up to 80% of the scrambled Cubes, where there are 17,675,698 initial Cube states in total and each run of evolution is limited to sampling 100 Cube configurations per generation (14% of scrambled Cubes are encountered *once* during training). Moreover, diversity maintenance ensures that the population is able to cumulatively solve 100% of the scrambled Cubes. The GP representation is limited to a generic set of operators originally employed for classification tasks, thus in no way specific to the Rubik’s Cube task. We also benchmark the Sarsa RL approach with radial basis function (RBF) value function approximation under the same task, thus quantifying to what degree performance under the Rubik’s Cube task is facilitated by the formulation of subtasks versus the approach to policy discovery.

2. BACKGROUND

2.1 Reinforcement learning

There are two basic machine learning approaches for addressing the temporal sequence learning problem: (value) function optimization [11], [26] and policy search/ optimization [20]. In the case of function optimization each state–action is assumed to result in a corresponding ‘reward’ from the task domain. Such a reward might merely indicate that the learner has not yet encountered a definitive ‘failure’ condition. A reward is generally indicative of the immediate cost of the action – as opposed to the ultimate quality of the policy. In this case the goal of the temporal sequence learner is to learn the relative ‘value’ of state–action pairs such that the ‘best’ action can be chosen given the current state. Such a framework explicitly supports online adaptation [26]. Given that there are typically too many state–action pairs to exhaustively enumerate (as is the case with the Rubik’s Cube), some form of function approximation is necessary. It is also generally the case that the gradient descent style credit assignment formulations frequently employed with value function methods (such as Q-learning or Sarsa) benefit from the addition of noise to the action in order to visit a wider range of states. Moreover, an annealing schedule might also be assumed for balancing the rate of stochastic versus deterministic actions of which ϵ -greedy represents a well known approach. For benchmarking purposes, we will later employ a combination of Sarsa and ϵ -greedy exploration where this particular combination has provided state-of-the-art results on multiple benchmarks, e.g. [28].

Policy optimization, on the other hand, does not make use of value function information [20]. Instead the performance of a candidate policy/decision maker represents a direct search over the space of policies, i.e. policies do not adapt while interacting with the application. Most evolutionary methods take this form, with neuroevolutionary algorithms such as CoSyNE [8], NEAT [24] or CMA-ES (for weight optimization) [10] representing specific examples.

2.2 Rubik’s Cube solvers

A body of research has concentrated on identifying the worst case number of moves necessary to solve an $n \times n \times n$ Rubik’s Cube using exhaustive search algorithms, e.g. IDA* [16, 17]. The basic idea is to use group theory to partition the task into subgroups / subproblems. An exhaustive search is deployed over complete enumerations of each sub-

problem in order to define specific twist sequences for solving an initially scrambled Cube. Naturally, building each complete enumeration for each subgroup is expensive, particularly with respect to duplicate detection [17]. Most recently, Terabytes of storage were used by a group of researchers at Google to prove that the so called *God’s number* for the special case of $n = 3$ is 20 [21].¹

Another way of looking at this process is to note that the subgroup/subproblem defines an invariance in which only the position of subsets of cubies are of relevance. Viewed from this light, the goal of a machine learning algorithm might be to discover the nature of the transform behind an invariance. Relative to database–exhaustive enumeration, such an approach would avoid the need to construct massive databases, i.e. a memory overhead is being traded for a requirement to learn.

El-Sourani et al. adopt such an approach to provide the insight for using a genetic algorithm (GA) to discover a sequence of moves capable of moving between sets of subgroups [7]. Specifically, Thistlethwaite’s Algorithm (TWA) was adopted to define a sequence of 4 subgroups. Instead of using an exhaustive search to define the order of moves, a GA was used to search for the sequence of moves that result in changing the state of the Cube between consecutive subgroups. The caveat implicit in such an approach was that every Cube configuration required the GA to be rerun. In this work, we are interested in discovering a general policy capable of transforming multiple scrambled Cubes directly between consecutive subgroups.

Two previous works have attempted to learn general strategies for unscrambling Rubik’s Cube configurations through policy search [1, 19]. In both cases the lack of an effective cost function limited the capability of the policies learnt, i.e. the only feedback was whether the Cube was solved or not.

2.3 Incremental evolution and Task transfer

Incremental evolution is an approach first demonstrated in evolutionary robotics in which progress to the ultimate objective is not immediately feasible [9, 2]. Instead, a sequence of objectives are designed and consecutively solved with respect to a common definition for the sensors characterizing the task environment (state space). Subsequently, there have been several generalizations, including Layered Learning [25] and Task Transfer [27, 23]. Unlike incremental evolution, the later developments also considered policies that were developed under independent task environments (source tasks) and then emphasized their reuse as a starting point to solve a new (target) task. Conversely, incremental evolution emphasizes continuous refinement of the same solution across a sequence of objectives. Thus, previous approaches to incremental evolution have been demonstrated under neuroevolutionary frameworks in which the topology is fixed, but weight values continue to adapt between different objectives [9, 2].

In this work, we assume that different cycles of evolution are performed for each objective. Diversity maintenance maximizes the number of potential solutions to a task. When an objective is suitably solved (across an entire population), then the population content is ‘frozen’ and

¹Analytically it has been shown that any specific Rubik’s Cube configuration may be solved with a cost of $\Theta(n^2/\log(n))$ [4]. However, finding optimal solutions to *subsets* of cubies in an $n \times n \times 1$ Rubik’s Cube is NP-hard.

a new population initialized with the next objective. The new population learns how to solve the next objective by reusing some subset of previously evolved programs (policies). Moreover, solutions take the form of policy trees in which only a fraction of the programs comprising the solution need be executed to make each decision.

In short, the approach assumed here is closer to that of task transfer than incremental involution, and has been demonstrated under the task of multi-agent half-field offense HFO [14]. However, the HFO task has completely different properties, emphasizing policy discovery under a real-valued state space (albeit of a much lower state and action dimensionality than under the Rubik’s Cube), with an emphasis on the relative contribution of diversity management versus incorporating source tasks from different environments. Conversely, the Cube (at least as played here) does not introduce noise into states or action actuators and (unlike HFO) assumes source tasks with common state and action spaces. With this in mind, we adopt as our starting point the original architecture of hierarchical SBB [5, 15, 13, 12] and investigate whether general policies can be evolved for transfer/reuse between consecutive task objectives.

3. GP METHODOLOGY

As noted above, El-Sourani et al. identify a sequence of four fitness functions corresponding to the consecutive subgroups associated with TWA [7]. However, in limiting themselves to a GA, each Cube start state requires a completely new evolutionary run in order to return a solution in the form of a corresponding sequence of moves. In this work, we assume a similar approach to the formulation of fitness functions, but with the goal of rewarding the identification of *policies* transforming between consecutive subgroups. This will be detailed in the first of the following subsections.

The general framework for incrementally constructing a hierarchy of programs (policy tree) will then be summarized or Symbiotic Bid-based GP (SBB) [5, 15, 13, 12]. SBB has previously been demonstrated under various continuous policy search domains (e.g., pin-ball, Acrobot); hence, there is some expectation that it will also be appropriate for task transfer under the Rubik’s Cube. Specifically, SBB will be deployed in two independent cycles to build each level of the policy tree under separate objectives, thus synonymous with the transfer of previous policies under different contexts. Moreover, the second cycle of evolution needs to successfully identify the relevant policies for reuse/transfer from the first cycle, i.e. a switching policy is evolved to select between a set of previously evolved policies.

3.1 Formulating fitness for task transfer

The first three subgroups for the Rubik’s Cube task under TWA (e.g., [7]) will take the form of the following two objectives:

Subgroup 1 – Source task Orient all the 12 edge cubies, where this does not imply correct position. Face colours are defined by the centre facelet of each face, as these never rotate. Thus, edge orientation without position implies that an edge is aligned with the correct faces, but not necessarily with colours matching. For example, a red–blue edge might be aligned with the red and blue faces, but with the red facelet matched with the blue face and blue facelet on the red face.

Subgroup 2 – Target task Position all the 12 edge cubies correctly and orient all 8 corner cubies. This implies that all 12 edges are in their correct final position and the 8 edges are on the correct edge (but not necessarily with colour alignment to the correct centre facelet). This actually represents a combination of objectives 2 and 3 as originally employed by [7].

Obviously, both of the above tasks denote a set of Rubik’s Cube states. In order to explicitly define these states and provide the basis for quantifying how efficiently solutions are found, we adopt the following process.

1. Start with a Rubik’s Cube in the final *ultimate* solved state and construct a database consisting of *all* 1 through 10 quarter twist Cube configurations. Such a database consists of $\approx 7.5 \times 10^9$ states [21].
2. Query the database to locate the Cube configurations conforming to Subgroup 1 (source task). Valid solutions to the source task must be to one of these states.
3. Relative to the configurations of the source task (Subgroup 1), query the database to identify all Cube configurations that lie 1 through 5 quarter twists away from this subgroup. There are over 50 million 5 twist configurations alone, which is too large to query during training without significant overhead. Moreover, as the number of twists increases there are increasing numbers of duplicate states. The database used in this work consists of 17,675,698 *unique* Cube configurations that include 1 through 5 quarter twists associated with this subgroup.
4. Target task (subgroup 2) is a further database search for the subset of the 17,675,698 configurations that satisfy the additional constraint for perfect position of the 12 edges, but with the 8 corners now limited to being in their respective orientations.
5. Relative to the configurations of the target task (Subgroup 2), query the database to identify all Cube configurations that lie 1 through 5 quarter twists away from this subgroup. As subgroup 2 is more constrained, there are a smaller number of such Cube states.

The above process enables us to explicitly identify Cube states that satisfy subgroup 1 and 2 given a five twist limit from the respective goal states. An ideal fitness function would therefore be the count for the number of twists away from subgroup 1 and 2. In general this would not be feasible, however, it does provide a baseline for what the ideal fitness function performance would be. Hereafter, such a fitness function is referred to as the **ideal fitness function**.

A second formulation for fitness is defined in terms of the count for the number of edges and/or corners that fail to match the subgroup [7], or

$$G_{source}(h_i, p_k) = \min_{t=0, \dots, T} (10 \times o_{edge} | h_i, p_k, t) \quad (1)$$

where o_{edge} is the count for the number of edges that are not oriented (minimum is best) as encountered over a sequence of up to a maximum of T moves ($= 5$) as dictated relative to initial Cube state p_k and policy h_i .

Figure 1: Basic architecture of SBB. Host population define ‘teams’ of symbiont programs, e.g. $h_i = \{s_1, s_4\}$. Fitness is evaluated relative to the content of the Point population, i.e. each Point population member, p_k , defines an initial state of for the Cube.

$$G_{target}(h_i, p_k) = \min_{t=0, \dots, T} (40 \times o_{corner} + 10 \times p_{edge} | h_i, p_k, t) \quad (2)$$

where o_{corner} is the count for the number of corners that are not oriented and p_{edge} is the count for the number of miss positioned edges (see target task definition). Likewise, fitness is expressed relative to an initial Cube configuration, p_k , and up to a maximum of $T(= 5)$ moves as identified by candidate policy h_i . Hereafter, fitness based on G_{source} and G_{target} are denoted the **approximate fitness function**. This latter fitness function does not require database queries, but is unable to provide as much quantification of state as the ideal fitness function.

3.2 Symbiotic Bid-based GP

As noted above, several works have previously deployed SBB in various RL tasks. In the following we therefore summarize the properties that make SBB uniquely appropriate for task transfer under the Rubik’s Cube. A total of three populations appear in the original formulation of SBB² [19, 5, 15] as employed here: point population, host population and symbiont population, Figure 1.

The **Point population** (P) defines the initial state for a set of training scenarios against which fitness is evaluated. At each generation some fraction of Point population individuals are replaced, or the ‘point gap’ (G_P). In the Rubik’s Cube task Point individuals, p_k , represent initial states for the Cube. For simplicity, the Point population content is sampled without replacement (uniform p.d.f.) from the set of training Cube initial configurations (Section 4.1), i.e. in the case of SBB, no attempt is made to begin sampling with initial Cube states ‘close’ to the goal state.

The **Host population** (H) represent a variable length³ GA that indexes some subset of the members of the Symbiont population (S). Each host defines a team of symbiont programs that learn how to decompose a task through inter-program bidding. Fitness is only estimated at the Host population and a diversity metric is used to reduce the likelihood of premature convergence. This work retains the use of fitness sharing as the diversity metric (discussed below). As per the Point population, a fraction of the Host individuals are deterministically replaced at each generation (G_H).

The **Symbiont population** (S) consists of bid-based GP individuals that may appear in multiple hosts [18]. Each symbiont sym_i is defined by an action, $sym_i(a)$, and pro-

Algorithm 1 Host, h_i , fitness evaluation w.r.t. initial Cube configuration $p_k \in P$. $\vec{s}(t)$ is the vector summarizing Cube state (Figure 2) and t is the index denoting the number of twists applied relative to the initial Cube state.

1. Initialize state space at $t = 0 : \vec{s}(t) \leftarrow p_k$;
2. While $((\vec{s}(t)! = \text{solved Cube}) \text{ AND } (t < 5))$
 - (a) For all symbionts, sym_j , indexed by h_i execute their programs relative to the current state, $\vec{s}(t)$
 - (b) Identify the program with maximum output or $sym^* = \arg(\max_{sym_j \in h_i} [sym_j(p) \leftarrow \vec{s}(t)])$
 - (c) IF $(sym^*.a) == \text{atomic action}$
THEN update Cube state with action
 $s(t = t + 1) \leftarrow \text{apply twist}[\vec{s}(t) : sym^*.a]$
 - (d) ELSE $h_i \leftarrow sym^*.a$
GOTO Step 2.(a)
3. ApplyFitnessFunction($\vec{s}(t)$)

gram, $sym_i(p)$. Algorithm 1 summarizes the process by which host fitness is evaluated relative to a Cube configuration. Each symbiont executes its program (Step 2.(a)) and the program with maximum output ‘wins’ the right to suggest its corresponding action (Step 2.(b)). Actions are discrete and represent either a task specific atomic action (i.e., one of the 12 quarter turn twists, Step 2.(c)) or a pointer to a previously evolved host (from an earlier cycle of evolution, Step 2.(d)). Unlike point and host populations, the size of the Symbiont population floats as a function of the mutation operator(s) adding new symbionts. Moreover, after G_H host individuals are deleted, any symbiont that does not receive a Host pointer is also deleted. There is no further concept of symbiont fitness, i.e. task specific fitness is only expressed at the level of the hosts.

Note that while the source task is under evaluation there is only one level to a policy, thus Step 2.(d) of Algorithm 1 is never called. During target task evaluation a new Point, Host and Symbiont population are evolved and the set of symbiont actions now represent the hosts as evolved under the source task. Thus, during host (fitness) evaluation, Algorithm 1 Step 2.(d) is first satisfied (as opposed to Step 2.(c)) resulting in a pointer being passed to the previously evolved host. A second round of symbiont evaluation then takes place relative to the symbionts of the previously evolved host. The symbionts of this host all have atomic actions (one of 12 possible quarter turn twists), thus the ‘winning’ symbiont updates the state of the Cube, Step 2.(c).

The interaction between Point and Host population assumes a Pareto archive formulation [18, 6]. This implies that individuals are first marked as dominated or not, with dominated Hosts prioritized for replacement. Points are rewarded for distinguishing between Hosts [3]. However, the number of non-dominated individuals is generally observed to fill the population, necessitating the use of a secondary measure for ranking individuals, or diversity maintenance,

vised different diversity mechanisms and focused on supporting task transfer between different environments [13, 14].

³Hosts are initialized with a symbiont compliment sampled with uniform probability over the interval $[2, \dots, \omega]$.

²More recent works have not used a point population, de-

(a) Cube

{0–8, 9–17, 18–26, 27–35, 36–44, 45–53}

(b) Vector

Figure 2: Representation. (a) Unfolded original Cube - $\{u, d, r, l, f, b\}$ denote ‘up’, ‘down’, ‘right’, ‘left’, ‘front’, ‘back’ faces respectively. Integers $\{0, \dots, 8\}$ denote facelet. (b) Equivalent vector representation as indexed by GP individuals. Colour content of each cell is defined by the corresponding ASCII encoded character string for each of the 6 facelet colours across the ‘unfolded’ Cube. Sarsa used a linear normalization to the unit interval.

where a fitness sharing formulation is assumed [22]. Thus shared fitness, s_i , of host h_i takes the form:

$$s_i = \sum_k \left(\frac{G(h_i, p_k)}{\sum_j G(h_j, p_k)} \right)^3 \quad (3)$$

where $G(h_i, p_k)$ is the task dependent reward defining the quality of policy h_i on initial Cube configuration p_k or, in the case of the approximate fitness function, Equations (1) and (2) for source and target tasks respectively.

Given the symbiotic relation between Host and Symbiont populations, variation operators operate hierarchically [18, 6]. Thus, crossover operates on a pair of parents from the host population. Mutation at the Host population may add or delete symbionts, relative to the current Symbiont population content. Finally, Symbionts can be modified by first cloning a symbiont and then introducing variation in the symbiont program through (instruction) add/delete or instruction field specific mutation.

4. EMPIRICAL STUDY

4.1 Parameterization

The state of a Rubik’s Cube is expressed as a vector of 54 integers. Each integer expresses the facelet colour at a specific location. The relationship between an unfolded

Table 1: Generic SBB parameters. t_{max} generations are performed for each task or $2 \times t_{max}$ generations in total. Host specific variation operators (P_D, P_A) define the prob. of deleting or adding a symbiont to the current host. Symbiont specific variation operators (P_m, P_s, P_d, P_a) set the prob. of mutating an instruction field, swapping a pair of instructions, and deleting or adding an instruction respectively.

Parameter	Value
Max. Symbionts per host (ω)	9
Population size (H_{size}, P_{size})	120, 150
Gap size (G_H, G_P)	20, 50
P_D, P_A	0.1
Max. Generations (t_{max})	50,000
Symbionts	
Max. Num. instructions	64
P_m, P_s, P_d, P_a	0.1

$3 \times 3 \times 3$ Cube and the vector is summarized by Figure 2. Note that cells corresponding to $\{f4, r4, d4, \dots, b4\}$ denote the centre facelet for each face and therefore the integer indicating their respective colour never changes. Such information is retained in order to provide the basis for GP to establish the correct colour for each face.

The publicly available jSBB distribution provided the code base from which modifications were made to support policy tree discovery under the Rubik’s Cube task.⁴ The parameters for SBB are summarized in Table 1. Note that the gap size is defined such that a subset of 20 (50) of the Host (Point) population are replaced per generation (or 100 Host/Point individuals are retained per generation). Given that there are 50,000 generations performed, then the total number of unique Cube configurations encountered during training is 2,500,100.⁵ This means that, after the first generation, the subset of 20 *new* hosts are evaluated on all 150 Cube configurations. Conversely, the remaining subset of 100 Hosts already have fitness evaluated on 100 of the Points, thus this subset of Hosts only need evaluation over the 50 new Points. This corresponds to 8,000 evaluations per generation or 4×10^8 per training cycle.

After training is complete for both source and target task, all possible 5 twist configurations conforming to the union of the subgroups are used for test purposes or 17,675,698 (of which 14% were encountered *once* during training).⁶ Each policy is allowed a maximum budget of 5 twists to solve a Cube configuration. The cost of policy search (especially w.r.t. evaluation using the ideal fitness function) implied that we were limited to performing a total of 5 independent runs for each fitness function.

A symbionts per host limit, ω , of 9 is assumed, where this is less than the total number of actions (12). This reflects a desire to ‘force’ task decomposition. Moreover, different hosts are free to assume different subsets of ac-

⁴<http://web.cs.dal.ca/~mheywood/Code/>

⁵The same subset of Cube configurations are employed for all runs.

⁶The Point population has the capacity to retain 100 previous Cube initializations, but the 50 new configurations sampled at each generation are unique.

Figure 3: Training reward received by Sarsa RBF during training on subtask 1 (source task). A final reward of 85 corresponds to getting 8.5 facelets in the correct position (50,000 of 100,000 epochs shown).

tions. Thus, as the policy tree is constructed, the second cycle of evolution may well sample a cross section of individuals as previously evolved, and therefore form a policy tree that samples the entire range of actions. Operators utilized (by symbionts) are unchanged from previous work: $\{+, -, \times, \div, \cos, \exp, \log, \text{cond}\}$ where ‘cond’ is a conditional operator that switches the sign of an operand when taken [18, 6]; hence no attempt is made to craft task specific operators.

4.2 Results

4.2.1 Sarsa ε -greedy value function approximation

A Radial Basis Function (RBF) representation was assumed on account of previous successes with this architecture (e.g., [26, 28]). Parameter sweeps conducted over the subgroup 1 task⁷ (5,000 epochs) identified learning rate ($\alpha = 0.5$), exploration parameter ($\varepsilon = 0.8$) and ε annealing factor (0.9999). Additional parameter choices included: 1) the resolution of the RBFs (15,552 appear in the resulting architecture); and, 2) $\gamma = 1$ and $\lambda = 0$ where the latter followed from previous experience.

Five runs were then performed at the preferred parameter setting for 100,000 epochs (i.e., 100,000 Cube configurations). Figure 4 summarizes the resulting (average) reward received during which only 203 *source task* configurations were solved. Given that the Sarsa value function approach to RL is generally observed to converge rapidly to solutions [28], this result appears to indicate that subtask 1 cannot be generalized using Sarsa in its current form. Indeed, significant amounts of prior knowledge are generally necessary to tune the resolution of the RBF ‘tiling’ to each task [26, 28].

4.2.2 SBB

Figure 4 summarizes the performance of the resulting policy on subgroup 2 – *target task* (Eq. (2)) – in terms of both

⁷Formulated for Sarsa as follows: 1) zero reward until end of epoch, 2) final reward of: $125 - (10 \times O_{edge} + 5)$

Figure 4: Average number of Cube configurations solved at subgroup 2 (target task) by SBB. Descending curves (solid) represent average individual-wise performance. Ascending curves (dashed) represent cumulative performance.

the average individual-wise and the population-wise performance. Specifically, after the last generation on the target task, all individuals from the population are evaluated across all 17,675,698 Cube configurations. We then rank each host in descending order of the number Cube configurations solved. This provides a monotonically decreasing curve summarizing the strength of individuals in the population after evolution. However, population diversity implies that individuals do not necessarily solve the same subset of Cube configurations. Thus, relative to the individual-wise ordering of hosts, we also build a count for the total number of unique Cube configurations solved, or the monotonically ascending curve in Figure 4. The third curve represents a population of random policies (content at generation zero).

It is apparent that at generation zero the best random policy is capable of solving $\approx 6\%$ of Cube configurations and up to $\approx 10\%$ over the population as a whole. This is somewhat better than Sarsa after 100,000 training epochs on the easier subtask 1 objective. Some of this is likely a function of different representational biases, i.e. the RBF representation explicitly indexes all states, whereas GP does not. This might provide more robustness to state changes that result in $\approx 37\%$ of the state variable values changing. Conversely, the best trained SBB individuals (red and blue curves) are able to solve $\approx 80\%$ of Cube configurations, and approach solving 100% of Cube configurations using the first 20 individuals alone.

Figure 5 presents the individual-wise and cumulative curves for Ideal fitness function and the Approximate fitness function in more detail. Under the Ideal fitness function the cumulative performance is such that all 5 runs resulted in 100% of the test cases being solved by some subset of the hosts. The Approximate fitness function approaches this, with three of the 5 runs solving 100% of the test cases and the two remaining cases solving over 96% of the test cases (across the population of hosts). In short, the Approximate

(a) Ideal fitness function

(b) Approximate fitness function

Figure 5: Percent of 17,675,698 Cube configurations solved at the 2nd subgroup. Individual-wise ranking (descending) and cumulative ranking (ascending). Distribution reflects the variation across 5 different runs per experiment.

fitness function is able to approximate the Ideal fitness function sufficiently well, while being feasible to deploy in practice, i.e. the Ideal fitness function requires an exhaustive database of twist states, thus considerable prior effort to efficiently construct.

Figure 6 summarizes the architecture of a solution post training (corresponds to one of the first ranked policies on the target task). Host ‘H1’ is the switching host evolved during the target task and consists of four symbionts. Each H1 symbiont uses a unique action in the form of a reference to a host evolved during the source task (labelled H2 through H5). H1 is therefore the ‘switching’ node for the overall policy tree. Given the current state of the Cube, the four H1 symbionts execute their programs to resolve which branch to take, i.e. identifies *one* layer 1 host (H2 through H5). This implies that only symbionts associated with one host at each layer are ever executed *per state* in order to determine an action, *not* the entire tree. The selected layer 1 host executes its symbiont programs (on the same Cube state as the switching host), so identifying a specific atomic action (layer 0). In this particular case, the (layer 1) Hosts typically employ 5 to 7 symbionts. Out of a total of 12 available atomic actions, 8 are actually used by this particular individual.

5. CONCLUSION

Machine learning approaches for solving the Rubik’s Cube have to date been limited to performing a search conducted relative to a specific scrambled Cube configuration, i.e. solutions are *not* capable of solving multiple Cube configurations. This work builds on two recent developments in order to provide a starting point for evolving policies that solve *multiple* configurations of the Rubik’s Cube. The first development is the formulation of an approximate fitness function that corresponds to the subgroups of TWA. The second provides a framework for the piece-wise hierarchical construction of GP individuals under policy search or SBB.

Figure 6: Policy tree solving 80% of the Cube configurations under the target task. Layer 0 represent atomic actions. Layer 1 represent hosts indexed as actions by symbionts from the single layer 2 host. Each atomic action is defined by an xy tuple in which $x \in \{B, G, O, R, Y, W\}$ denote one of six colour Cube faces, and $y \in \{L, R\}$ denote left (counter clockwise) or right (clockwise) quarter turns.

The combination of the two leads to knowledge transfer between policies associated with consecutive subgroups.

In this work we demonstrate the feasibility of this scheme for the first two subgroups. We are able to show that the approximate fitness function provides a very close approximation to performance of an ideal fitness function (the latter based on an exhaustive database of Cube configurations); albeit limited to 1 through 5 twists away from the subtask. The latter is too expensive to deploy in practice, but can be used under the restricted task considered here. It also appears that this task is difficult for Sarsa-RBF value function approximation (fails to solve the first subtask).

The work reported here is limited to a 5 twists in order to

facilitate comparison to an ideal fitness function. Extending the framework to scrambled Cubes with more than 5 twists might also be possible, but only under the approximate fitness function, i.e. we let the point population incrementally increase the number of twists with which scrambled Cubes are sampled. Evolving policy trees of more than two layers will also be considered with the objective of identifying a single champion policy solving $> 90\%$ of Cube configurations.

6. ACKNOWLEDGEMENTS

The authors are grateful for support from the NSERC Discovery program (Canada).

7. REFERENCES

- [1] E. B. Baum and I. Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12:2743–2775, 2000.
- [2] J. Bongard. Behavior chaining – Incremental behaviour integration for evolutionary robotics. In *Proceedings of the International Conference on the Synthesis and Simulation of Living Systems*. MIT, 2008.
- [3] E. D. de Jong. A monotonic archive for pareto-coevolution. *Evolutionary Computation*, 15(1):61–93, 2007.
- [4] E. D. Demaine, M. L. Demaine, S. Eisenstat, A. Lubiw, and A. Winslow. Algorithms for solving Rubik’s Cubes. In *ESA*, volume 6942 of *LNCS*, pages 589–700, 2011.
- [5] J. A. Doucette, P. Lichodziejewski, and M. I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 97–104, 2012.
- [6] J. A. Doucette, P. Lichodziejewski, and M. I. Heywood. Symbiotic coevolutionary genetic programming. *Genetic Programming and Evolvable Machines*, 13:71–101, 2012.
- [7] N. El-Sourani, S. Hauke, and M. Borschbach. An evolutionary approach for solving the Rubik’s Cube incorporating exact methods. In *EvoApplications: Part I*, volume 6024 of *LNCS*, pages 80–89, 2010.
- [8] R. M. F. Gomez, J. Schmidhuber. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.
- [9] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.
- [10] W. Jaskowski, M. Szubert, P. Liskowski, and K. Krawiec. High-dimensional function approximation for knowledge-free reinforcement learning: A case study in SZ-Tetris. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 567–574, 2015.
- [11] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Machine Learning Research*, 4:237–285, 1996.
- [12] S. Kelly and M. I. Heywood. Genotypic versus behavioural diversity for teams of programs under the 4-v-3 keepaway soccer task. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2014.
- [13] S. Kelly and M. I. Heywood. On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In *European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 75–86. Springer, 2014.
- [14] S. Kelly and M. I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis. In *ACM Genetic and Evolutionary Computation Conference*, pages 1143–1150, 2015.
- [15] S. Kelly, P. Lichodziejewski, and M. I. Heywood. On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pages 3245–3252, 2012.
- [16] R. Korf. Finding optimal solutions to Rubik’s Cube using pattern databases. In *Proceedings of the Workshop on Computer Games (IJCAI)*, pages 21–26, 1997.
- [17] D. Kunkle and G. Cooperman. Solving the Rubik’s Cube: Disk is the new RAM. *Communications of the ACM*, 51(4):31–33, 2008.
- [18] P. Lichodziejewski and M. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *ACM Genetic and Evolutionary Computation Conference*, pages 363–370, 2008.
- [19] P. Lichodziejewski and M. Heywood. The Rubik’s Cube and GP temporal sequence learning: An initial study. In *Genetic Programming Theory and Practice VIII*, chapter 3, pages 35–54. Springer, 2010.
- [20] D. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Machine Learning Research*, 11:241–276, 1999.
- [21] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. God’s number is 20. <http://cube20.org>, 2010.
- [22] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5:1–29, 1997.
- [23] D. G. Shapiro, H. Munoz-Avila, and D. Stracuzzi. Special issue on structured knowledge transfer. *AI Magazine*, 32(1), 2011.
- [24] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
- [25] P. Stone. *Layered learning in multiagent systems*. MIT Press, 2000.
- [26] R. R. Sutton and A. G. Barto. *Reinforcement Learning: An introduction*. MIT Press, 1998.
- [27] M. E. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8:2125–2167, 2007.
- [28] S. Whiteson, M. E. Taylor, and P. Stone. Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Journal of Autonomous Agents and Multi-Agent Systems*, 21(1):1–27, 2009.