On Run Time Libraries and Hierarchical Symbiosis

Stephen Kelly[1], Peter Lichodzijewski[1], and Malcolm I. Heywood[1]

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

Abstract

Run time libraries (RTL) in genetic programming (GP) represent a scenario in which individuals evolved under an earlier independent evolutionary run can be potentially incorporated into a following GP run. To date, schemes for exploiting the RTL metaphor have emphasized syntactic over behavioural approaches. Thus, instructions are added to the later run such that the previous code can be explicitly indexed. In this work we demonstrate how the RTL concept is naturally supported by adopting a symbiotic framework for coevolution. The approach is demonstrated under the Pinball reinforcement learning task. We demonstrate how the initial RTL can be coevolved under a simpler formulation of the task and then use this as the basis for providing solutions to a more difficult target task under the same domain. The resulting solutions are stronger than an RTL as coevolved against the target task alone or symbiosis as evolved without support for RTL.

1 Introduction

Classically, genetic programming (GP) is deployed on a target task with respect to a specific parameterization, typically over multiple initializations; hereafter a *run*. Should solutions not be forthcoming, then futher run(s) are repeated under a new parameterization e.g., larger population and / or more generations. Conversely, it might be expected that GP individuals from one run, although not solving the task outright, represent candidates for in some way informing or biasing the next attempt; as in code reuse. Broadly speaking, run time libraries (RTL) represent a scenario in which individuals from an earlier run are reused by a later independent run. This is somewhat different from the related goal of evolving modules within a run, an approach that does not make use of genetic 'material' from outside the current run e.g., Automatically Defined Functions [8].

In this work we are interested in pursuing a coevolutionary approach to RTL. To date, most approaches to coevolving solutions in GP assume a teaming metaphor. Thus, a team of programs are coevolved in parallel from a single run of a coevolutionary process. Generally, one of two schemes are pursued: Pareto based ensemble generation (e.g., [15]) or group fitness frameworks (e.g., [2, 19, 21]). To date such schemes have not directly considered the potential for making use of code previously evolved as in an RTL context. Moreover, solutions from such coevolutionary schemes typically assume that programs are deployed in parallel, with either a majority or maximum vote defining the outcome. One disadvantage from an RTL context is the lack of support for explicitly hierarchical relationships i.e., the ability to 'call' previously evolved programs. The approach typically assumed for addressing this is to add instructions to permit subroutine calls. Thus, it is not coevolution that provides support for RTL but an extended instruction set. Moreover, it has also been recognized that biases are often necessary in order to force the new code to reference the library e.g., [16, 10, 18]; where such biases might have an unforeseen impact on the resulting solutions.

The approach taken in this research builds on the recently proposed Symbiotic Bid-based (SBB) framework for coevolving teams of programs [13]. One of the key concepts of this framework is that programs (symbionts) adopt a bid-based model of GP [12], thus an explicit separation of action and context. Actions are merely scalars representing a task specific atomic action – as in a *discrete* set of legitimate actions for a robot – where each symbiont can only have a single action. The context is provided by executing the symbiont's program and is established relative to a team of programs defined by a member from an independent host population. Thus each host defines a team of symbionts and the action of a host is established by executing all the symbionts identified by the host. The symbiont with largest program output is declared the 'winner' and presents its action.

From the perspective of RTL, conducting the first run of SBB results in a set of candidate *meta actions* i.e., the content of the host–symbiont populations at the end of the run. Although these hosts may not represent complete solutions on their own they could, given the relevant context, represent meta actions for constructing stronger policies in a *following* evolutionary run. This also opens the opportunity for meta action discovery under different fitness formulations for the original and following SBB runs i.e., RTL is evolved under one objective but deployed under a similar but different objective.

This work illustrates the utility of SBB to automatically discover a population of meta actions and then reuse them to solve a larger task under the 'Pinball' reinforcement learning (RL) task; a task recently proposed for demonstrating the utility of a hierarchical RL algorithm [6]. Training scenarios are defined under a tabula rasa sampling of task configurations by a point population. Moreover, we demonstrate how the point population itself may be used to define functionally different goals during the first run of SBB, thus providing more a effective RTL for reuse at the subsequent SBB run.

2 Related Research

From the perspective of previous research on run time libraries in GP, the subtree encapsulation scheme collected statistics about subtree use following completion of a run [16]. Previously evolved subtrees then appeared as terminals in a new independent run. Experiments with frequency based subtree inclusion versus uniform selection of subtrees for inclusion indicated that uniform selection performed better on the tasks considered. Conversely, layering metaphors similar to that of stacked generalization have been proposed [14]. A run is performed and the best individual selected. The output of this individual augments the task attributes, and a new run performed. The process has some sensitivity to degenerate individuals that need penalizing, but is capable of providing very effective cascades of classifiers for supervised learning contexts. The Mnemosyne framework builds a library of different arity subtrees and assigns each a real-valued number or 'tag' [5]; thus defining the RTL. The tag provides the mechanism for linking library content to a calling program. In short, the calling program evolves a tree containing function nodes that declare tag values and leaf nodes. There is therefore no code within the calling program, just tag references to subtrees in the library, and domain specific leaf nodes. The purpose of evolution is therefore to establish the relevant tags in the population of calling programs. Tags from the calling program are matched with those from sub-trees in the RTL using a distance function. This scheme was recently generalized to the case of coevolving code modules and calling program, although in this case, both code modules and the calling code were coevolved together [18]. Extension to the RTL context would be straightforward enough.

3 Symbiotic Bid-Based Policy Search

The generic architecture for SBB explicitly enforces symbiosis by separating host and symbiont into independent populations, Figure 1. Each host represents a candidate solution in the form of a set of symbiont programs existing independently in the symbiont population. Performance is measured relative to the interaction between a subset of initializations from the task domain (points) and host, whereas symbionts 'die' when they no longer receive any host indexes. A breeder model of evolution is assumed, thus a fixed number of hosts and points are deleted/ introduced at each generation. The respective properties of symbiont and host population are developed below (for a detailed presentation see [13]). Explicit support for constructing programs with a hierarchical decomposition, thus evolution of a RTL, is presented in Section 3.4.

3.1 Representation and execution

3.1.1 Symbiont Population

Members of the symbiont population assume a Bid-Based GP representation [11]. As such, each symbiont, sym, is represented as a tuple $\langle a, p \rangle$; where a is an action as selected from the set of atomic actions associated with the task domain and p is the corresponding symbiont's program. A program defines a bidding behaviour, thus context for deploying its action. Without loss of generality, a linear representation is assumed [2]. Execution of a symbiont's program results in a corresponding real-valued outcome in the output register, $R[0]$. In order to encourage a common bidding range from the outset, this is mapped to the unit interval through the sigmoid operator, or $sym(bid) = (1 + \exp(-R[0]))^{-1}$. The linear representation leads to programs being defined by a simple register addressing language of the form: 1) Two argument instructions, or $R[x] \leftarrow R[x] < op_2 > R[y]; op_2 \in \{+, -, \div, \times\}$; 2) Single argument instructions, or $R[x] \leftarrow < op_1 > (R[y]); op_1 \in \{\cos, \ln, \exp\}$; 3) A conditional statement of the form "IF $(R[x] < R[y])$ THEN $(R[x] \leftarrow -R[x])$. In addition, $R[y]$ can be either a register reference or index a state variable.

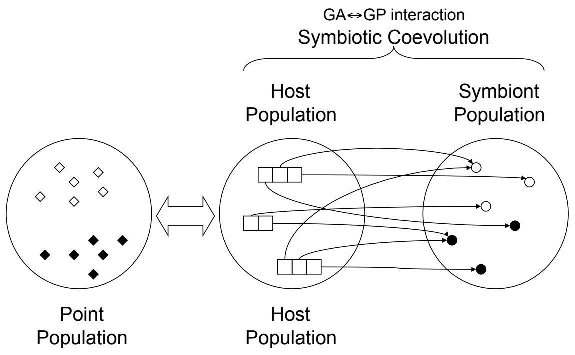

Figure 1: Generic architecture of Symbiotic Bid-Based GP (SBB). A *point population* represents the subset of training scenarios (sampled from the task domain) over which a training epoch is performed. The *host population* conducts a combinatorial search for the best symbiont partnerships; whereas the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

3.1.2 Host Population

Symbionts are explicitly limited to deploying a single action. Thus a host needs to identify relevant subsets of symbionts that are capable of collaborating. To do so, each host indexes a subset $[2, ..., \omega]$ of the symbionts currently existing in the symbiont population. Relative to a single host, h_i, fitness evaluation is conducted against a set of initial configurations of the task domain, as defined by individuals from the point population, p_j. Such a process has the following form: 1) Present the state variables for the current time step t_s of the task domain or $\vec{s}(t_s)$; 2) Execute all (symbiont) programs identified by host, h_i, resulting in a matching number of symbiont bids; 3) Identify the 'winning' symbiont as that with the maximum bid from host h_i or $sym^* = \arg_{sym \in h_i} \max(sym(bid(\vec{s}(t_s))))$; 4) Present the action from the winning symbiont to the task domain and update the state variables accordingly. Symbionts therefore use bidding to establish the context for deploying their respective action. The number of symbionts per host and 'mix' of actions represented by a host are both an artifact of the evolutionary cycle. Task performance across the host population will be discounted under competitive fitness sharing [17] to provide the overall measure of a host's performance once all task scenarios defined by the point population have been assessed, Section 3.2.2.

3.1.3 Point Population

The role of the point poplation is to sample initial conditions from the task with sufficient diversity to provide variation in the behaviours of hosts as measured through the fitness function. This has implications for both maintaining engagement during training [3] and post training generalization [9]. A tabula rasa approach is assumed, where this is taken to imply minimal information regarding the sampling of an initial task configuration, $s(t = 0)$. Moreover, from the perspective of episodic tasks in particular, the (global) goal condition remains fixed in all cases: there is no 'shaping' of the reward function [1]. Note that we differentiate between 'goal condition' and 'terminal condition,' with terminal conditions generally subsuming goal conditions.

For example, under maze style domains there is generally a single entrance and exit location. Thus, the overall goal might be to successfully navigate between the two; whereas the terminal condition expresses the computational 'budget' within which the goal condition should be encountered. Solving this 'ultimate configuration' directly under an evolutionary policy search might not be feasible i.e., the disengagement problem [3].

In summary, while remaining true to the tabula rasa creation of content for the point population, we let individuals from the point population define training configurations in terms of *both* start states and (sub)goals. Thus, start and (sub)goal states will be defined stochastically with only tests applied to establish their validity relative to constraints of the task domain, see Section 4.2. Our hypothesis is that the hierarchical SBB architecture is in a position to make explicit use of this in terms of first evolving a population of meta actions relative to the sub-goals defined by the point population during the first application of SBB and then establish how to deploy subsets of meta actions relative to a common global goal under a second independent application of SBB (Section 3.4).

3.2 Selection and Replacement

3.2.1 Point Population

At each generation P_{gap} points are removed with uniform probability and a corresponding number of new points introduced. The process for generating points is naturally a function of the task domain itself and will be detailed in Section 4.2 once the 'pinball' maze task has been defined, Section 4.1.

3.2.2 Host Population

As per the point population, a fixed number of hosts, H_{gap} are removed at each generation. Host removal is applied deterministically with the worst H_{gap} hosts targeted for removal at each generation. However, assuming a competitive fitness sharing formulation [17] maintains diversity in the host population. Thus shared fitness, s_i of host h_i takes the form:

$$s_i = \sum_k \left(\frac{G(h_i, p_k)}{\sum_j G(h_j, p_k)} \right)^3 \tag{1}$$

where $G(h_i, p_k)$ is the task dependent reward defining the quality of policy h_i on test point p_k (see Section 4.2).

Naturally, deleting the worst H_{gap} hosts may result in some symbionts no longer receiving indexes. This is taken to imply that such symbionts must have been associated with the worst performing hosts, these symbionts are also deleted. A secondary implication of this is that symbiont population size will vary whereas the host population size remains fixed [13].

3.3 Variation Operators

Symbiosis is an explicitly hierarchical coevolutionary process. From a exploration/ exploitation perspective it is important not to disrupt 'good' symbiont combinations while simultaneously continuing to search for better hosts. Moreover, variation needs to be maintained at the symbiont level without disrupting symbionts that are already effective. The process for maintaining exploration (diversity) without unduly disrupting the better host–symbiont relationships therefore follows an explicitly hierarchical formulation. Following the removal of H_{gap} hosts, the remaining $H_{size} - H_{gap}$ hosts are sampled for cloning with uniform probability. The resulting clones have both their host and symbiont content modified. As such, it is ensured that new symbionts are only associated with new hosts and therefore no disruption of previous host behaviour takes place. For a detailed presentation of this process see [13].

3.4 Hierarchical redeployment of meta actions

The above description summarizes the process as applied to a single 'level' of symbiosis, or Figure 1. Thus, the discrete set of actions, s, initially assumed by symbionts correspond to the set of *atomic actions* specific to the target task domain. However, after evolving for a fixed number of generations, the content of the host–symbiont population pair might not provide any outright solutions. Rather than begin evolution afresh from a completely new host–symbiont parameterization/ initialization, the current host population is considered to represent a run time library (RTL) for constructing more complex policies. To do so, each host *previously evolved* at level 'l' defines the set of meta actions (RTL content) as indexed by a new symbiont population at level '$l + 1$'. Thus the only difference between evolution at each level is the declaration of the actions that symbionts may index. At level $l = 0$ symbiont actions are always defined by the task domain, or 'atomic actions.' Thereafter, for symbionts at level $l > 0$ the set of (meta) actions is defined by the hosts as previously evolved at level $l - 1$ or $a \in \{H(l-1)\}$. For simplicity we assume that evolution is only ever conducted at the highest 'level'; all earlier levels having their content 'frozen'. Moreover, there is no requirement for each host at level l to index all available meta actions from level $l - 1$. The subset of actions utilized by each host is a function of policy search.

Evaluating host i at level l (or h_i^l) now has the following generic form:

1. Present the state variables describing the current state of the task domain, or $\vec{s}(t_s)$;
2. For all symbionts in host h_i^l – that is $\forall sym^l \in h_i^l$ – identify the corresponding level l symbiont bid, or $sym^l(bid(\vec{s}(t_s)))$
3. Identify the 'winning' symbiont for host h_i^l, or $sym^* = \arg_{sym^l \in h_i^l} \max(sym^l(bid(\vec{s}(t_s))))$;

4. IF $l == 0$ THEN Step (5) ELSE
   (a) Update level pointer: $l = l - 1$;
   (b) Update the host under evaluation as indexed by the 'winning' symbiont sym^* from Step 3, or $h_i^l \leftarrow sym^*(a)$
   (c) RETURN to Step (2);
5. Apply the atomic action from the winning symbiont to the task domain and update any state variables accordingly, or $\vec{s}(t_s + 1) \leftarrow \langle \text{task domain} \rangle \leftarrow sym^*(a)$.

Hereafter, this will be referred to as hierarchical SBB.

4 Empirical Evaluation

4.1 Pinball task

The 'pinball' domain was first utilized by [6] in their demonstration of a value function framework for hierarchical reinforcement learning under a continuous state space. The basic goal of the task is to maneuver a ball across a maze like world into a single goal location defined by a hole, Figure 2. What makes the task interesting is that collisions with the (typically non-parallel) walls of the domain are elastic, thus the policy learner may make use of obstacles to change direction of the ball. State variables are derived from a Cartesian coordinate frame, or $\vec{s} = \{x, y, \dot{x}, \dot{y}\}$. The ball is manipulated directly by the policy learner through the application of one of five atomic actions, or $a \in \{\pm\dot{x}, \pm\dot{y}, 0\}$ where zero implies no change and the other actions imply that the corresponding velocity is incremented/ decremented.

4.2 Task Specific Parameterization

As introduced in Section 3.1.3, the point population defines properties of the environment against which evolution takes place, thus providing the basis for a simple ecosystem. However, we are also interested in having the environment specify *both* start states and (sub)goals for each training episode. The motivation being that constructing useful meta actions need not be directly related with solving the final goal state.[1] Indeed the task domain may display different properties in different regions. Thus independent goals initially might facilitate the parallel investigation of multiple useful meta actions.

Initializing the point population and generating new point population content takes the form of the following stochastic process: 1) With uniform probability select x and y co-ordinates defining the initial location of the ball (and when applicable ball goal location) over the interval $[0, 1]$; 2) Initial velocity is zero. 3) Each pair of co-ordinate pairs are tested to verify their legality i.e., they cannot appear within a wall object. 4) Test the candidate (sub)goal location to verify that it is not already 'touching' the ball or corresponds to the global goal location.

Fitness evaluation applies a competitive fitness sharing function (Equation (1)) to characterize the overall utility of each policy. To do so, each host, h_i, is evaluated against each point, p_k, thus requiring a domain specific characterization of the reward collected over an episode $G(h_i, p_k)$. The pinball domain defines an instantaneous reward, $r(t_s)$, of $-1(-5)$ for each use of the no change (any other) action respectively and a 'goal' reward τ of $10,000$ [6]. The latter is used here to represent the exploratory 'budget' for the pinball domain. An episodic reward, $g(h_i, p_k)$, is therefore defined in terms of the accumulated instantaneous reward: $\sum_{t_s=0}^{E_{max}} r(t_s)$, where this is always a negative value. However, we are interested in enforcing a dynamic limit on the amount of time a host spends on any one episode. Let us assume that after completing each episode a corresponding accumulated reward is estimated and normalized relative to τ to establish the remaining evaluation 'budget'. Thus, relative to point evaluation k, $\hat{\tau}(h_i, k) = -\frac{\tau}{k} \sum_{j=0,\dots,k-1} g(h_i, p_k)$ i.e., $\hat{\tau}(\cdot)$ is always positive. Thus as long as the difference between the remaining evaluation budget and episode specific reward is positive, the evaluation of host h_i against point p_k continues. We can now characterize the reward collected over an episode as follows:

$$\begin{array}{rc} \text{IF} & (\hat{\tau}(h_i, k) + g(h_i, p_k)) > 0 \\ \text{THEN} & G(h_i, p_k) \leftarrow 1 + \frac{g(h_i, p_k)}{\tau} \\ \text{ELSE} & G(h_i, p_k) \leftarrow 0 \end{array} \tag{2}$$

[1]Conversely, the value function approach proposed by [6] explicitly limits exploration to those policies that end where a previous policy left off.

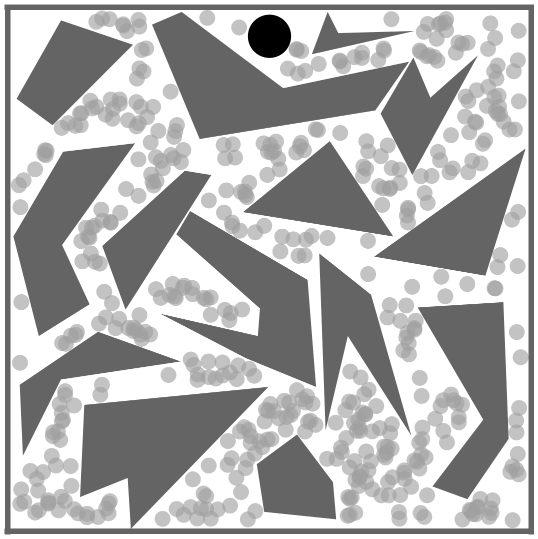

Figure 2: The maze of the pinball domain as defined by [6]. The single global goal location appears as the large 'black hole' (centre top). Grey circles correspond to initial ball locations employed during test.

Table 1: SBB Parameterization. t_{max} is the total generation limit over all levels; ω is the maximum number of symbionts a host may support under a variable length representation; H, P is the host/ point population size; H_{gap}, P_{gap} is the number of individuals replaced during breading in the host/ point population; p_{xx} denote the frequency with which different search operators are applied; *numRegisters* and *maxProgSize* represent the number of register and maximum instruction count for (symbiont) programs.

Host (solution) level			
Parameter	Value	Parameter	Value
t_{max}	210	ω	10
H, P	120	H_{gap}, P_{gap}	60, 20
p_{md}	0.7	p_{ma}	0.7
p_{mm}	0.2	p_{mn}	0.1
Symbiont (program) level			
numRegisters	8	*maxProgSize*	48
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

where the assumption is made that the 'average reward' for the first episode assumes a value of $\hat{\tau}(h_i, 0) = 500$, thereafter the estimated values from each completed episode(s) are employed.

The generic parameterization for SBB essentially follows from that used in the earlier study under a supervised learning context [13], Table 1. In this case, however, the introduction of hierarchical policy search implies that the total generation limit, t_{max}, is divided equally across each level.

A total of three experiments will be considered: **Case 1 – Single level:** SBB only builds a single level, thus no capacity exists for defining hierarchical policies. Relative to the parameterization of Table 1, the symbiont per host and program length limits are doubled ($\omega = 20$, *maxProgSize* = 96). **Case 2 – Two level, common goal:** This scenario introduces hierarchical policy discovery (two levels), but with respect to the single 'global' goal or maze exit (Figure 2). **Case 3 – Two level, independent goals:** Hierarchical policy discovery again appears (two levels), but this time level 0 lets the point population define goal locations as well as start conditions for the pinball. Evolution at level 1 is again conducted relative to the common 'global' goal.

4.3 Baseline value function approximation method

The focus of this research lies in policy search as opposed to value function approximation. However, we also include a baseline result using a recently proposed Fourier Basis for the SARSA value function method [7]. The source code for this was recently made available, as is the Pinball task domain itself.[2] In essence, a model based

[2]http://people.csail.mit.edu/gdk/software.html

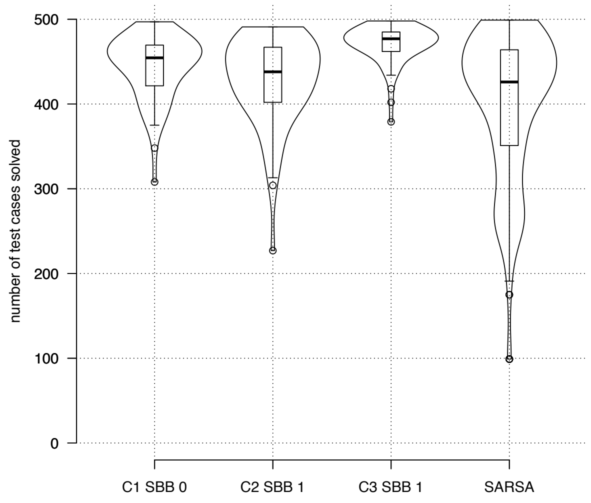

Figure 3: Generalization performance or count of number of test points solved by the champion host (y-axis). 500 test points in total. 'SBB x' denotes performance at highest available level 'x'; 'Cx' distinguishes between different SBB deployments or Case 1 through 3. SARSA is the baseline Fourier Basis value function approximator.

Table 2: p-value for pairwise Mann-Whitney non-parametric hypothesis test of configuration 'C3' versus all others.

Pairwise comparison	p-value
C3 versus SARSA	1.56×10^{-13}
C3 versus C1	2.9×10^{-6}
C3 versus C2	7.2×10^{-8}

value approximation identifies regions with common state–actions. The Fourier basis was recently shown to be competitive with value function methods employing previously proposed basis functions [7]. The principle motivation for including a value function approximation method here is to establish a baseline for generalization under the the pinball task. Specifically, the previous work in this domain only considered post training evaluation against two start positions [6]. Given that only one value function method will be utilized, hereafter we refer to it as SARSA.

4.4 Results

4.4.1 Generalization test

500 initial pinball locations (sampled with uniform probability), or the small grey balls in Figure 2, are used to assess post training performance of a single 'champion' host[3] from each trial. Naturally, the single global goal remains the same i.e., find a path to the 'exit' hole at the top centre of the maze. The distribution of test cases solved by the champion host is established across 60 independent trials. A violin plot summarizes the resulting distribution for each of the above three SBB configurations and SARSA, Figure 3. The wide variation in the distribution of SARSA solutions illustrates that the task is indeed nontrivial. All SBB solutions demonstrated better consistency.

The single level experiment effectively establishes a median base line in which 50 test cases cannot be solved ('C1' in Figure 3). Adding an additional level (at a corresponding reduced host size, program limit and generation per level limit) is unable to provide any improvement when the same global goal condition is retained at both levels ('C2' in Figure 3). Indeed, the use of a common goal for both levels of evolution results in a much lower solution consistency. Conversely, letting the point population define arbitrary target locations during the evolution of meta actions (level 0) and reintroducing the global goal location at level 1 provides a statistically significant improvement relative to either baseline ('C3' in Figure 3). Table 2 summarizes p-values under the Mann-Whitney non-parametric hypothesis test; Case 3 is independent in all cases. The natural implication of this is that encouraging diversity at level 0 is more significant in constructing 'good' meta actions for later use.

[3]Identified w.r.t. an independent validation sample of ball locations.

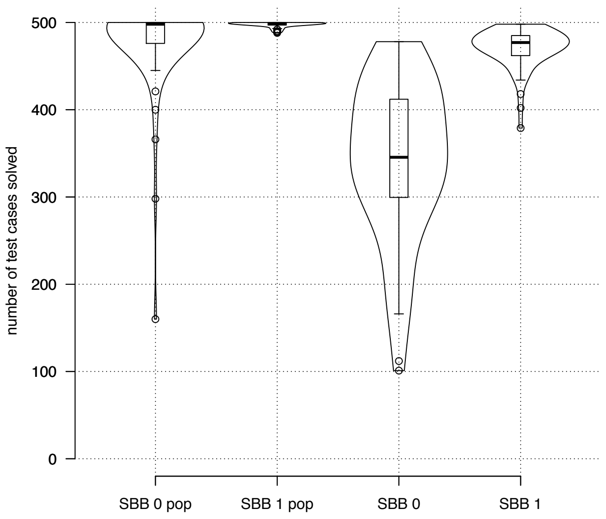

Figure 4: Performance of Case 3 SBB hosts over 500 test points. SBB 0 and SBB 1 denote first and second levels of SBB. "pop" identifies columns with cumulative solution count performance as estimated across the entire final population whereas the other columns denote performance of the single best individual from a run.

Further insight into this conclusion can be obtained by looking more closely at the Case 3 experiment. Figure 4 describes the test performance of the single best level 0 host (meta action) and level 1 host from Case 3 as well as the corresponding cumulative *population wide* performance. That is to say, the cumulative population wide performance counts the total number of unique test cases solved by all members of the host population as opposed to the single 'champion' host detailed in Figure 3. It is now apparent that meta actions (SBB 0) typically only solve 50 percent of the test cases. However, when measuring the cumulative population wide performance of meta actions (SBB 0 pop) all the test cases might have solutions. Thus, competitive fitness sharing has successfully cached a diverse set of behaviours across the host population.

These behaviourally diverse level 0 hosts thus represent good candidate meta actions for host–symbiont development at level 1. This is how hierarchical SBB supports 'complexification' without falling into the over-learning trap. Thus, the champion performance at level 1 ('SBB 1' column, Figure 4) generally matches the cumulative population wide performance at level 0 ('SBB 0 pop' column) with a much tighter consistency. Hosts at level 1 have thus been able to leverage the diversity of meta actions to form a single highly fit solution at level 1. Note, however, that the above analysis is all post evolution, whereas during evolution effective hierarchies need to be discovered without reference to such 'global' pictures of host capability.

4.4.2 Solution Complexity

From an architectural perspective, the relative counts for the number of symbionts deployed per level and corresponding instruction counts (post intron removal), give some insight to the relative complexity of solutions. Figure 5 summarizes the symbiont counts per host as a function of experiment (C1, C2, C3) and layer (SBB 0 and SBB 1). The non-hierarchical scenario (C1) naturally results in hosts with the most variance and highest single layer symbiont counts (column 'C1 SBB 0'). Conversely, both C2 (no goal variation) and C3 (goal variation specified at level 0) observe greater variance in symbiont counts at level 0 than at level 1 (compare Cx SBB 0 to the corresponding Cx SBB 1). However, it is also apparent that the C3 configuration (goal variation) also maintains greater variation than C2 (no goal variation).

In the case of instruction count, a similar pattern is noted (Figure 6). However, a significant increase in the level 0 instruction count appears for hierarchical SBB with goal variation (column C3 SBB 0 versus C2 SBB 0). This is most likely an artifact of the more diverse scenarios encountered during training. Again, when SBB is limited to a single level the instruction counts per symbiont also undergo a significant increase. By way of comparison, under the original study of the pinball domain [6], a value function representation assumed a 'Fourier basis' in which between 256 and 1296 basis functions were required *per policy*. Under SBB there are typically 5-10 symbiont programs per host and 7-15 instructions per symbiont program or between 35 to 150 instructions per policy.

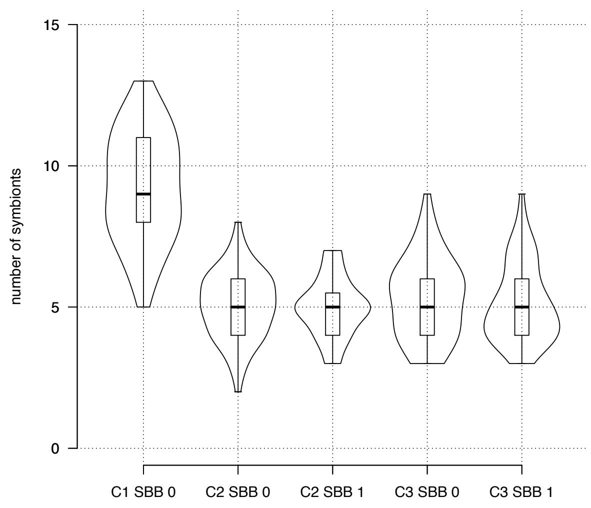

Figure 5: Symbiont Counts per Host. 'SBB *x*' denotes performance at highest available level '*x*'; 'C*x*' distinguishes between different SBB deployments or Case 1 through 3.

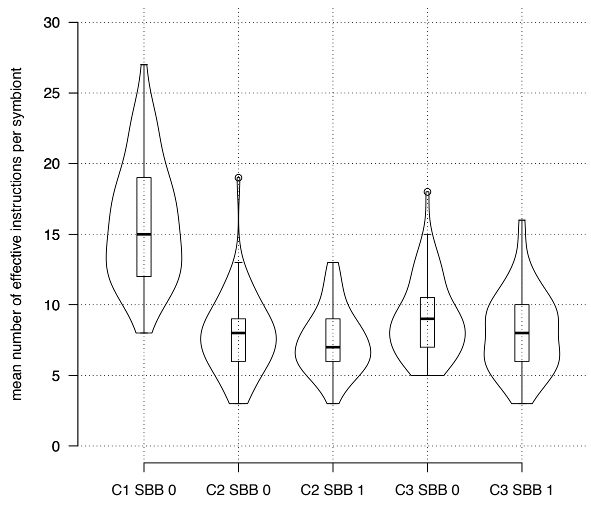

Figure 6: Average Instruction Counts per Symbiont. 'SBB *x*' denotes performance at highest available level '*x*'; 'C*x*' distinguishes between different SBB deployments or Case 1 through 3.

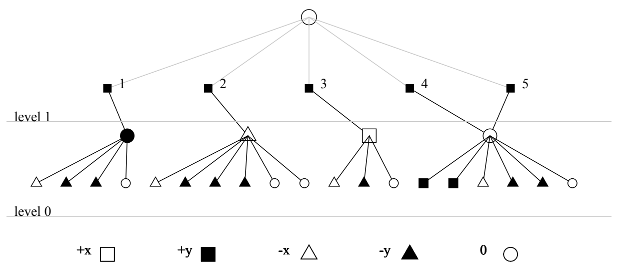

Figure 7: Structure of a hierarchical SBB solution solving 492 of 500 test cases. Top-centre circle represents level 1 host. Black squares represent level 1 symbionts indexed by this host. Each level 1 symbiont assumes one level 0 host as its meta action, represented here by the black/ white circle/ square. Each level 0 host indexes multiple level 0 symbionts. The shape of each level 0 symbiont denotes which atomic action is assumed (see legend at the bottom of the figure).

4.4.3 Deploying meta actions

In order to demonstrate the application of meta actions evolved at level 0 we need to establish to what degree their purposeful application appears. Thus, in order to provide some insight into this we consider both structural and behavioural aspects of a specific C3 SBB 1 solution solving 492 of the 500 test cases.

Figure 7 summarizes the architecture for this particular solution. The *large* circle – top centre – denotes the single level 1 host or root of the decision tree. This level 1 host is comprised of 5 unique symbionts, or *small* black squares labeled 1 through 5. Each of these level 1 symbionts assumes a meta action as defined by a host evolved at level 0, or the *large* black circle, white triangle, white square, and white circle of Figure 7. The level 0 hosts (defining meta actions) index a subset of level 0 symbionts each with a corresponding atomic (task specific) action. Naturally, as each symbiont learns a unique context for deploying its action, the same atomic action may appear in multiple places at level 0. Likewise, multiple level 1 symbionts may assume the same meta action as their action cf, meta actions deployed in multiple contexts. Figure 7 supplies a legend mapping level 0 symbiont symbols to corresponding atomic actions.

Having established the basic structure of a hierarchical policy, we can now map the states at which level 0 hosts are deployed by symbionts from the level 1 host (root node of the decision tree) relative to a specific test condition, Figure 8. The start condition for this particular trace corresponds to the large grey circle (bottom left) and global goal is the large black circle (top centre). The overall trajectory is expressed in terms of shapes corresponding to the meta action deployed at each time step (defined by Figure 7).

At least three factors are now readily apparent: 1) the degree of interleaving or cooperation between different meta actions (level 0 hosts); 2) the relative specialization of meta actions; and, 3) the degree of reliance on wall bouncing / exploration for reorientation of the pinball versus linear/ greedy exploitation of a given direction at different parts of the trajectory. Thus, for example, the level 0 host / meta action represented by a white triangle seems to be deployed predominantly on the left side of the board while the white square, white ellipse, and black ellipse hosts are used predominantly on the right side. This decomposition is a direct result of the communication between level 1 symbionts at each time-step. The level 1 symbionts have learnt appropriate contexts for their meta actions and are able to express this information through bidding.

We can also summarize the consistency of bidding behaviours across all test points by plotting the (x, y) co-ordinate for which different level 1 symbionts (thus host 0 deployment) represent the winning bid, Figure 9. The resulting spatial organization clearly reflects a specific distribution of meta actions in which only the case of (level 0) host 4 appears to be redundant.

5 Conclusion

A symbiotic framework for evolutionary policy search using GP is presented. To do so a hierarchy of programs is evolved through independent runs. An initial run discovers appropriate meta actions (level 0) – synonymous with the development of a run time library (RTL). The second run evolves contexts for deploying the meta actions / RTL content. Unlike previous schemes for utilizing RTL no additional syntactic constructs are necessary. Specific factors contributing to this include:

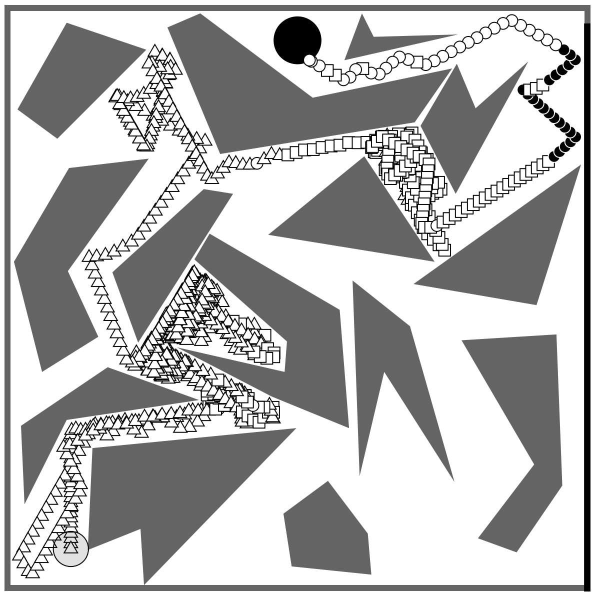
Figure 8: Trajectory in terms of meta action deployment w.r.t. hierarchical policy of Figure 7. Large grey circle represents start location for this specific test case. Large black circle represents the global target. Shapes correspond to lev 0 hosts (meta actions) deployed at each location/time-step in trajectory, as mapped to specific hosts in Figure 7.

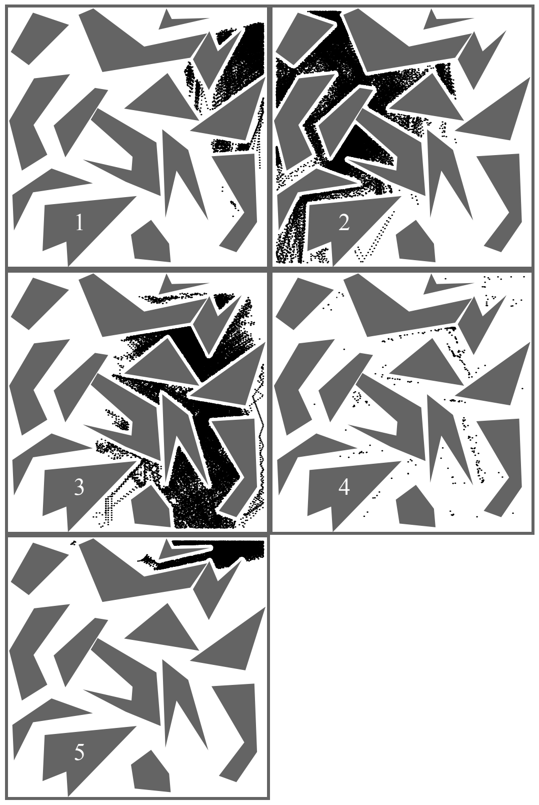

Figure 9: Distribution of winning bids across all test conditions w.r.t. level 1 symbionts of Figure 7. The winning bid from a level 1 symbiont defines the context under which meta actions are deployed. A clear partitioning of the state space results.

- Symbiont programs explicitly separate action (as applied to the task domain) from context (a bidding behaviour).
- Hosts define candidate policies by selecting combinations of symbionts for evaluation as a group. Fitness is only directly expressed at the level of hosts. A valid policy requires a minimum of two symbionts with dissimilar actions. Evaluation of a host is established by determining the bid values of symbionts currently indexed by the host. Only the symbiont with highest bid gets to suggest their action.
- Meta actions are discovered by letting the hosts of a first 'run' of SBB define a RTL. A second run discovers the context for deploying the previously discovered meta actions. Future research could consider letting the actions of later levels of the hierarchy continue to incorporate atomic actions. Moreover, there is no reason why meta actions cannot be developed over more than two levels.
- The point population defines training *and* sub-goals during the discovery of meta actions / RTL. This was conducted under a tabula rasa model of point generation. Future research could consider the utilization of feedback between host and point population (competitive coevolution [3, 4]) to direct development more explicitly. Similar use could be made of best practices from layered learning [20] and or robot shaping / scaffolding [1]. What is clear from this study however, is that good RTL / meta action discovery appears to benefit from maintaining as much environmental diversity as possible.

6 Acknowledgements

The authors acknowledge the support of NSERC and CFI New Opportunities grants in conducting this research.

References

[1] J. Bongard. Behaviour chaining: Incremental behavior integration for evolutionary robotics. In *International Conference on Artificial Life XI*. MIT Press, 2008.

[2] M. Brameier and W. Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.

[3] J. Cartlidge and S. Bullock. Combating coevolutionary disengagement by reducing parasite virulence. *Evolutionary Computation*, 12(2):159–192, 2004.

[4] E. D. de Jong. A monotonic archive for Pareto-coevolution. *Evolutionary Computation*, 15(1):61–94, 2007.

[5] 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*, pages 531–542, 2004.

[6] G.D. Konidaris and A.G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1015–1023, 2009.

[7] G.D. Konidaris, S. Osentoski, and P. Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 380–385, 2011.

[8] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, 1992.

[9] I. Kushchu. Genetic programming and evolutionary generalization. *IEEE Transactions on Evolutionary Computation*, 6(5):431–442, 2002.

[10] X. Li, C. Zhou, W. Xiao, and P. C. Nelson. Direct evolution of hierarchical solutions with self-emergent substructures. In *International Conference on Machine Learning Applications*, pages 337–342, 2005.

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

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

[13] P. Lichodzijewski and M. I. Heywood. Symbiosis, complexification and simplicity under GP. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 853–860, 2010.

[14] P. Lichodzijewski, M. I. Heywood, and A. N. Zincir-Heywood. CasGP: Building cascaded hierarchical models using niching. In *IEEE Congress on Evolutionary Computation*, pages 1180–1187, 2005.

[15] A. R. McIntyre and M. I. Heywood. Classification as clustering: A Pareto cooperative-competitive GP approach. *Evolutionary Computation*, 19(1):137–166, 2011.

[16] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In *European Conference on Genetic Programming*, pages 160–175, 2001.

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

[18] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1419–1426, 2011.

[19] R. Thomason and T. Soule. Novel ways of improving cooperation and performance in ensemble classifiers. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1708–1715, 2007.

[20] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving soccer keepaway players through task decomposition. *Machine Learning*, 59:5–30, 2005.

[21] S. X. Wu and W. Banzhaf. A hierarchical cooperative evolutionary algorithm. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 233–240, 2011.