The Rubik cube and GP Temporal Sequence learning: An initial study*

Peter Lichodzijewski and Malcolm Heywood†

20–22 May 2010

Abstract

The 3×3 Rubik cube represents a potential benchmark for temporal sequence learning under a discrete application domain with multiple actions. Challenging aspects of the problem domain include the large state space and a requirement to learn invariances relative to the specific colours present. The latter element of the domain making it difficult to evolve individuals that learn 'macro-moves' relative to multiple cube configurations. An initial study is presented in this work to investigate the utility of Genetic Programming capable of layered learning and problem decomposition. The resulting solutions are tested on 5,000 test cubes, of which specific individuals are able to solve up to 350 (7 percent) cube configurations and population wide behaviours are capable of solving up to 1,200 (24 percent) of the test cube configurations. It is noted that the design options for generic fitness functions are such that users are likely to face either reward functions that are very expensive to evaluate or functions that are very deceptive. Addressing this might well imply that domain knowledge is explicitly used to decompose the task to avoid these challenges. This would augment the described generic approach currently employed for Layered learning/ problem decomposition.

*Paper published in Genetic Programming Theory and Practice - VIII: 2010, K. Vladislavleva, T. McConaghy and R. Riolo (eds) Genetic and Evolutionary Computation Series – Copyright 2010 Springer-Verlag

†Faculty of Computer Science, Dalhousie University. Halifax. NS. Canada.

1 Introduction

Evolutionary Computation as applied to temporal sequence learning problems generally assumes a phylogenetic framework for learning [1]. That is to say, policies are evaluated in their entirety on the problem domain before search operators are applied to produce new policies. Conversely, the ontogenetic approach to temporal sequence learning performs incremental refinement over a single candidate solution with respect to each state–action pair [1]. The latter is traditionally referred to as reinforcement learning, however, the distinction is often ignored, with reinforcement learning frequently used as a general label for any scenario in which the temporal credit assignment problem/ delayed reward exists; not least because algorithms are beginning to appear which combine both phylogenetic and ontogenetic mechanisms of learning [15].[1] Examples of the temporal sequence learning problem appear in many forms, from control style formulations in which the goal is to learn a policy for controlling a robot or vehicle to games in which the general objective is to learn a strategy. In this work we are interested in the latter domain, specifically the case of learning a strategy to solve multiple configurations of the 3×3 Rubik cube.

The problem of learning to solve Rubik cube configurations presents multiple challenges of wider interest to the temporal sequence learning community. Specific examples might include: 1) a large number of states ranging from trivial to demanding, 2) the problem is known to challenge human players, 3) a wide variation in start states exists, therefore resilient to self play dynamics that might simplify board games such as back-gammon [13], 4) generalization to learn invariances/ symmetries implicit in the game.

Approaches for finding solutions to scrambled configurations of a Rubik cube fall into one of two general approaches: optimal solvers or macro-moves. In the case of solving a cube using a minimal (optimal) number of moves, extensive use is made of lookup tables to provide an exact evaluation function as deployed relative to a game tree summary of the cube state. Thus with respect to the eight corner cubies, the position and orientation of a single cubie is defined by the other 7; or $8! \times 3^7 = 88{,}179{,}840$ combinations. An iterative deepening breadth first search would naturally enumerate all such paths between goal and possible configurations for the corner cubies, forming a “pattern database” for later use. Most emphasis is therefore on the utilization of appropriate hash codings and graph symmetries to extend this enumeration over all possible legal states of a cube (in total there are 4.3252×10^{19} legal states in a 3×3 cube). Such an approach recently identified an upper bound on the number of moves necessary to solve a worst case cube configuration as 26 [7].

Conversely, non-optimal methods rely on ‘macro-moves’ which establish the correct location for specific cubies without disrupting the location of perviously positioned cubies. This is the approach most widely assumed by both human players and ‘automated’ solvers. Such strategies generally take 50 to 100 moves

[1] In the following we will use the terms reinforcement and temporal sequence learning interchangeably, particularly where there is a previous established history of terminology e.g., as in hierarchical reinforcement learning.

to solve a scrambled cube [6]. The advantage this gives is that "general purpose" strategies might result that are appropriate to a wide range of scenarios; thus giving hope for identifying machine learning approaches that generalize. However, from the perspective of cube 'state' we can also see that once one face of a cube is completed the completion of the remaining faces will increasingly result in periods when the relative entropy of the cube will go up considerably.[2] Moreover, from a learning system perspective macro-moves need to be associated with any color combination to be effective, a problem that represents a requirement for learning invariances in a scalable manner.

Two previous published attempts to evolve solutions to the Rubik cube using evolutionary methods have taken rather different approaches to the problem. One attempts to evolve a generic strategy under little a priori information [2]; whereas the second concentrates on independently evolving optimal move sequences to each scrambled cube [3], making use of domain knowledge to formulate appropriate constraints and objectives. In this work we assume the motivation of the former, thus the goal is to evolve a program able to provide solutions to as many scrambled cubes as possible.

The approach taken by [2] employed a domain specific language under the Hayek framework for phylogenetic temporal sequence learning. A domain specific representation included the capability to 'address' specific faces of the cube and compare content with other faces as well as tests for the number of correct cubies. Two approaches to training were considered, either incrementally increasing the difficulty of cube configurations (e.g., one or two twist modifications relative to a solved cube) with binary (solved/ not solved) feedback or cubes with a 100 twist 'scrambling' and feedback proportional to the number of correctly placed cubies. Two different formulations for actions applied to a cube were also considered, with an action space of either three (90 degree turn of the front face, and row or column twists of the cube) or a fixed 3-dimensional co-ordinate frame on which a total of twelve 90 degree turns are applied (the scheme employed here). The most notable result from Hayek relative to this work was that up to 10 cubies could be correctly placed (one face and some of the middle). In effect Hayek was building macro-moves, but could not work through the construction of the remaining cube faces without destroying the work done on the first face.

In the following we develop the Symbiotic Bid-Based (SBB) GP framework and introduce a generic approach to layered learning that does not rely on the a priori definition of different goal functions for each 'layer' (as per the classical definition of Layered learning [14]). The use of layering is supported by the explicitly symbiotic approach adopted to evolution. A discussion of the domain specific requirements will then be made, with results and conclusions establishing the relative success and future work in the Rubik cube domain.

[2]Consider the case of completing the final face if all other cubies are correctly positioned.

2 Layered learning in Symbiotic Bid-Based GP

Symbiosis is a process by which symbionts of different species – in this case computer programs – receive sufficient ecological pressure to cooperate in a common host compartment [5]. Over a period of time the symbionts will either develop the fitness of the host or not, as per natural selection. Thus, fitness evaluation takes place at the level of hosts not at the level of individual symbionts, or a serial dependence between host and symbionts (Figure 1).

In the case of this work, hosts are represented by an independent population – a Genetic Algorithm in this case – each host individual defining a compartment by indexing a subset of individuals from an independent symbiont population (Figure 1). However, rather than symbionts from the same host having their respective outcomes combined in some form of a voting policy – as in ensemble methods – we explicitly require each symbiont to learn the specific context in which they operate. To do so, symbionts assume the bid-based GP framework [8]. Thus, each symbiont consists of a program and a scalar. The program is used to evolve a *bidding strategy* and the scalar expresses a domain dependent *action*, say class label or turn right. The program evolves whereas the action does not. Within the context of a host individual each symbiont executes their program on the current state of the world/ training instance. The symbiont with the largest bid winning the right to present its action as the outcome from that host under the current state. Under a reinforcement learning domain this action would update the state of the world and the process repeats, with a new round of bidding between symbionts from the same host w.r.t. the updated state of the world. Fitness evaluation is performed over the worlds/ training instances defined by the point population (Algorithm 1). Competitive coevolution therefore facilitates the development of point and host populations, with co-operative coevolution developing the interaction between symbionts within a host. Competitive coevolution again appears between hosts in the host population (speciation) to maintain host diversity. This latter point is deemed particularly important in supporting 'intrinsic motivation' in the behaviours evolved,[3] where this represents a central tenet for hierarchical reinforcement learning in general [12].

The above Symbiotic Bid-Based GP or 'SBB' framework – as summarized by Figure 1 and Algorithm 1 – provides a natural scheme for layered learning by letting the content of the (converged) host population represent the actions for a new set of symbionts in a second application of the SBB algorithm; hereafter 'Layered SBB'. The association between the next population of symbionts and the earlier population of hosts is explicitly hierarchical. However, there is no explicit requirement to re-craft fitness functions at each layering (although this is also possible). Instead, the reapplication of the SBB algorithm results in a

[3] Intrinsic motivations or goals are considered to be those central to supporting the existence of an organism. In addition to behaviour diversity, the desire to reproduce is considered an intrinsic motivation/ goal. Conversely, 'extrinsic motivations' are secondary factors that might act in support of the original intrinsic factors such as food seeking behaviours, where these are learnt during the lifetime of the organism and might be specific to that particular organism.

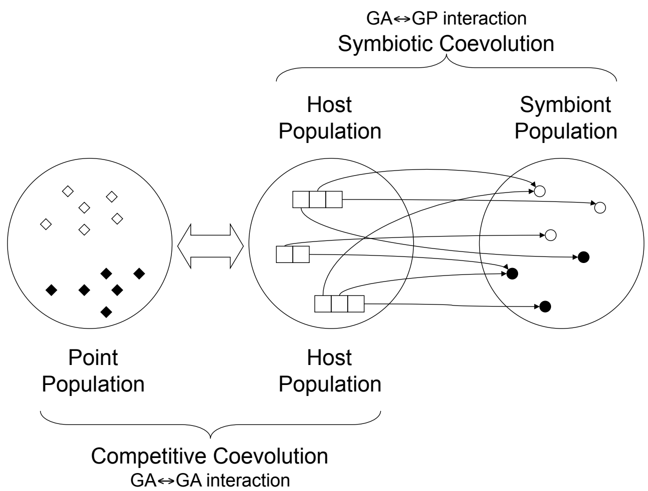

Figure 1: Generic architecture of Symbiotic Bid-Based GP (SBB). A *point population* represents the subset of training scenarios over which a training epoch is performed. The *host population* conducts a combinatorial search for the best symbiont partnerships; whereas the *symbiont population* contains the bid-based GP individuals who attempt to learn a good context for their corresponding actions.

second layer of hosts that learn how to combine previously learnt behaviours in specific contexts. The insight behind this is that SBB bidding policies under a temporal sequence learning domain are effectively evolving the conditions under which an action begins and ends its deployment. This is the general goal of hierarchical reinforcement learning. However, the SBB framework achieves this without also requiring an a priori formulation of the appropriate subtasks, the relation between subtasks, or a modified credit assignment policy; as is generally the case under hierarchical reinforcement learning [12]. In the following we summarize the core SBB algorithm, where this extends the original SBB framework presented in [9] and was applied elsewhere in a single layer supervised learning context [10]; the reader is referred to the latter for additional details of regarding host–symbiont variation operators.

2.1 Point Population

As indicated in the above generic algorithm description, a competitive coevolutionary relationship is assumed between point and host population (Figure 1). Specifically, variation in the point population supports the necessary development in the host population. This implies that points have a fitness and are subject to variation operators. Thus, points are created in two phases on account of assuming a breeder style of replacement in which the worst P_{gap} points are removed (Step 13) – hereafter all references to specific 'Steps' are

Algorithm 1 The core SBB training algorithm. P^t, H^t, and S^t refer to the point, host, and symbiont populations at time t.

```
1:  procedure Train
2:      t = 0                                              ▷ Initialization
3:      initialize point population P^t
4:      initialize host population H^t (and symbiont population S^t)
5:      while t ≤ t_max do                                 ▷ Main loop
6:          create new points and add to P^t
7:          create new hosts and add to H^t (add new symbionts to S^t)
8:          for all h_i ∈ H^t do
9:              for all p_k ∈ P^t do
10:                 evaluate h_i on p_k
11:             end for
12:         end for
13:         remove points from P^t
14:         remove hosts from H^t (remove symbionts from S^t)
15:         t = t + 1
16:     end while
17: end procedure
```

w.r.t. Algorithm 1 – and a corresponding number of new points are introduced (Step 6) at each generation. New points are created under one of two paths. Either a point is created as per the routine utilized at initialization (no concept of a parent point) or offspring are initialized relative to a parent point, with the parent selected under fitness proportional selection. The relative frequency of each point creation scheme is defined by a corresponding probability, p_{genp}. Discussion of the point population variation operators is necessarily application dependent, and is therefore presented later (Section 3).

The evaluation function of Step 10 assumes the application of a domain specific reward that is a function of the interaction between point (p_k) and host (h_i) individuals, or $G(h_i, p_k)$. This is therefore defined later (Equation (6), Section 3) as a weighted distance relative to the ideal target state.

The global / base point fitness, f_k, may now be defined relative to the count of hosts, c_k, within a neighbourhood [10], or

$$f_k = \begin{cases} 1 + \frac{1-c_k}{H_{size}} & \text{if } c_k > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where H_{size} is the host population size, and count c_k is relative to the arithmetic mean μ_k of outcomes on point p_k or,

$$\mu_k = \frac{\sum_{h_i} G(h_i, p_k)}{H_{size}} \tag{2}$$

where $\mu \to 0$ implies that hosts are failing on point p_k and c_k is set to zero. Otherwise, c_k is defined by the number of hosts satisfying $G(h_i, p_k) \geq \mu_k$; that is the number of hosts with an outcome reaching the mean performance on point p_k.

Equation (1) establishes the global fitness of a point. However, unlike classification problem domains, points frequently have context under reinforcement learning domains i.e., a geometric interpretation. This enables us to define a local factor by which the global reward is modulated in proportion to the relative 'local' uniqueness of the candidate point. Specifically, each point is rewarded in proportion to the distance from the point to a subset of its nearest neighbours using ideas from outlier detection [4]. To do so, all the points are first normalized by the maximum pair-wise Euclidean distance – as estimated across the point population content, therefore limiting local reward to the unit interval – after which the following reward scheme is adopted:

1. The set of K points nearest to p_k is identified;
2. The local reward r_k is calculated as,

$$r_k = \left(\frac{\sum_{p_l} (D(p_k, p_l))^2}{K} \right)^2 \tag{3}$$

   where the summation is taken over the set of K points nearest to p_k and $D(\cdot, \cdot)$ is the application specific distance function (Equation (7), Section 3).
3. The corresponding final fitness for point p_k is defined in terms of both global and local rewards or

$$f'_k = f_k \cdot r_k \tag{4}$$

With the normalized fitness f'_k established we can now delete the worst performing P_{gap} points (Step 13).

2.2 Host and Symbiont Population

Hosts are also subject to the removal and addition of a fixed number of H_{gap} individuals per generation, Steps 14 and 7 respectively. However, in order to also promote diversity in the host population behaviours, we assume a fitness sharing formulation. Thus, shared fitness, s_i of host h_i has the form,

$$s_i = \sum_{p_k} \left(\frac{G(h_i, p_k)}{\sum_{h_j} G(h_j, p_k)} \right)^3 \tag{5}$$

Thus, for point p_k the shared fitness score s_i re-weights the reward that host h_i receives on p_k relative to the reward on the same point as received by all hosts. As per the earlier comments regarding the role of fitness sharing in supporting 'intrinsic motivation,' a strong bias for diversity is provided through the cubic power. Evaluation takes place at Step 10, thus all hosts, h_i, are evaluated on all points, p_k.

Once the shared score for each host is calculated, the H_{gap} lowest ranked hosts are removed. Any symbionts that are no longer indexed by hosts are considered ineffective and are therefore also deleted. Thus, the symbiont population size may dynamically vary, with variation operators having the capacity to add additional symbionts [10], whereas the point and host populations are of a fixed size.

3 Domain specific design decisions

3.1 Cube representation and actions

The representation assumed directly indexes all 54 facelets comprising the 3×3 Rubik cube. Indexing is sequential, beginning at the centre face with cubie colours differentiated in terms of integers over the interval $[0, ..., 5]$. Such a scheme is simplistic with no explicit support for indicating which facelets are explicitly connected to make corner or edges. Actions in layer 0 define a 90 degree clock-wise or counter clock-wise twists to each face; there are 6 faces resulting in a total of 12 actions. When additional layers are added under SBB, the population of host behaviours from the previous population represent the set of candidate actions. As such additional layers attempt to evolve new contexts for previously evolved behaviours/ build larger macro-moves.

3.2 Reward and distance functions

The reward function applies a simple weighting scheme to the number of quarter turn twists (i.e., actions) necessary to move the final cube state to a solved cube. Naturally, such a test becomes increasingly expensive as the number of moves applied in the 'search' about the final cube state increases. Hence, the search is limited to testing for up to 2 moves away from the solution, resulting in the following reward function,

$$G(h_i, p_k) = \frac{1}{(1 + D(s_f, s^*))^2} \tag{6}$$

where s_f is the final state of the cube relative to cube configuration p_k and sequence of moves defined by host h_i; s^* is the ideal solved cube configuration, and; $D(s_2, s_1)$ defines the weighted distance function, or

$$D(s_2, s_1) = \begin{cases} 0, & \text{when 0 quarter twists match state } s_2 \text{ with } s_1 \\ 1, & \text{when 1 quarter twists match state } s_2 \text{ with } s_1 \\ 4, & \text{when 2 quarter twists match state } s_2 \text{ with } s_1 \\ 16, & \text{when} > 2 \text{ quarter twists match state } s_2 \text{ with } s_1 \end{cases} \tag{7}$$

Naturally, curtailing the 'look-ahead' to 2 quarter turn twists from the presented solution casts the fitness function into that of a highly deceptive 'needle in a haystack' style reward i.e., feedback is only available when you have all but provided a perfect solution. Adding additional twist tests however would result in tens of thousands of cube combinations potentially requiring evaluation before fitness could be defined. Other functions such as counting the number of correct facelets or cube entropy generally appeared to be less informative. The utility of combined metrics or a priori defined constraints might be of interest in future work.

3.3 Symbiont representation

Symbionts take the form of a linear GP representation, with instruction set for the Bid-Based GP individuals consisting of the following generic set of operators $\{+, -, \times, \div, ln(\cdot), cos(\cdot), exp(\cdot), if\}$. The conditional operator '*if*' applies an inequality operator to two registers and interchanges the sign of the first register if its value is smaller than the second. There are always 8 registers and a maximum of 24 instructions per symbiont.

3.4 Point initialization and offspring

Initialization of points – cube configurations used during evolution (Step 3) – takes the form of: (1) uniform sampling from the interval $[1, ..., 10]$ to define the number of twists applied to a solved cube; (2) stochastic selection of the sequence of quarter twist actions used to 'scramble' the cube, and; (3) test for a return to the solved cube configuration (in which case the quarter twist step is repeated). Thereafter, new points introduced during breeding (Step 6) follow one of two scenarios: adding twists to a parent point to create a child with probability p_{genp} or create a new point as per the aforementioned point initialization algorithm with probability $1 - p_{genp}$. The point offspring/ parent-wise creation is governed by the following process:

1. Select parent point, $p_i \in P^t$, under fitness proportional selection (point fitness defined by Equation (4), Section 2);

2. Define the number of additional twists, w_i, applied to create the child from the parent in terms of a normal p.d.f., or

$$w_i = abs(N(0, \sigma_{genTwist})) + 1 \tag{8}$$

   where $N(0, \sigma_{genTwist})$ is a normal p.d.f. with zero mean and variance $\sigma_{genTwist}$. Naturally, this is rounded to the nearest integer value;

3. Until the twist limit (w_i) is reached, select faces and clockwise/ counter clockwise twists with uniform probability relative to the parent cube configuration, p_i;

4. Should the resulting cube be a solved cube, the previous step is repeated.

Table 1: Parameterization at Host (GA) and Symbiont (GP) populations. As per Linear GP, a fixed number of general purpose registers are assumed ($numRegisters$) and variable length programs subject to a max. instruction count ($maxProgSize$).

Host (solution) level			
Parameter	Value	Parameter	Value
t_{max}	1 000	ω	24
P_{size}, H_{size}	120	P_{gap}, H_{gap}	20, 60
p_{md}	0.7	p_{ma}	0.7
p_{mm}	0.2	p_{mn}	0.1
Symbiont (program) level			
numRegisters	8	*maxProgSize*	24
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

4 Results

4.1 Parameterization

Runs are performed over 60 initializations for both the case of Layered SBB (two layers) and single layer SBB base cases. The latter are parameterized to provide the same number of fitness evaluations/ upper bound on the number of instructions executed as per the total Layered SBB requirement. In the case of this work this implies a limit of 72000 evaluations or a $maxProgSize$ limit of 36 under the single layer baseline; hereafter '*big prog*'. Likewise reasoning brings about a team size limit (ω) of 36 under the single layer SBB baseline; hereafter '*big team*'. Relative to the sister work in which the current SBB formulation was applied to data sets from the supervised learning domain of classification [10], three additional parameters are introduced for point generation (Section 2): (1) outlier parameter $K = 13$; (2) the probability of creating points $p_{genp} = 0.9$; and, (3) the variance for defining the number of additional twists necessary to create an offspring from a parent point $\sigma_{genTwist} = 3$. All other parameters are unchanged relative to those of the classification study (Table 1).

4.2 Sampled Test Set

Post training test performance is evaluated w.r.t. 5,000 unique 'random' test cubes, created as per the point initialization algorithm. Table 2 summarizes the distribution of cubes relative to the number of twists used to create them. A combined violin / quartile box plot is then used to express the total number of cube configurations solved. Figures 2 and 3 summarize this in terms of a single champion individual from each run[4] and corresponding cumulated population wide performance.

[4]Identified post training on an independent validation set generated as per the stochastic process used to identify the independent test set.

Table 2: Distribution of test cases. Samples selected over 1 to 10 random twists relative to solved cube resulting in 5,000 unique test configurations.

Number of twists	# of test cases	Number of twists	# of test cases
1	9	6	662
2	86	7	640
3	403	8	728
4	527	9	673
5	588	10	683

It is immediately apparent that the population wide behaviour (Figure 3) provides a significant source of useful diversity relative to that of the corresponding individual-wise performance (Figure 3). This is a generic property of fitness sharing implicit in the base SBB algorithm; Equation (5). However, it is also clear that under SBB 1 – in which second layer symbionts assume the hosts from layer 0 as their actions – the champion individuals are *unable* to directly build on the cumulative population wide behaviour from SBB 0. Conversely, under the case of real-valued reinforcement problem domains – such as the truck backer-upper [11] – SBB 1 individuals were capable of producing champions that subsumed the SBB 0 population-wise performance. We attribute this to the more informative fitness function available under the truck backer-upper domain than that available under the Rubik cube.

Relative to the non-layered SBB base cases, no real trend appears under the individual-wise performance (Figure 2). Conversely, under the cumulative population wide behaviour (Figure 3), SBB 1 provides a significant improvement as measured in terms of a two-tail Mann-Whitney test with 0.01 significance level (Table 3); effectively identifying the most consistently effective solutions. This appears to indicate that Layered SBB is able to build configuration specific sub-sets of Rubik cube solvers – that is to say, the strategies for solving cube configurations are not colour invariant. Specifically, the macro moves learnt at SBB 0 cannot be generalized over all permutations of cube faces. Thus, at SBB 1, subsets of hosts from SBB 0 can be usefully combined. However, this only results in the median performance improving by approximately 50 (200) test cases between layers 0 and 1 under single champion (respectively population-wise) test counts. Overall, neither increasing the instruction count limit per symbiont or maximum limit on the number of symbionts per host is as effective as layering at leveraging the performance from individual-wise to population wide performance.

4.3 Exhaustive test set

A second test set is designed consisting of all 1, 2 and 3 quarter twist cube configurations – consisting of 12, 114 and 1,068 unique test cubes respectively.[5]

[5] These counts are somewhat lower than those reported in [6] because we do not include 180° twists in the set of permitted actions.

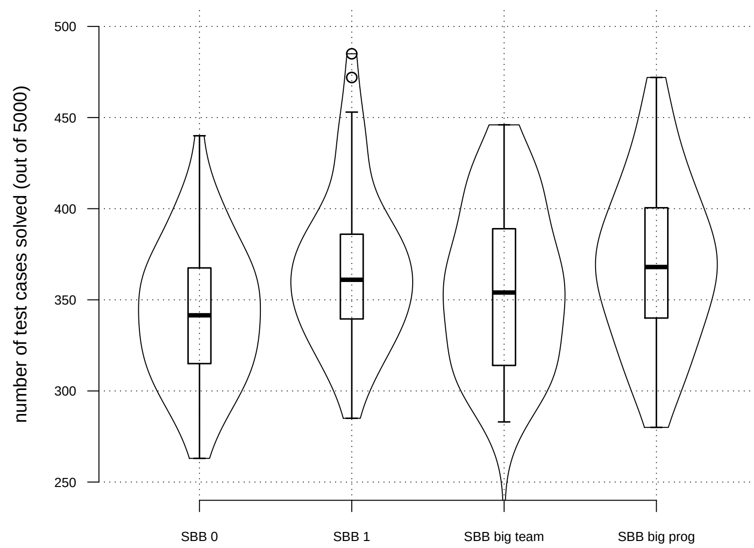

Figure 2: Total test cases solved by single best individual per run under SBB with and without layering under the stochastic sampling of 5,000 1 to 10 twist cubes. '*SBB 0*' and '*SBB 1*' denote first and second layer Layered SBB solutions. '*big team*' and '*big prog*' represent single layer SBB runs with either larger host or symbiont instruction limits.

Table 3: Two-tailed Mann-Whitney test comparing total solutions under the Sampled Test Set provided by Layered SBB (second level) against single layer SBB parameterizations (big team (SBB-bt) and big program (SBB-bt)). The table reports p-values for the pair-wise comparison of distributions from Figures 2 and 3. Cases where the Layered SBB medians are higher (better) than non-layered SBB medians are noted with a $\star$.

Test Case	Champion individual	Population wide
Layered SBB vs SBB-0	0.002499$\star$	3.11e-15$\star$
Layered SBB vs SBB-bt	0.1519$\star$	1.003e-10$\star$
Layered SBB vs SBB-bp	0.5566	0.0002617$\star$

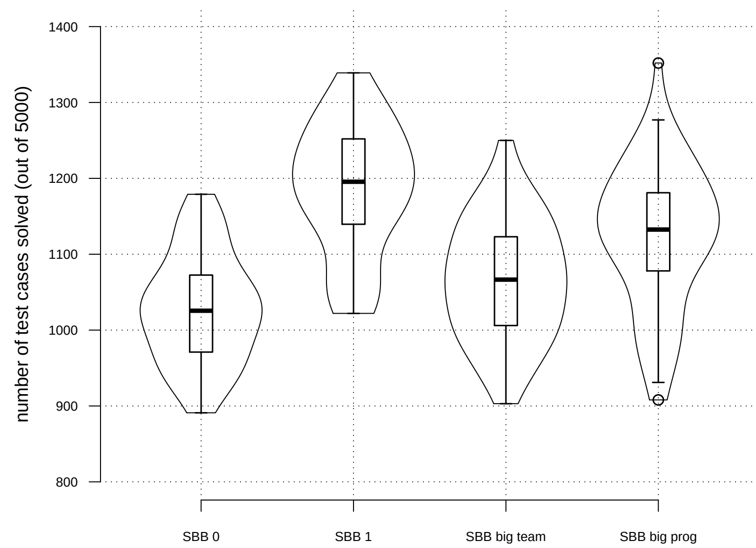

Figure 3: Total test cases solved by cumulated population wide performance per run under SBB with and without layering under the stochastic sampling of 5,000 1 to 10 twist cubes. '*SBB 0*' and '*SBB 1*' denote first and second layer Layered SBB solutions. '*big team*' and '*big prog*' represent single layer SBB runs with either larger host or symbiont instruction limits.

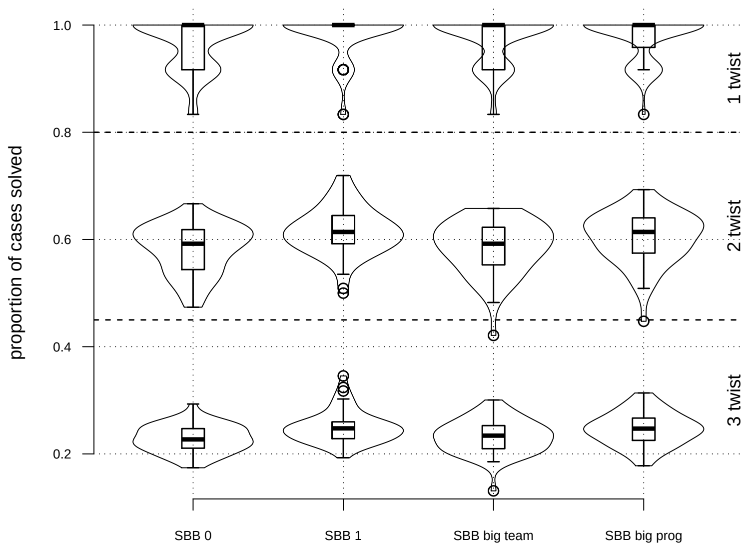

Figure 4: Percent of cases solved by single best SBB individuals as estimated under the exhaustive enumeration of 1, 2 and 3 quarter twist test cases. SBB 0 and SBB 1 denote the first and second layer solutions under Layered SBB; 'big team' and 'big prog' denote the base case SBB configurations without layering.

Naturally, there is no a priori bias towards solving these during training; cubes being configured stochastically relative to points selected under fitness proportional selection. Figure 4 summarizes this as a percentage of the number of 1, 2 and 3 twist configurations solved by the single best individual in each run.[6] The impact of layering is again evident, both from a consistency perspective and in terms of incremental improvements to the number of cases solved with each additional layer. Relative to the baseline single layer models, it is interesting to note that both 'SBB big team' and 'SBB big prog' had difficulty consistently solving the 1 twist configurations, whereas all SBB 1 first quartile performance corresponds to all test cases solved. Of the two baseline configurations, 'SBB big prog' was again the more effective, implying that more complexity in the symbionts was more advantageous than larger host–symbiont capacity.

Finally, we can also review the (mean) number of twists used to provide solutions to each test configuration (Figure 5). The resulting distributions are grouped by the original twist count. The move counts are averaged over all cases solved by an individual, thus although some, say, 1 twist test cases might be solved in one twist, cases that used three moves would naturally increase the average move count above the ideal. Application of a two-tailed Man-Whitney test indicates that the 'SBB 1' move counts are lower than the 'SBB-bp' ('big

[6] The same 'champion' individual as identified under the aforementioned validation sample a priori to application of the sampled test set.

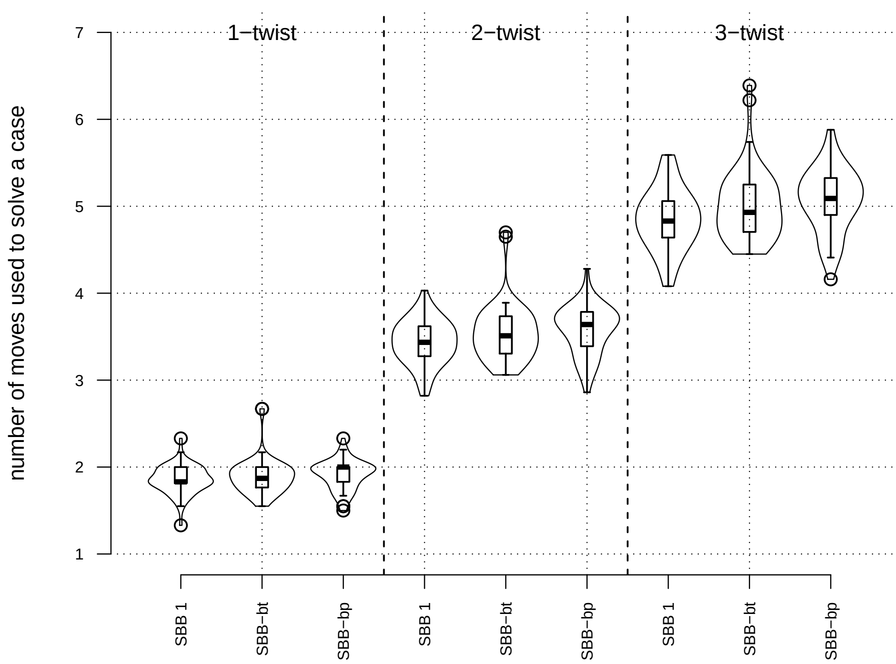

Figure 5: Number of moves used by champion individual to solve 1-, 2- and 3-twist points. '*SBB 1*' is the second layer from Layered SBB, '*SBB-bt*' and '*SBB-bp*' denote the corresponding single layer SBB big team and big program parameterizations.

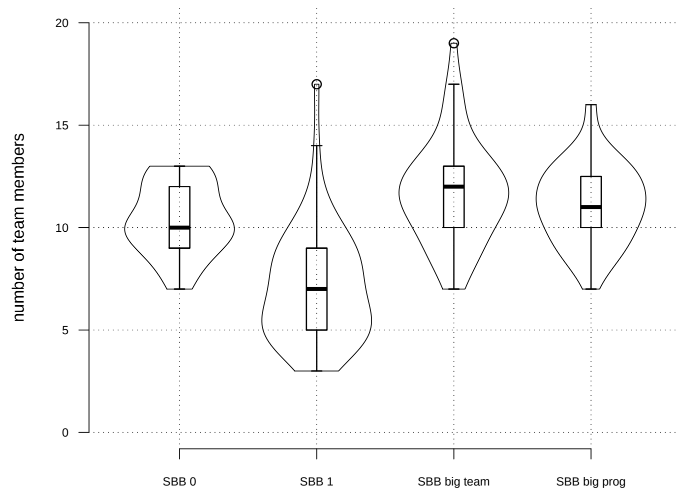

Figure 6: Number of symbionts per host over SBB runs.

Table 4: Two-tailed Mann-Whitney test results comparing solution move counts for champion individuals with Layered SBB (second level) against single layer SBB parameterizations (big team (SBB-bt) and big program (SBB-bt)). The table reports p-values for the pair-wise comparison of distributions from Figure 7. Cases where the single layer SBB medians are higher (worse) than Layered SBB medians are noted with a $\star$.

Test Case	1-twist	2-twist	3-twist
Layered SBB vs SBB-bt	0.4976$\star$	0.1374$\star$	0.0534$\star$
Layered SBB vs SBB-bp	0.02737$\star$	0.001951$\star$	0.0007957$\star$

program') move counts on 2- and 3-twist test cases at a 0.01 significance level (Table 4). Thus, although Layered SBB and SBB big program solved a similar total number of test cases (Figure 4), Layered SBB is able to solve them using a statistically significant lower number of moves. Conversely, SBB big team was not able to solve as many test cases, but when it did provide solutions, a similar number of moves as Layered SBB where used.

4.4 Model complexity

Finally, we can also consider model complexity, post intron removal. Relative to the typical number of symbionts utilized per host (Figure 6), layer 0 clearly utilizes more symbionts per host than layer 1. This implies that at layer 1 there are 5 to 8 hosts from layer 0 being utilized. As indicated in Section 2, this is possible because each of the hosts from layer 0 is now associated with a symbiont bidding behaviour as evolved at level 1. Further analysis will be

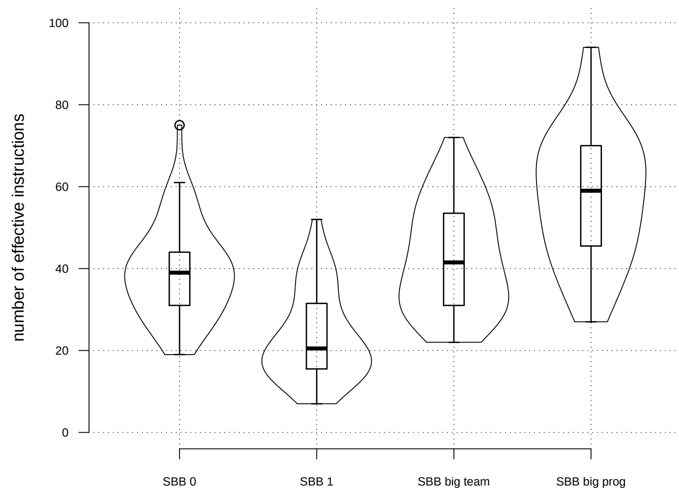

Figure 7: Number of instructions per host over SBB runs.

necessary to identify what the specific patterns of behaviour associated with these combinations of hosts represent. Both base cases appear to use more symbionts per host, understandable given that they do not have the capacity to make use of additional layers. The same bias towards simplicity again appears relative to instruction count (Figure 7), thus SBB 1 uses a significantly lower instruction count than SBB 0 and the 'SBB big prog' naturally results in the most complex symbiont programs. Needless to say, SBB 1 solutions will use some combination of SBB 0 solutions, however, relative to any one move, only two hosts are ever involved in defining each action.

5 Conclusions

Temporal sequence learning represents the most challenging scenario for establishing effective mechanisms for credit assignment. Indeed, specific challenges under the temporal credit assignment problem are generally a superset of those experienced under supervised learning domains. Layered learning represents one potential way of extending the utility of machine learning algorithms in general to temporal sequence learning [14]. However, in order to do so effectively, solutions from any one 'layer' need to be both diverse and self-contained; properties that evolutionary computation may naturally support. Moreover, when building a new layer of candidate solutions the problem of automatic context association must be explicitly addressed. The SBB algorithm provides explicit support for these features and thus is able to construct layered solutions without recourse to hand designed objectives for each candidate component contributing to a solution [11]. This is in marked contrast to the original Layered learning methodology or the more recent developments in hierarchical reinforcement learning [14].

The Rubik cube as a whole is certainly not a 'solved' problem from a learning algorithm perspective. The current state-of-the-art evolves solutions for each cube configuration [3], or as in the work reported here, provides a general strategy for solving a subset of scrambled cubes [2]. The discrete nature of the Rubik problem domain makes the design of suitable fitness and distance functions less intuitive/ more challenging than in the case of continuous valued domains. Indeed, specific examples of the effectiveness of SBB style layered learning under continuous valued reinforcement learning tasks are beginning to appear [11]. It is therefore anticipated that future developments will need to make use of more structural adaptation to the point population and/ or make use of a priori constraints in the formulation of different fitness functions per layer, as in the case of more classical approaches to building Rubik cube 'solvers'.

Acknowledgments

Peter Lichodzijewski has been a recipient of Precarn, NSERC-PGSD and a Killam Postgraduate Scholarships. Malcolm Heywood holds research grants from NSERC, MITACS, CFI, SwissCom Innovations SA. and TARA Inc.

References

[1] A. M. S. Barreto, D. A. Augusto, and H. J. C. Barbosa. On the characteristics of sequential decision problems and their impact on Evolutionary Computation and Reinforcement learning. In *Proceedings of the International Conference on Artificial Evolution*, page in press, 2009.

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

[3] N. El-Sourani, S. Hauke, and M. Borschbach. An evolutionary approach for solving the Rubik's cube incorporating exact methods. In *EvoApplications Part – 1: EvoGames*, volume 6024 of *LNCS*, pages 80–89, 2010.

[4] S. Harmeling, G. Dornhge, F. Tax, F. Meinecke, and K. R. Muller. From outliers to prototypes: Ordering data. *Neurocomputing*, 69(13-15):1608–1618, 2006.

[5] M. I. Heywood and P. Lichodzijewski. Symbiogenesis as a mechanism for building complex adaptive systems: A review. In *EvoApplications: Part 1 (EvoComplex)*, volume 6024 of *LNCS*, pages 51–60, 2010.

[6] 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.

[7] D. Kunkle and G. Cooperman. Twenty-six moves suffice for rubik's cube. In *Proceedings of ACM International Symposium on Symbolic and Algebraic Computation*, pages 235–242, 2007.

[8] P. Lichodzijewski and M. I. Heywood. Pareto-coevolutionary Genetic Programming for problem decomposition in multi-class classification. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 464–471, 2007.

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

[10] P. Lichodzijewski and M. I. Heywood. Symbiosis, complexification and simplicity under gp. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2010. To appear.

[11] P. Lichodzijewski and M.I. Heywood. A symbiotic coevolutionary framework for layered learning. In *AAAI Symposium on Complex Adaptive Systems*, 2010. Under review.

[12] P. Y. Oudeyer, F. Kaplan, and V. V. Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286, 2007.

[13] J. B. Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32:225–240, 1998.

[14] P. Stone. Learning and multiagent reasoning for autonomous agents. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 13–30, 2007.

[15] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:887–917, 2006.