Coevolving Deep Hierarchies of Programs to Solve Complex Tasks

Robert J. Smith[1] and Malcolm I. Heywood[1]

[1]*Faculty of Computer Science, Dalhousie University, Halifax, NS. Canada*

Article originally appears at GECCO'17 under ACM copyright 2017
http://dl.acm.org/citation.cfm?doid=3071178.3071316

Abstract

Scaling genetic programming to organize large complex combinations of programs remains an under investigated topic in general. This work revisits the issue by first demonstrating the respective contributions of coevolution and diversity maintenance. Competitive coevolution is employed to organize a task in such a way that the most informative training cases are retained. Cooperative coevolution helps discover modularity in the solutions discovered and, in this work, is fundamental to constructing complex structures of programs that still execute efficiently (the policy tree). The role of coevolution and diversity maintenance is first independently established under the task of discovering reinforcement learning policies for solving Rubik's Cubes scrambled with 5-twists. With this established, a combined approach is then adopted for building large organizations of code for representing policies that solve 5 to 8-twist combinations of the Cube. The resulting 'deep' policy tree organizes hundreds of programs to provide *optimal* solutions to tens of millions of test cube configurations.

1 Introduction

Deep learning architectures emphasize the utilization of layers of neurons in order to represent increasingly abstract features, or a factorized representation [19]. Such a representation is more efficient than the single hidden layer methodology as epitomized by the support vector machine or multi-layer perceptron. One key development for constructing such 'deep learning architectures' was the adoption of a layer-wise approach to model building, although many basic issues are still under debate (e.g., is a layer wise approach even strictly necessary and what representations are most effective and when?). However, it seems safe to say that in general the deep learning architecture is here to stay.

In this work, we are interested in learning whether very deep organizations of programs can be usefully discovered and efficiently deployed. The mechanism adopted takes the form of the repeated application of a genetic programming (GP) metaphor for coevolving teams of programs. Such an approach provides for the bottom up development of a hierarchy of teams. Each team level is the result of an independent *cycle of evolution*. The resulting hierarchy consists of tens of layers of teams, reminiscent of a neural-like topology (each node is a team of programs, with many teams at the lowest level, and a single team at the top). However, once evolved, the layers are descended *top-to-bottom* with only a single team being executed at each level. The action resolved at each level is to decide which team in the next lower level to execute next. On reaching 'level 0' the final team's action takes the form of a task specific atomic action (e.g. class label, discrete joystick direction). As such only a small fraction of the architecture is executed per decision, making such an architecture particularly efficient to execute despite its overall complexity.

Our underlying interest is to scale GP to increasingly difficult tasks. However, we also recognize the need to structure the task such that incrementally more difficult instances (of the task) are encountered as additional levels are evolved. Two basic mechanisms will be adopted for this purpose, competitive coevolution and incremental evolution. Competitive coevolution will be employed to look after the *low-level* caching of 'interesting' training configurations. Task transfer will guide the high-level complexification of the task and corresponding cycles of evolution by introducing sub-tasks of increasing difficulty as the capability of solutions improves.

The task domain adopted in order to illustrate the potential of the approach takes the form of developing policies to solve the Rubik's Cube. Factors influencing this choice include: 1) adding additional twists (relative to the solved Cube configuration) is synonymous with increasing task difficulty, thus the task itself is incrementally

tuneable. Specifically, each additional twist of the Cube introduces a requirement to transfer previously evolved policies as a starting point to solve the new problem as opposed to restarting evolution from a random population each time; 2) the goal can be defined in terms of a single unchanging objective, thus independent of model complexity or task difficulty; 3) there are a very large number of states, but there are also invariances/symmetries in the task that a successful policy should be able to identify and generalize.

In short, we desire the resulting coevolved deep program hierarchy to represent a general policy for solving progressively more difficult Rubik's Cubes through a reinforcement learning interaction with the task. Policies acquired for solving (Cube) sub-tasks over earlier levels of the architecture will be reused at later levels. Competitive coevolution will identify specific Cube configurations that are useful for guiding the development of program teams. Diversity maintenance will promote the development of multiple policies within any cycle of evolution. New cycles of evolution will be initiated when either a performance threshold, γ, for the single best individual is encountered, or a computational limit is encountered. The former condition triggers an increase in task difficulty, the latter does not. Diversity maintenance implies that multiple policies collectively solve $\geq \gamma$ of the Cube configurations. Thus, invoking a new cycle of evolution implies that single policies can potentially lever the collective capabilities of multiple policies from the previous cycle of evolution.

2 Background

2.1 Solving Rubik's Cubes

The Rubik's Cube[1] is most frequently utilized as a benchmark for informed/heuristic search strategies (e.g., [18]). The complexity of the task is well known, with in the order of 4.3×10^{19} unique states. However, it also possesses invariances that could potentially be learned as opposed to searched. Where search and learning differ is that the result of a search process is a sequence of moves that provide a preferably optimal solution to a specific start state. Changes to the start state require the search process to begin again from scratch. A good search strategy therefore provides optimal solutions in as small a time–space complexity as possible. Conversely, learning algorithms attempt to discover general policies that provide solutions to as many task configurations as possible, i.e. the policy generalizes to unseen instances of the task,[2] discovering optimal policies is an added benefit.

The work of El-Sourani et al. proposed a framework for evolving sequences of moves relative to specific scrambled Cube configurations [8]. Specifically, Group theory was used to define a sequence of four subgroups (goals) that incrementally guided the search for Cube twists that reached the goal. Each goal represented a (Cube) subgroup with, say, corner cubies restricted to be in a certain orientation/position whereas edge cubies might be undefined. The resulting sequence of actions would take the Cube from an initial scrambled state to the solution state. However, changing the initial scrambled Cube by a single action would require the entire evolutionary search process to begin from 'scratch'. Solutions were not policies, just sequences of actions applied relative to a specific start state, as in a search strategy.

In the case of evolving policies for directly solving Cube configurations, we note three works in particular. The approach of [3, 20] both encountered limitations from the fitness function. However, a recent study indicated that approximating a pair of the (Group theoretic) subgroup goals from El-Sourani et al. was sufficient to evolve GP policies for Cubes scrambled with a 5-twists [23]. Specifically, Smith et al. demonstrate that task transfer between the policies trained under the first goal (subgroup) could be transferred to that of the second (subgroup). It was also remarked that policies might be directly evolved relative to the second subgroup without the first. Section 4.2 demonstrates that this is indeed possible, and then extends the 5-twist limit to cover up to 8-twist Cubes (or a 4 order of magnitude increase in the number of states).

2.2 Task transfer and code reuse

Task transfer collectively refers to a body of research in which the basic goal is to use solutions from different yet related tasks as the starting point to leverage solutions to a new task. To do so, a task is implicitly or explicitly being decomposed because it is too difficult to solve directly. Early examples included: 1) Incremental evolution [10] in which a sequence of different goals under the same environment are sequentially solved using the same population of policies. 2) Layered learning in which a set of independent tasks are solved, one policy per task, and then recombined either by learning a switching policy or inserting them into a predefined decision tree [25].

[1] The $3 \times 3 \times 3$ puzzle will be assumed throughout and 'Cube' will hereafter refer to this specific instance of the puzzle.

[2] Learning algorithms might be used to provide a distance metric for a heuristic search strategy [11, 2], where the distance metric has a lot of impact on solution optimality. However, this is outside the purview of this work.

Examples of this in GP include prior decompositions to aid the identification of policies for keepaway soccer [12] and general solutions to the multiplexer task [1]. More recently, the scope of task transfer has been extended to include transferring between domains with differing state spaces and goals [26]. For example, a GP population might provide solutions to the keepaway soccer task and another shooting on goal, whereas a third population would learn how to transfer these tasks into policies for playing half-field offence [16].

From a GP perspective the code reuse issue has appeared most recently under the general guise of tagging. Tagging represents a mechanism by which approximate references can be achieved between calling and called code within the same run [24]. Tagging also appears as a reference mechanism for reusing code from an earlier run. Specifically, Keijzer et al. take code fragments from an earlier run and organize them into different categories on the basis of the code arity [14]. Individuals from the referencing population may only define solutions in terms of trees composed from references to tagged code. Finally, Jaśkowski et al. present a special form of crossover that identifies code snippets from an earlier run for incorporation into individuals from the present run [13]. In both cases, the authors recognize that the source of code need not be from the same task.

Generally missing in the above approaches, however, is the ability to explicitly distinguish context when incorporating previously evolved code. This is central to the approach adopted here, and explicit in say, layered learning frameworks for task decomposition, albeit with the use of prior knowledge. Our challenge is to discover how to reuse the results from previous runs without explicit information regarding what code should be used where. One framework that does explicitly address this issue is teaming with a (cooperatively) coevolved bidding mechanism [21, 27]. The output from a team is resolved by executing all the programs and identifying the single program with 'maximum' output. Such a program has won the right to suggest its action, which is just a discrete atomic action. However, such an action could also be a pointer to a previously evolved team [20], in which case, a mechanism now exists for hierarchically organizing previously evolved teams. To date such a framework has been used to construct two layers of code [20, 17, 6, 15, 16], but scaling beyond this to organize a large body of code into hierarchies of ten or more has not been demonstrated.

2.3 Symbiotic Bid-based GP

As noted above, the cooperative coevolution of programs for GP teams (hereafter symbiotic bid-based GP (SBB)) has been demonstrated as a building block for organizing code hierarchies of depth 2. As such SBB will provide the basic 'module' from which we will construct much deeper organizations of code, hence will be summarized in more detail. A total of three populations appear in the original SBB formulation [20, 6, 17] as employed here: point population, team population and bid-based program population.

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).

Team population (T) individuals assume a variable length representation that indexes some subset of the members of the (bid-based) program population (B). Thus, teams are initialized with a program compliment sampled with uniform probability over the interval $[2, ..., \omega]$. Fitness is only estimated at the Team population and a diversity metric (detailed below) used to reduce the likelihood of premature convergence. As per the Point population, a fraction of the Team individuals are deterministically replaced at each generation (G_T).

Fitness evaluation between Point and Team population assumes a Pareto archive formulation [9, 4]; thus, teams are marked as dominated or not, with dominated Teams prioritized for replacement. Points are rewarded for providing pairwise distinctions between teams, however, the number of non-dominated individuals is generally observed to fill the population. A secondary measure for ranking individuals is then employed to maintain diversity. Specifically, a fitness sharing formulation is employed [22, 9, 21]:

$$s_i = \sum_k \left(\frac{G(tm_i, p_k)}{\sum_j G(tm_j, p_k)} \right) \quad (1)$$

where $G(tm_i, p_k)$ is the task dependent reward defining the quality of policy tm_i on Cube configuration p_k.

Bid-based program population (B) consists of bid-based GP individuals that may appear in multiple teams [21]. Each bid-based individual, bGP_i, defines an action, $bGP_i.(a)$, and program, $bGP_i.(p)$. Algorithm 1 summarizes the process of evaluating each team relative to a Cube configuration. Each program 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 team (from an earlier cycle of evolution, Step 2.(d)). Unlike Point and

Algorithm 1 Evaluation of team, tm_i on initial Cube configuration $p_k \in P$. $\vec{s}(t)$ is the vector summarizing Cube state (Figure 1) and t is the index denoting the number of twists applied relative to the initial Cube state. A is the set of atomic actions, in this case the 12 quarter turn twists (clockwise–counter clockwise 90° twist to each Cube face).

1. Initialize state space or $t = 0 : \vec{s}(t) \leftarrow p_k$;
2. While $((\vec{s}(t) \neq \text{SolvedCube})$ AND $(t < MaxTwist))$
   (a) For all programs, bGP_j, indexed by tm_i execute their programs relative to the current state, $\vec{s}(t)$
   (b) Identify the program with maximum output or
   $bGP^* = \arg(\max_{bGP_j \in tm_i}[bGP_j.(p) \leftarrow \vec{s}(t)])$
   (c) IF $(bGP^*.(a) == A)$
   THEN update Cube state with action
   $s(t = t+1) \leftarrow \text{ApplyTwist}[\vec{s}(t) : bGP^*.(a)]$
   (d) ELSE $tm_i \leftarrow bGP^*.(a)$
   GOTO Step 2.(a)
3. ApplyFitnessFunction$(\vec{s}(t))$

Team populations, the size of the Program population floats as a function of the variation operator(s), Table 1 . Moreover, after G_T team individuals are deleted, any program that does not receive a Team pointer is also deleted, i.e. task specific fitness is only expressed at the level of teams.

Note that Algorithm 1 incorporates the concept of program hierarchy [20, 6, 17, 15]. Specifically, after the first cycle of evolution, we anticipate that the best team will *not* outright solve the task. However the diversity mechanism will reward teams for solving different points. Thus, a second cycle of evolution can be conducted with entirely new point, team and program populations in which the only difference is that actions in the level of a $c + 1$ cycle of evolution are pointers to teams evolved in cycle c. In this way programs can be hierarchically organized in to policy trees [20, 6, 17, 15]. Evaluation always begins at the single team from the point–team–program population combination undergoing fitness evaluation. However, until teams from level $c = 1$ are encountered, the action associated with a winning program will always be a pointer to a team at a lower level (Step 2.(d)). Hence, team evaluation is recursive until a first level team is evaluated, returning an atomic action (Step 2.(c)), in this case one of 12 quarter turn twists.

Given the symbiotic relation between Team and Program populations, variation operators operate hierarchically [21, 7]. Thus, crossover operates on a pair of parents from the team population. Mutation at the Team population may add or delete programs, relative to the current Program population content. Finally, Programs can be modified by first cloning a program and then introducing variation in the program program through (instruction) add / delete or instruction field specific mutation.

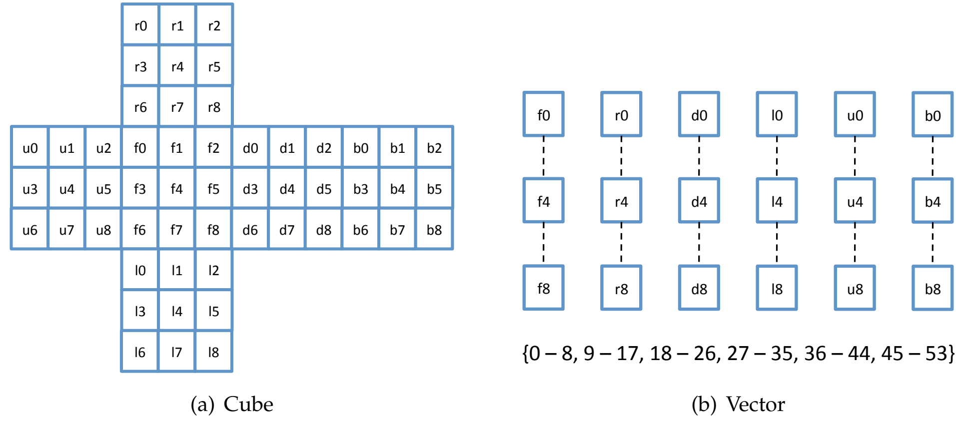
3 Evolving deep program hierarchies

This work will build on the works of [8] and [23] in the following specific ways in order to evolve policies that provide *optimal* solutions for Rubik's Cube configurations scrambled up to 8 twist:

A single subgroup will represent the goal state. This corresponds to the penultimate subgroup from [8], or G_3. The G_3 subgroup consists of 6.63×10^5 unique states. In comparison there are a total of 4.3×10^{19} unique Cube states spread over 0 to 20 twists. Moreover, transforming between G_3 and G_4 (the solution Cube configuration) only employs half turns [8].

Incremental evolution will be assumed to steadily increase the difficulty of the task (Point population content) from 5-twist to 8-twist. The earlier study of [23] demonstrated that under 5-twist Cube configurations performance of a single best policy might solve $\approx 80\%$ of the scrambled Cubes. However, other members of the population typically solve the additional 20% of Cube configurations. Providing that we are able to maintain sufficient policy diversity, adding more layers will eventually build single policies solving more of the Cube configurations.

Diversity maintenance is fundamental for ensuring that policy diversity within a layer is maintained. This will be supported through a combined approach of competitive coevolution and fitness sharing (Section 2.3). Without this we demonstrate that the quality of an individual's best solution and the cumulative performance across all



(a) Cube (b) Vector

Figure 1: Rubik's Cube Representation. (a) Unfolded original Cube - $\{u, d, r, l, f, b\}$ denote 'up', 'down', 'right', 'left', 'front', 'back' faces respectively. Integers $\{0, ..., 8\}$ denote facelet. (b) Equivalent vector representation as indexed by GP individuals. Colour content of each cell is defined by the corresponding ASCI encoded character string for each of the 6 facelet colours across the 'unfolded' Cube.

policies is significantly worse (Section 4.2.1).

Algorithm 2 summarizes the process we adopt for incrementally evolving arbitrarily deep policy trees through multiple independent cycles of evolution. At each evolutionary cycle a new level, c, of teams is added. Each team in the current cycle may represent a root node potentially solving all test cases (Figure 2). However, the combination of fitness sharing and competitive coevolution has the potential to ensure that different policies at each level solve different Cubes.

On each new cycle of evolution, c, the Point and Team populations are initialized to the population size (less 'Gap' individuals). $P(c, 0)$ initialization reflects some number of twists applied to a Cube, which when $c = 1$, $twist = 5$, Step 1a. The Team population is initialized and by implication the initial Program population is initialized with twice as many programs as there are teams (the minimum team size). The main loop of Step 1c performs up to t_{max} generations or stops when the best team (tm^*) solves $\gamma = 60\%$ of the Point population content.

A breeder model of evolution is assumed, thus at each generation a fixed number of points (G_P) and teams (G_T) are added and deleted, e.g. Steps 1(c)i and 1(c)vii in the case of the Point population. Adding teams implies that the variation operators are applied, potentially also resulting in the generation of new programs. Step 1(c)iii represents the inner loop during which fitness evaluation is performed and then fitness sharing enforced (Step 1(c)iv). The best team is identified (tm^*) and then used to test for adding a new cycle of evolution under: the current Cube (*twist*) difficulty; or with the next level of Cube difficulty (Step 1d).

Each time a new cycle of evolution completes, a new level is added to a policy tree. Each team in the current cycle of evolution, c, represent candidate **root nodes** for a policy tree, were Algorithm 1 defines how execution passes down through the policy tree until an atomic action is finally returned. Only atomic actions change the state of the Cube. The underlying hypothesis of this work is that the strength of policies improves as the depth increases.

The performance function identifies how close a Cube state is to the G_3 subgroup [23]. Unlike the earlier work of [23] we assume that SBB may directly identify policies finding states corresponding to the G_3 subgroup without additional guidance from a separate G_2 subgroup. Specifically, fitness is defined by:

$$G(tm_i, p_k) = \min_{t=0,...,twist} (40 \times o_c + 10 \times p_e | tm_i, p_k, t) \quad (2)$$

where o_c is the count for the number of corners that are not oriented and p_e is the count for the number of miss positioned edges. Likewise, fitness is expressed relative to an initial Cube configuration, p_k, and up to a maximum of *twist* moves as identified by candidate policy tm_i. Note that this is substituted into Eqn. (1) in order to provide a measure of behavioural fitness sharing, Step 1(c)iv.

As noted above, G_3 consists of 6.63×10^5 unique (goal) states, each of may then have twists applied to them to denote a scrambled Cube (Steps 1a and 1(c)i of Algorithm 2). Given that we are interested in developing policies finding *optimal* paths to any G_3 subgroup configuration we generate extensive databases for $twist = \{5, 6, 7, 8\}$ away from the subgroup. Indeed, this step is more expensive than the development of policies, however, it is

Algorithm 2 Process for incremental evolution of a deep policy tree. $twist(= 5)$ is the number of twists introduced to create a candidate scrambled Cube relative to the goal state. $MaxTwist(= 9)$ sets a scrambled Cube twist limit. $P(c, t)$ is the Point population content at level c generation t. $T(c, t)$ is the equivalent for the Team population. G_P, G_T are the number of points/teams removed/added per generation. tm^* records the best team performance at a scrambled Cube difficulty set by $twist$.

1. **while** ($twist < MaxTwist$)
   (a) $P(c, 0) \leftarrow \text{Seed}(P_{size} - G_P, \text{Cube}(twist))$
   (b) $T(c, 0) \leftarrow \text{Seed}(T_{size} - G_T, \omega)$
   (c) **for** ($t = 0; t < t_{max}$ AND $tm^* < \gamma; t = t + 1$)
      i. $P(c, t) \leftarrow \text{Seed}(G_P, \text{Cube}(twist))$
      ii. $T(c, t) \leftarrow \text{VarOp}(G_T)$
      iii. $\forall tm_i \in T(c, t)$ AND $\forall p_k \in P(c, t)$
         A. Eval $G(tm_i, p_k)$
      iv. $\forall tm_i \in T(c, t) :$ Eval(s_i)
      v. Rank($P(c, t)$)
      vi. Rank($T(c, t)$)
      vii. Delete($P(c, t), G_P$)
      viii. Delete($T(c, t), G_T$)
      ix. Update(tm^*)
   (d) **if** $tm^* \geq \gamma$ **then** $twist = twist + 1$
   (e) c++
   (f) Reset(tm^*)

necessary in order to guarantee the correct validation under test.

4 Empirical Study

4.1 Parameterization

The 'classic' $3 \times 3 \times 3$ Rubik's Cube is composed from 26 cubies of which 8 are corner, 12 are edge and 6 are centre cubies. Centre cubies have a fixed location, thus define the colour for each face. Given this overall composition there are a total of $8 \times 3 + 12 \times 2 + 6 = 54$ facelets defining the state of any cube configuration. Figure 1 summarizes the relation between the vector of 54 integers as 'seen' by GP and an unwrapped Cube. Each set of 9 integers represents a discrete face of the cube itself. These faces are identified by their 5th integer $\{f4, r4, d4, ..., b4\}$ which represents the centre facelet and therefore the required colour for each 9 integer set (Cube face).

Parameters for SBB assume those used by [23] and represent the training effort for each cycle of evolution (Table 1). This implies that after a cold start, the $G_T = 20$ new teams are evaluated on $P_{size} = 150$ Cube configurations from the Point population. The 100 Teams carried over from the previous generation only need evaluation on the $G_P = 50$ new Points introduced at this generation. Thus, there are 8,000 evaluations per generation and a total of $t_{max} \times G_P + 100 = 2,500,100$ Cube configurations encountered (typically just once) per evolutionary cycle c.

Training commences with Cubes scrambled with $twist = 5$ quarter turn twists and finishes once $twist = 8$ is successfully solved ($MaxTwist = 9$). The number of training Cube configurations per $twist$ is always 2,500,100; which are repeated if a target Cube $twist$ cannot be solved in a single cycle of evolution. Testing is performed with an additional 15,000,000 unique Cube configurations per $twist(= \{5, 6, 7, 8\})$. In all cases, each scrambled Cube configurations needs solving by a policy in the correct number of twists in order for it to be considered solved (i.e. the solution is optimal).

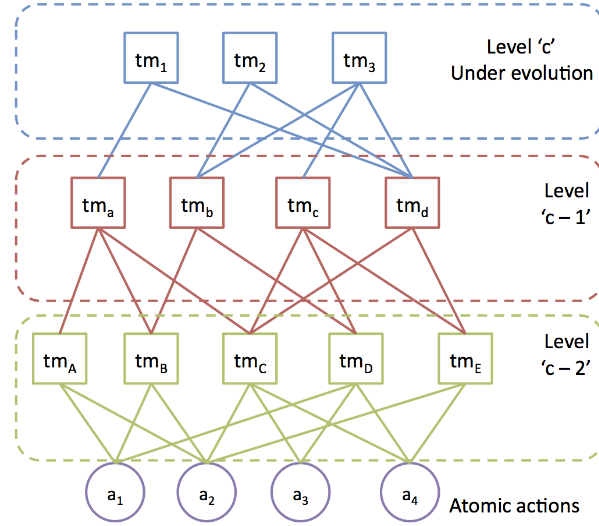


Figure 2: Illustrative example for generic structure of policy trees during evolution at level c. Teams at level c are under evolution. Teams at any other level are no longer evolved. Arcs represent program actions, e.g. tm_1 is a team with two programs with actions pointing to teams tm_a and tm_d at level $c-1$. During execution of team tm_1 either team tm_a or tm_d would be selected (Algorithm 1). The process of evaluating programs within a team would then repeat at levels $c-1$ and $c-2$. The actions at level $c-2$ take the form of atomic actions, thus applying an action to the Cube.

Table 1: SBB parameters. t_{max} generations are performed for each cycle of evolution. Team specific variation operators P_D, P_A define the prob. of deleting or adding a program. Program specific variation operators P_m, P_s, P_d, P_a define the prob. of instruction mutation, instruction swapping, and deleting or adding an instruction. 'cond' is a conditional operator that switches the sign of a operand if taken [21, 7].

Parameter	Value
Max. Programs per team (ω)	9
Population size (T_{size}, P_{size})	120, 150
Gap size (G_T, G_P)	20, 50
P_D, P_A	0.1
Max. Generations (t_{max})	50,000
Programs	
Max. Num. instructions	64
P_m, P_s, P_d, P_a	0.1
Operands	$\{+, -, \times, \div, \cos, \exp, \log, \text{cond}\}$

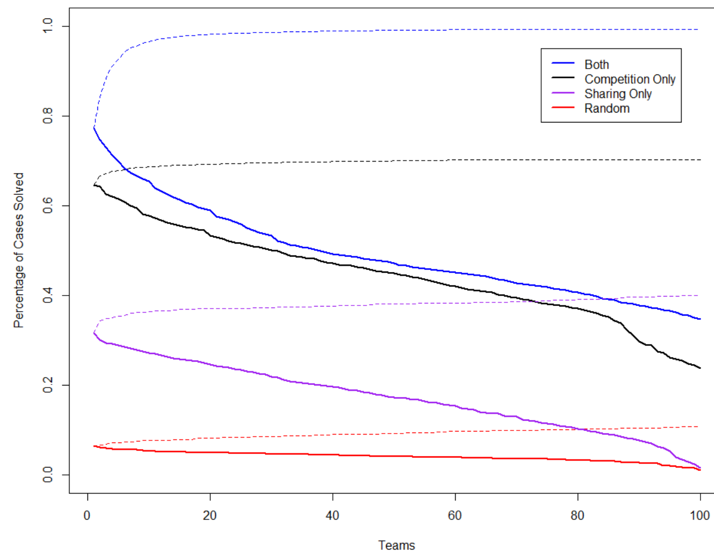


Figure 3: Median percent of 5-twist test Cubes solved across the entire population. Solid curve are individual-wise performance of team policies. Dashed curves are the cumulative performance assuming descending rank order.

4.2 Results

4.2.1 Role of diversity

In order to construct an effective deep program hierarchy (policy tree) it is necessary for there to be sufficient diversity of policies as represented by the teams at any level in the hierarchy. Without this there cannot be any leverage of policy abilities when a new cycle of evolution is invoked (see levels in Figure 2). Moreover, although 2,500,100 scrambled Cubes might be encountered during the construction of teams at any level of the policy tree, at any training epoch there are only $P_{size} = 150$ Cube configurations present. Given that we only reward a policy when it solves a Cube optimally, an efficient decomposition of the overall task between different teams at the same level of the policy tree is considered desirable. There are two basic components contributing to this:

- the ability of competitive coevolution to archive[3] useful Cube configurations for retention within the Point population between consecutive generations, and;
- fitness sharing maintaining diversity in what Cube configurations different teams (policy trees) are able to solve.

The contribution of each was validated on the first scrambled Cube twist configuration of $twist = 5$ and $gamma = 100\%$ in order to remove any other factors from the experiment. Runs were performed with: (1) *both* competitive coevolution and fitness sharing enabled or **Both**; (2) competitive coevolution enabled, but fitness sharing *disabled* or **Comp. only**; (3) competitive coevolution *disabled*, but fitness sharing enabled or **Sharing only**, i.e. G_P Cube configurations are selected for replacement entirely randomly at each generation.

Figure 3 summarizes the resulting median performance of the resulting population of policies on the 15,000,000 test Cube configurations (expressed as a percentage of Cubes that each team solved optimally). Note that both the individual-wise performance of each team is expressed and the *cumulative* performance of the population of teams (as ordered in descending individual team-wise order). For reference purposes we also included a curve for the performance of policies as initialized (no training).

A clear ranking appears with support for competitive coevolution having the biggest single impact on the quality of policies. Moreover, the (ranked) population distributions for each configuration will compared using the (two-tailed) pair-wise t-Test for: Both-versus-Comp. only; Comp. only-versus-Sharing only; and Sharing only-versus-Random initial policies (Table 2). Applying a post hoc test in the form of the Holm-Bonferroni method at the p-values from Table 2 does not change the outcome (Null hypothesis rejected, each sequentially applied configuration is significantly better than the previous).

[3]Archive size is the difference between population size and gap $P_{size} - G_P = 100$.

Table 2: Statistics for ranked population under configurations representing Both, Competition only, Sharing only and Initial population of policies as trained/tested on 5-twist Cubes. Two-tailed pair-wise t-Test with $\alpha = 0.05$.

	Both	Comp. only	Share only	Random
Mean	49.22	44.46	17.18	4.07
Variance	1.01	0.98	0.54	0.01
Pair-wise test				
Row-V-Column		Both	Comp. only	Share only
Comp. only		3.6×10^{-32}		
Share only			9.9×10^{-102}	
Random				1.05×10^{-37}

Table 3: General architectural properties of policy trees evolved as task difficulty (*twist*) increases.

Properties of policy tree				
Task	5 twist	6 twist	7 twist	8 twist
# levels (per task)	2	3	13	28
Total levels constructed	2	5	18	46
Properties of best individual				
Total # of Teams	17	62	295	887
Total # of Programs	20	115	306	919
Total # of Instructions	785	7185	19,125	54,811

It is also apparent that the population wide diversity is only maintained when both competitive coevolution and fitness sharing are present. That is to say, in anything other than the 'Both' parameterization, the cumulative curves provide little additional increase in the number of test cases solved. However, under the combined parameterization, not only is the single best individual significantly better than either of the other configurations, but the cumulative performance of the top 20 teams is sufficient for solving all test configurations. Naturally, this says nothing about how easy or difficult it might be to resolve the conditions under which switching between the policy of different teams should take place. This will be the subject of the evaluation of the framework for evolving deep program hierarchies.

4.2.2 Complexification of policies

Evolution of the deep hierarchy of programs (a deep policy tree) is an interactive process. Given the conditions for evolving a new level of the hierarchy (Section 3) multiple levels might appear for the same Cube twist count. Table 3 summarizes the development of levels as a function of Cube twist. Naturally, as the number of Cube twists increases (synonymous with task difficulty increasing) there is an increase in the number of levels added. However, what was interesting was the dramatic increase in the number of levels once we reached Cubes scrambled with 8 twists. Moreover, the number of teams/programs/instructions associated with a solution (policy) is much higher than the authors have seen reported to date in the literature. It is important to note, however, that only a fraction of the complexity embedded in a policy tree need be executed per decision. Specifically, only *one team per level* is evaluated per (state–action) decision. Thus, even in the case of the most complex policy tree the worst case number of programs executed would be $46 \times \omega = 432$. In comparison, a deep learning neural network architecture requires all neurons to be evaluated for every single decision, resulting in millions of calculations per action and a requirement for special purpose hardware support, e.g. GPUs.

Figure 4 provides an illustration of the resulting policy tree for the case of the champion after evolution against the first two Cube twists ($twist = \{5, 6\}$). The first row of nodes are not teams, but illustrate programs employed by level 1 teams. Nodes at levels 1 through 5 represent teams. Thus, the number of arcs emulating from each node to the *layer below* represent the number of programs within a team. The linkage of an arc between a node at level c and $c - 1$ represent which team is selected for execution next should the corresponding program provide the 'winning bid' (see also Figure 2 and the associated commentary).

Table 4: Percent of each Cube twist *test set* optimally solved for top 5 team policies (rank identified relative to strongest training performance on highest training level). Training level denotes the maximum level of Cube twist encountered during training. Test level defines the level of Cube twist deployed during test.

Training level	5				6				7				8			
Test level	5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
Rank 1 policy	71.7	43.2	11.9	6.5	73.6	67.5	23.5	7.9	71.4	70.1	68.6	29.4	72.6	71.1	64.8	63.5
Rank 2 policy	69.8	40.1	13.9	7.2	71.5	63.0	17.7	7.6	70.6	69.3	66.3	21.8	73.7	71.6	66.0	61.6
Rank 3 policy	67.1	34.7	10.3	7.0	66.4	61.1	23.8	4.9	71.6	70.1	60.6	26.5	75.8	72.5	64.4	57.2
Rank 4 policy	62.3	39.6	11.6	6.2	69.8	56.5	14.0	6.2	71.3	65.3	58.2	13.6	72.1	70.9	63.0	56.6
Rank 5 policy	61.3	31.6	10.8	6.7	72.0	56.2	17.5	7.0	70.8	64.2	57.3	21.2	73.6	71.6	61.5	53.9
Mean	66.5	37.8	11.7	6.7	70.6	60.9	19.3	6.7	71.1	67.8	62.2	22.5	73.6	71.5	63.9	58.6

Table 5: Ranking of the training–test treatments for the Friedman test.

Test	Training exposure			
set	5-twist	6-twist	7-twist	8-twist
5-twist	66.5 (4)	70.6 (3)	71.1 (2)	73.6 (1)
6-twist	37.8 (4)	60.9 (3)	67.8 (2)	71.5 (1)
7-twist	11.7 (4)	19.3 (3)	62.2 (2)	63.9 (1)
8-twist	6.7 (3.5)	6.7 (3.5)	22.6 (2)	58.6 (1)
Avg. Rank (R_j)	3.875	3.125	2	1

4.2.3 Generalization

So far we have demonstrated the contribution of competitive coevolution and fitness sharing within a single layer, and the non-linear relation between the number of Cube twists employed to scramble Cubes and the number of policy tree levels incrementally introduced during training. Naturally, unless the resulting models generalized over the test partition ($15,000,000$ unseen Cube configurations per twist) there would not be any point in pursuing such complexity. Table 4 summarizes performance of the top five ranked policies (from the population) over the test partition as each additional twist is added to the Cube.

The Friedman test may now be applied to establish the significance of the treatment, in this case does providing additional twists result in stronger policies [5]. Specifically, the Friedman test provides a non-parameteric alternative to the repeated measures ANOVA test. First the ranks of the average model performance are identified (Table 5), the null hypothesis that we are initially attempting to reject is that there is no pattern to the ranking. If the null hypothesis can be rejected, then the Nemenyi post hoc test will be applied to establish the significance.

The resulting Freidman statistic (χ_F^2) has a value of 11.375, whereas the number of degrees of freedom are $k = 4$ and $n = 4$. The corresponding critical value for a significance level of 0.1 is 6.3, hence the null hypothesis is rejected (i.e., $\chi_F^2 > 6.3$). Applying the Nemenyi Post Hoc test defines the critical difference: $CD = q_\alpha \sqrt{k(k+1) \div (6N)}$ by which models should differ for a significant difference to appear. Again assuming a significance level of 0.1 implies that $q_\alpha = 2.291$, resulting in a CD of 2.09. For a significant difference to appear there needs to be a difference in the pairwise average ranks of $CD + R_j$. Relative to Table 5, the 7- and 8-twist treatments are significantly different from the 5- and 6-twist cases.

In summary the following general properties are apparent: (1) During training a performance threshold for the single best individual policy $\gamma = 60\%$ was assumed. On the first instance of satisfying a new performance goal (Training level *twist*, Table 4) the resulting test performance is in the interval of $60 - 70\%$. (2) As more twists are introduced during training, the performance of policies under earlier Cube twist counts continues to see small improvements.[4] Thus, solving Cube configurations with a larger twist count does not detract from the ability to solve Cube configurations at the lower twist counts. Moreover, the average performance of the top five policies generally improves as performance improves on the later twist counts. (3) Performance on as yet unseen Cube twists (e.g. a policy trained on 5-twist Cubes, but tested on 6-, 7-, 8-twist Cubes) appears to drop off by at least 40% for each additional twist. This might be correlated with the $\approx 30\%$ change in state between consecutive twists.

[4] A 1% improvement in the number of Cube configurations solved corresponds to an additional $150,000$ Cube configurations.

5 Conclusion

A framework is proposed for incrementally evolving deep policy trees. Each 'node' of the resulting tree is a team of programs. Evolution appears in multiple independent cycles, each cycle evolving a new set of root nodes to potential policy trees. Policy trees are evolved bottom up (starting from policies consisting of a single node team at level 1). As additional cycles of evolution are performed, the new teams learn to select between teams evolved at the previous cycle. This means that large numbers of teams and programs (representing tens of thousands of instructions) can be organized, yet still be 'light weight' to execute (in comparison to, say, the computational cost of deep learning or monolithic GP).

The framework is validated using a scalable approach to solving Rubik's Cube configurations through reinforcement learning. Thus, a policy tree needs evolving in which the goal is to solve a scrambled Cube in the optimal number of steps. We demonstrate that competitive coevolution and fitness sharing are critical components of the framework. Testing on 4×15 million $= 60$ million unseen test Cubes indicates that the best single policy is able to solve $\approx 68\%$ of the test Cubes *optimally*, or 40.8 million Cubes.

Future work will consider alternative parameterizations in which the training target is increased from 60% of the training Cubes solved, to say 80% of the Cubes solved. What impact this would have on the complexity of the resulting policy trees remains to be seen.

6 Acknowledgements

Support through RUAG Schweiz AG is readily acknowledged.

References

[1] Isidro M. Alvarez, Will N. Browne, and Mengjie Zhang. Human-inspired scaling in learning classifier systems: Case study on the n-bit multiplexer problem set. In *ACM Genetic and Evolutionary Computation Conference*, pages 429–436, 2016.

[2] S. J. Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175:2075–2098, 2011.

[3] E. B. Baum and I. Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12:2743–2775, 2000.

[4] E. D. de Jong. A monotonic archive for Pareto-coevolution. *Evolutionary Computation*, 15(1):61–93, 2007.

[5] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1):1–30, 2006.

[6] J. A. Doucette, P. Lichodzijewski, and M. I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *ACM Genetic and Evolutionary Computation Conference*, pages 97–104, 2012.

[7] J. A. Doucette, A. R. McIntyre, P. Lichodzijewski, and M. I. Heywood. Symbiotic coevolutionary genetic programming: A benchmarking study under large attribute spaces. *Genetic Programming and Evolvable Machines*, 13:71–101, 2012.

[8] 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.

[9] S. G. Ficici and J. B. Pollack. Pareto optimality in coevolutionary learning. In *European Conference on Advances in Artificial Life*, volume 2159 of *LNAI*, pages 316–325, 2001.

[10] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[11] A. Hauptman, A. Elyasaf, M. Sipper, and A. Karmon. GP-Rush: Using genetic programming to evolve solvers for the rush hour puzzle. In *ACM Genetic and Evolutionary Computation Conference*, pages 955–962, 2009.

[12] William H. Hsu, Scott J. Harmon, Edwin Rodriguez, and Christopher Zhong. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer. In *Genetic and Evolutionary Computation Conference – Late Breaking Papers*, 2004.

[13] W. Jaśkowski, K. Krawiec, and B. Wieloch. Knowledge reuse in genetic programming applied to visual learning. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1790–1797, 2007.

[14] M. Keijzer, C. Ryan, and M. Cattolico. Run transferable libraries – learning functional bias in problem domains. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 3103 of *LNCS*, pages 531–542, 2004.

[15] 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.

[16] S. Kelly and M. I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis: Building simple programs for a complex task. In *ACM Genetic and Evolutionary Computation Conference*, pages 1143–1150, 2015.

[17] S. Kelly, P. Lichodzijewski, and M. I. Heywood. On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pages 3245–3252, 2012.

[18] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI Conference on Artificial Intelligence*, pages 700–705, 1997.

[19] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[20] P. Lichodzijewski and M. Heywood. The Rubic's Cube and GP temporal sequence learning: An initial study. In *Genetic Programming Theory and Practice VIII*, chapter 3, pages 35–54. Springer, 2010.

[21] P. Lichodzijewski and M. I. Heywood. Managing team-based problem solving with Symbiotic Bid-based Genetic Programming. In *ACM Genetic and Evolutionary Computation Conference*, pages 363–370, 2008.

[22] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5:1–29, 1997.

[23] Robert J. Smith, Stephen Kelly, and Malcolm I. Heywood. Discovering rubik's cube subgroups using coevolutionary gp: A five twist experiment. In *ACM Genetic and Evolutionary Computation Conference*, pages 789–796, 2016.

[24] L. Spector, K. Harrington, and T. Helmuth. Tag-based modularity in tree-based genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 815–826, 2012.

[25] P. Stone. *Layered learning in multiagent systems*. MIT Press, 2000.

[26] Matthew E. Taylor and Peter Stone. An introduction to inter-task transfer for reinforcement learning. *AI Magazine*, 32(1):15–34, 2011.

[27] S. Wu and W. Banzhaf. Rethinking multilevel selection in genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1403–1410, 2011.

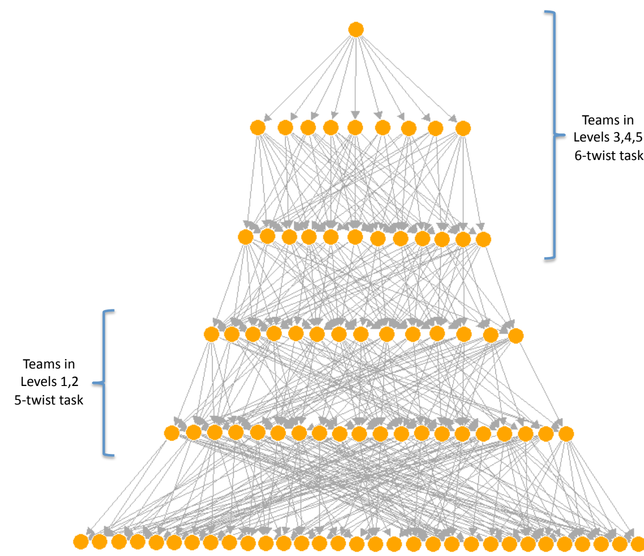


Figure 4: Graphical representation of a 6-twist policy tree where only (approximately) the 80% most used edges are accounted for.