EVOLVING POLICIES TO SOLVE THE RUBIK'S CUBE: EXPERIMENTS WITH IDEAL AND APPROXIMATE PERFORMANCE FUNCTIONS

by

Robert Smith

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2016

© Copyright by Robert Smith, 2016

Dedicated to puppies, kittens, hamsters, interesting parrots, black licorice (it needs some love), party mix, hoola hoops, cyberpunk sci-fi, Vietnamese cuisine, turtles (the tortoise's smug ocean cousin), performance functions, function performance, graphics processing units, horror movies, medical science, non-medical science, pilots, Coheed & Cambria, Steam, webcomics, Satoshi Kon, William Gibson, internet outrage, and shoes.

Table of Contents

List of Tables . . . . **v**

List of Figures . . . . **vi**

Abstract . . . . **viii**

Acknowledgements . . . . **ix**

Chapter 1 Introduction **1**

Chapter 2 Background **3**

2.1 Reinforcement learning 3

2.2 Solving the Rubik's Cube through heuristic search 4

2.3 General Problem Solver programs 6

2.4 Decomposing the Rubik's Cube Search Space 6

2.5 Incremental evolution and Task transfer 8

2.6 Symbiotic Bid-based GP 10
2.6.1 Coevolution 14
2.6.2 Code Reuse and Policy Trees 16

Chapter 3 Expressing the Rubik's Cube task for Reinforcement Learning **18**

3.1 Formulating fitness for task transfer 19
3.1.1 Subgroup 1 - Source task 19
3.1.2 Subgroup 2 - Target task 19
3.1.3 Ideal and Approximate Fitness Functions 20

3.2 Representing the Rubik's Cube 22

3.3 Policy tree structure 23

Chapter 4 Evaluation Methodology **25**

4.1 Parameterization 25

4.2 Qualifying experimentation 26

4.2.1 Disabling Policy Diversity 26
4.2.2 Random Selection of Points 27

Chapter 5 Results **28**

5.1 Standard 5 Twist Model 28

5.2 Disabling Policy Diversity 31

5.3 Random Selection of Points 34

5.4 Phasic task generalization 34

Chapter 6 Conclusions and Future Work **39**

6.1 Conclusions 39

6.2 Future Work 40
6.2.1 5 Twist Completion 40
6.2.2 Twist Expansion 41
6.2.3 Complexification of Policy Trees 42
6.2.4 Rubik's Cube as a reinforcement learning benchmark 42

Appendix A – Constructing the 10 twist database **44**

Bibliography **46**

List of Tables

Table 2.1 Count of unique states enumerated by IDA* search tree as a function of depth. Depth is equivalent to the number of twists from the solved Cube. Table assumes three different twists per face (one half twist, two quarter twists). 5

Table 3.1 The Rubik's Cube group is defined as $(G, \cdot)$ where G represents the set of all possible actions which may be applied to the cube and the $\cdot$ operator represents a concatenation of those actions. . 23

Table 4.1 Generic SBB parameters. t_{max} generations are performed for each task or $2 \times t_{max}$ generations in total. Team specific variation operators P_D, P_A pertain to the probability of deleting or adding a learner to the current team. Learner specific variation operators P_m, P_s, P_d, P_a pertain to the probability of mutating an instruction field, swapping a pair of instructions, and deleting or adding an instruction respectively. 26

List of Figures

Figure 2.1 Basic architecture of SBB. Team population defines 'teams' of learner programs, e.g. $tm_i = \{s_1, s_4\}$. Fitness is evaluated relative to the content of the Point population, i.e. each Point population member, p_k, defines an initial state of for the Cube. 10

Figure 2.2 Pareto archive of outcomes for three teams tm_i and three points p_i. 15

Figure 2.3 Phased architecture for code/policy reuse in SBB. After the first evolutionary cycle has concluded, the Phase 1 team population represent actions for the Phase 2 learner population. Each Phase 2 team represents a candidate switching/root node in a policy tree. Teams evolved during Phase 2 are learning which previous Phase 1 knowledge to reuse in order to successfully accomplish the Phase 2 task. 17

Figure 3.1 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. 24

Figure 5.1 Average number of Cube configurations solved at subgroup 2 (target task) by SBB. Descending curves (solid) represent average individual-wise performance. Ascending curves (dashed) represent cumulative performance. The y-axis represents the percent of $17,675,698$ unique scrambled Cube configurations solved. 29

Figure 5.2 Percent of $17,675,698$ Cube configurations solved at the Target subgroup. Individual-wise ranking (descending) and cumulative ranking (ascending). Distribution reflects the variation across 5 different runs per experiment. 30

Figure 5.3 Policy tree solving 80% of the Cube configurations under the Target task. Level 0 nodes represent atomic actions. Level 1 nodes represent teams indexed as actions by learners from the single phase (level) 2 team. Each atomic action is defined by an xy tuple in which $x \in \{B, G, O, R, Y, W\}$ denote one of six colour Cube faces, and $y \in \{L, R\}$ denote left (counter clockwise) or right (clockwise) quarter turns. 31

Figure 5.4 Mean solution rate for five team populations across a test set against a 2nd subgroup target task without diversity maintenance. Individual-wise ranking with an average best team solving approximately 64% of all cases. 32

Figure 5.5 Distribution of solution rates for five team populations across a test set against a 2nd subgroup target task without diversity maintenance. Individual-wise ranking with the median best team solving approximately 64% of all cases. 33

Figure 5.6 Mean solution rate for five team populations across a test set against a 2nd subgroup target task using random point selection. Individual-wise ranking (descending) and mean cumulative ranking (ascending) with an average best team solving approximately 32% of all cases. 35

Figure 5.7 Distribution of solution rates for five team populations across a test set against a 2nd subgroup target task using random point selection. Individual-wise ranking with a median best team solving approximately 33% of all cases. 36

Figure 5.8 Phasic task generalization. Distribution of fitness for five team populations across a test set against a 2nd subgroup target task using the target task as a goal for 2-phase populations. Individual-wise ranking (descending) and cumulative ranking (ascending) with an average best team solving approximately 78% of available cases. 38

Abstract

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

A second set of experiments are then performed to qualify the relative contributions for two components for discovering policy trees: Policy diversity maintenance and Competitive coevolution. Both components prove to be fundamental. Without support for each, performance only reaches $\approx 55\%$ and $\approx 23\%$ respectively.

Acknowledgements

I'd like to acknowledge that we all get a little hungry and if nothing else reading this thesis will provide you with a great way to appease a case of the nums. Therefore, below you will find a recipe for pancakes that I've been using for a long time. Like most good recipes it's unassuming and simple while being incredibly satisfying. This recipe can be found on AllRecipes and it was posted by Dakota Kelly, the superstar of the pancake universe. At least, I assume she is.

To start, the best way I've found to cook pancakes is not to add oil to a heated surface and throw the batter into it all willy-nilly. Instead, I find it far better to put the fat into the batter itself and give it a good wisk. Obviously your experience may vary based on the kind of cooking surface you use: this would likely work better on non-stick by the nature of the surface itself. Don't skip out on the butter just because we're adding oil to the batter, however. More fat will make the pancakes more moist and butter is a much better flavour enhancer, so we don't want to lose it! With that said, here are the ingredients you'll need (in metric, to accomodate the majority of the world):

192 g of all-purpose flour

20 ml of baking powder

5 ml of salt

15 ml of white sugar

320 ml of milk

1 large egg

45 ml of melted butter

45 ml of vegetable oil (or other flavourless oil of your choice).

1. In a large bowl sift together flour, baking powder, salt, and sugar. Make a well in the centre. Pour in the milk, egg, oil, and melted butter; mix until smooth, preferrably with a wisk.
2. Heat a griddle or frying pan over medium-high heat. Pour or scoop the batter onto the griddle, using approximately 1/4 cup for each pancake. Brown on both sides and serve hot.

Chapter 1

Introduction

Invented in 1974, the Rubik's Cube has been the target of attempted optimization tasks due to the inherent complexity of the puzzle itself. The 'classic' $3 \times 3 \times 3$ Rubik's Cube (hereafter, the Rubik's Cube or Cube) represents a game of complete information consisting of a discrete characterization of states and actions. Actions typically take the form of a clockwise or counter clockwise twist (quarter turn) relative to each of the 6 cube faces, i.e. a total of 12 *atomic* actions. A Cube consists of 26 cubies of which there are 8 corner, 12 edge and 6 centre cubies; the latter never changing their position, thus defining the colour for each face. Each face consists of 9 facelets that, depending on whether they are edges or corners, are explicitly connected to 1 or 2 neighbouring facelets. The total number of states is in the order of 4.3×10^{19} [23] and, unlike many continuous domains, even single actions result in a third of the cubies changing position. Thus, as more cubies appear in their correct position, applying actions is more likely to *increase* the entropy of the Cube's state. Conversely, the Cube possesses many symmetries, thus sequences of moves can potentially define operations that move (subsets of) cubies around the Cube without displacing other subsets of cubies; or, from a group theoretic perspective, 'invariances' are identified that provide transforms between subgroups.

In short, the Rubik's Cube task has several properties that make the task an interesting candidate for solving using reinforcement learning (RL) techniques. The Cube is described by a 54 dimensional vector, or large enough to potentially result in the curse of dimensionality [37], but small enough to warrant direct application of a machine learning algorithm without requiring specialized hardware support. Moreover, the number of possible actions (12) is also higher than typically encountered in RL benchmarks, also further contributing to the curse of dimensionality. The latter point is particularly true when solutions are sought that solve an initial Cube configuration in a minimum number of moves. Finally, given that it is already known that

invariances exist for transforming the Cube between different subgroups, it seems reasonable that a learning algorithm should be capable of discovering such invariances. It is currently unknown whether RL algorithms can address these issues for the Rubik's Cube task domain. Moreover, I am not interested in adopting a solution that assumes the availability of task specific instructions/operators.

I investigate these questions under a coevolutionary genetic programming (GP) framework for policy search that has the capacity to incrementally construct 'policy trees' from multiple (previously evolved) programs [5, 22, 20, 19]. Thus, the term *policy tree* has nothing to do with the representation assumed for each program, but refers to the ability to construct solutions through an explicitly hierarchical organization of previously evolved code. Moreover, each 'individual' (or policy) is composed from multiple programs that learn to decompose the original task through a bidding metaphor or cooperative coevolution [27].

This study will develop the approach to task transfer between sequences of objectives using two subgroups representing consecutive fitness objectives for solving the Rubik's Cube. The resulting two level policy tree is demonstrated to produce a single *individual* that solves up to 80% of the scrambled Cubes, where there are $17,675,698$ initial Cube states in total and each run of evolution is limited to sampling 100 Cube configurations per generation (14% of scrambled Cubes are encountered *once* during training). Moreover, diversity maintenance ensures that the population is able to cumulatively solve 100% of the scrambled Cubes. The GP representation is limited to a generic set of operators originally employed for classification tasks, thus in no way specific to the Rubik's Cube task. Indeed, the same 'generic' instruction set appears for RL tasks such as the Acrobot [5], Keepaway soccer [20] and Half Field Offense [21].

As a means of justifying the algorithmic features of the formulated GP, this thesis also investigates how diversity maintenance and selection policies effect the overall accuracy of generated policy trees. I demonstrate that in order to address high dimensional state spaces, such as those encountered within the context of the Rubik's Cube, it is necessary to explicitly promote policy diversity and learn which training scenarios are more informative. Without these capabilities only 23% to 55% of the Cube configurations might be solved.

Chapter 2

Background

In the following I present related material pertinent learning to identifying strategies to the Rubik's Cube. In essence I am interested in learning by interacting with the Cube. Hence, from a generic machine learning perspective, this is an example of a reinforcement learning task (Section 2.1). However, research to date concentrates on discovering sequences of moves for solving the Rubik's Cube using: Heuristic Search methods (Section 2.2) or General problem solver programs (Section 2.3), i.e. no learning algorithm. There is also a body of research – historically utilized with heuristic search methods – that formulates information on appropriate search objectives specific to the Cube (Section 2.4). I will make use of this later for defining suitable objectives for my GP approach, particularly with regards to learning how to reuse policies under different objectives (Section 2.5). Finally, Section 2.6 presents the overall framework for Symbiotic Bid-Based (SBB) GP. This represents the only GP framework that provides for automated task decomposition, code reuse, and competitive coevolution – properties that I will later show are all necessary to successfully solve the Rubik's Cube task. I develop a Java code base to implement SBB, but the framework itself was originally proposed by [26].

2.1 Reinforcement learning

There are two basic machine learning approaches for addressing the temporal sequence learning problem: (value) function optimization [17], [37] and policy search/ optimization [29]. In the case of function optimization each state–action is assumed to result in a corresponding 'reward' from the task domain. Such a reward might merely indicate that the learner has not yet encountered a definitive 'failure' condition. A reward is generally indicative of the immediate cost of the action – as opposed to the ultimate quality of the policy. In this case the goal of the temporal sequence learner is to learn the relative 'value' of state–action pairs such that the 'best' action can

be chosen given the current state. Moreover, such a framework explicitly supports online adaptation [37]. Given that there are typically too many state–action pairs to exhaustively enumerate (as is the case with the Rubik's Cube), some form of function approximation is necessary. Moreover, it is also generally the case that the gradient descent style credit assignment formulations frequently employed with value function methods (such as Q-learning or Sarsa) benefit from the addition of noise to the action in order to visit a wider range of states. Moreover, an annealing schedule might also be assumed for balancing the rate of stochastic versus deterministic actions of which ε-greedy represents a well known approach.

Policy optimization, on the other hand, does not make use of value function information [29]. Instead the performance of a candidate policy/ decision maker is assessed relative to other policies with the ensuing episode (sequence of state–action pairs) left to run until some predefined stop criterion is encountered. This represents a direct search over the space of policies that a representation can describe. Most evolutionary methods take this form, with neuroevolutionary algorithms such as CoSyNE [8], NEAT [35] or CMA-ES (as applied to optimizing neural network weights) [16] representing specific examples.

2.2 Solving the Rubik's Cube through heuristic search

Notable examples of optimal Rubik's Cube solutions were performed on $3 \times 3 \times 3$ Rubik's Cubes using iterative-deepening A* (IDA*) [15, 23]. IDA* is a shortest path graph traversal algorithm which begins at a root state node and performs a modified depth-first search until a goal state node has been reached. Rather than using the standard metric of depth as the current shortest distance to the root,[1] IDA* utilizes a compound depth-cost function where the search depth is a function of the current cost to travel from the root node to a level and the heuristic estimation of cost from the current level to a goal state. In the case of the Cube, a combined twist metric of 90 and 180 degree twists was originally used [23]. The IDA* search process yielded 577,368 search nodes at a search depth of 5 and increased to 244,686,773,808 at a search depth of 10. Depths greater than 10 yield state node counts of trillions and greater (Table 2.1). This function of depth does not account for duplicate states

[1] Hence, the mechanism adopted for prioritizing which node to open next.

Table 2.1: Count of unique states enumerated by IDA* search tree as a function of depth. Depth is equivalent to the number of twists from the solved Cube. Table assumes three different twists per face (one half twist, two quarter twists).

Depth	Nodes	Depth	Nodes
1	18	2	243
3	3,240	4	43,254
5	577,368	6	7,706,988
7	102,876,480	8	1,373,243,544
9	18,330,699,168	10	244,686,773,808
11	3,266,193,870,720	12	43,598,688,377,184
13	581,975,750,199,168	14	7,768,485,393,179,328
15	103,697,388,221,736,960	16	1,384,201,395,738,071,424
17	18,476,969,736,848,122,368	18	246,639,261,965,462,754,048

(such as states generated by performing two 180 degree twists on the same side), but provides insight into how quickly the problem space grows. As the outcome of the IDA* algorithm is to provide an optimal path from a root node to any state within a set of pre-determined goal nodes, the researchers created a problem space of 10 Rubik's Cubes which had 100 random twists applied and attempted to determine the upper bound on the number of twists required to solve any Rubik's cube configuration. They shared results for 10 experiments in which the optimal depths were found to be between 16 and 18 twists. In order to find these optimal paths, they needed to generate up to 1 trillion search nodes [23].

A joint project between the University of Alberta and the University of Regina involved solving puzzles using heuristic-search algorithms (mainly IDA*) whereby a neural network–IDA* hybrid was proposed for learning how to create and adjust a heuristic function across multiple iterations of the search. In their approach they used multiple instances of the Korf solvable cubes [23] and allowed IDA* to attempt to find a solution for each. Once a certain amount of time has passed or a certain number of solvable instances have been successfully solved, the algorithm will reconfigure based on important features and restart the search on the remaining unsolved cubes. While this method shows definite improvement over time, it also generates a huge number of search states (even on small solvable instances) and takes a very long time to complete. In the first iteration (the base IDA* algorithm) they solved approximately

50% of the solvable instances. By iteration 7 they had solved 75.4% of the solvable instances at the cost of 11 days and 7 hours. During the final iteration of 14, they had solved 98.78% of all the solvable instances, but it had taken them 31 days and 15 hours. In that time their algorithm generated nearly 90 billion search nodes in total. While this is significantly better than the trillions of nodes required by Korf, the number of nodes being generated to perform heuristic search is intimidating when attempting to build on previous work.

2.3 General Problem Solver programs

One programmatic approach toward solving the Rubik's Cube is the General Problem Solver program. A General Problem Solver should be able to view the state of a system and produce an appropriate solution. This leads to another state under which the program will offer a newly discerned solution [24]. Since the program does not specialize on any feature of the system, but rather produces some policy for solving a 'big picture' view of the current state, it should be capable of solving a system of substates until a goal state is reached. For problems with a relatively small number of potential states or a large number of goal states a general solution is much easier to obtain. However, as the states of the system become more complex or difficult to solve, we begin to see the limitation of an approach under current computational boundaries. The solutions generated by a General Problem Solver program are defined by a series of high-level operations which are broken down into a series of low-level operations. In the case of a Rubik's Cube, we could define a general solution for putting a Rubik's Cube in a state of edge orientation (a high-level operation) by the series of twists applied to the faces of the cube (a series of low-level operations).

2.4 Decomposing the Rubik's Cube Search Space

A body of research has concentrated on identifying the worst case number of moves necessary to solve an $n \times n \times n$ Rubik's Cube using exhaustive search algorithms, e.g. IDA* [23, 25]. The basic idea is to use group theory to partition the task into subgroups / subproblems. An exhaustive search is deployed over complete enumerations of each subproblem in order to define specific twist sequences for solving an initially

scrambled Cube. Naturally, building each complete enumeration for each subgroup is expensive, particularly with respect to duplicate detection [25]. Most recently, Terabytes of storage were used by a group of researchers at Google to prove that the so called *God's number* for the special case of $n = 3$ is 20 under a half-twist metric [31]. The same group also applied their method to the quarter-twist metric, finding that key twist value to be 26.[2]

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

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

Two previous works have attempted to learn general strategies for unscrambling Rubik's Cube configurations through policy search [1, 28].Specifically, in [1] Baum and Durdanovic evolve programs under a learning classifier system in which they were able to successfully discover policies that took an initial scrambled cube configuration and moved it into a state in which half of the Cubies were in the solved state. To do so, an instruction set specific to the Cube task was introduced (not the case in this work of this thesis), and performance expressed in terms of a mixture of three metrics quantifying heuristic combinations of the number of correctly placed Cubies.

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

However, performance of the resulting system always encountered plateaus after which the performance function was not able to provide further guidance to process.

Conversely, Lichodzijewski and Heywood assumed a fitness function in which only Cube configurations up to 3 twists away from the solved cube were distinguished [27], i.e. any cube state beyond three twists resulted in the same (worst case) fitness. As a consequence, performance was essentially limited to solving for 1, 2 and 3 twists away from the solved state with frequencies of 100%, $\approx$ 60% and $\approx$ 20%. In this work, we assume the same coevolutionary GP framework as Lichodzijewski and Heywood, but build on the subgroup formulation utilized by El-Sourani et al in order to provide a fitness function able to guide the coevolutionary properties much more effectively. The objective being to evolve general policies for transforming scrambled Cubes into the penultimate subgroup (the last subgroup assumes a different set of actions, i.e. half twists as opposed to quarter twists).

For completeness, we also note one attempt to treat the Rubiks Cube as a problem in which the goal is to learn pair-wise instances of Cube states [14].[3] In this case, a sequence of K moves are applied to a Cube in the solution state. A neural network is then rewarded for applying the twist that moved the Cube from state K to $K-1$. Naturally, there is no attempt to guarantee the optimality of the sequence learnt, as the sequence of moves used to create Cube states are random, thus may even revisit previously encountered states. Moreover, the boosting algorithm assumed was not able to discover more meaningful neural networks for the task. Performance under test conditions (100,000 Cube configurations) was such that best performance was achieved for sequence lengths of 3 twists from the solved state ($\approx$ 90% of sequences solved), whereas sequences of 2 twists were solved at a lower accuracy ($\approx$ 80%).

2.5 Incremental evolution and Task transfer

Incremental evolution is an approach first demonstrated in evolutionary robotics in which progress to the ultimate objective is not immediately feasible [10, 2]. Instead, a sequence of objectives are designed and consecutively solved with respect to a common definition for the sensors characterizing the task environment (state space). Subsequently, there have been several generalizations, including Layered Learning

[3] This is an unpublished manuscript.

[36] and Task Transfer [38, 33]. Unlike incremental evolution, the later developments also considered policies that were developed under independent task environments (source tasks) and then emphasized their reuse as a starting point to solve a new (target) task. Conversely, incremental evolution emphasizes continuous refinement of the same solution across a sequence of objectives. Thus, previous approaches to incremental evolution have been demonstrated under neuroevolutionary frameworks in which the topology is fixed, but weight values continue to adapt between different objectives [10, 2].

In this work, we assume that different cycles of evolution are performed for each objective. Diversity maintenance maximizes the number of potential solutions to a task. When an objective is suitably solved (across an entire population), then the population content is 'frozen' and a new population initialized with the next objective. The new population learns how to solve the next objective by reusing some subset of previously evolved programs (policies). Moreover, solutions take the form of policy trees in which only a fraction of the programs comprising the solution need be executed to make each decision. Hence, although the overall policy tree might organize four to five hundred instructions over twenty to thirty programs, each decision only requires a quarter of the instructions/programs to be executed [21].

In short, the approach assumed here is closer to that of task transfer than incremental involution, and has been demonstrated under the task of multi-agent half-field offense HFO [21]. However, the HFO task has completely different properties, emphasizing policy discovery under a real-valued state space (albeit of a much lower state and action dimensionality than under the Rubik's Cube) with an emphasis on incorporating source tasks from different environments. Conversely, the Cube (at least as played here) does not introduce noise into states or action actuators and (unlike HFO) assumes source tasks with common state and action spaces. With this in mind, we adopt as our starting point the original architecture of hierarchical SBB [5, 22, 20, 19] and investigate the impact of providing different task objectives and identifying the contribution of different forms of diversity maintenance.

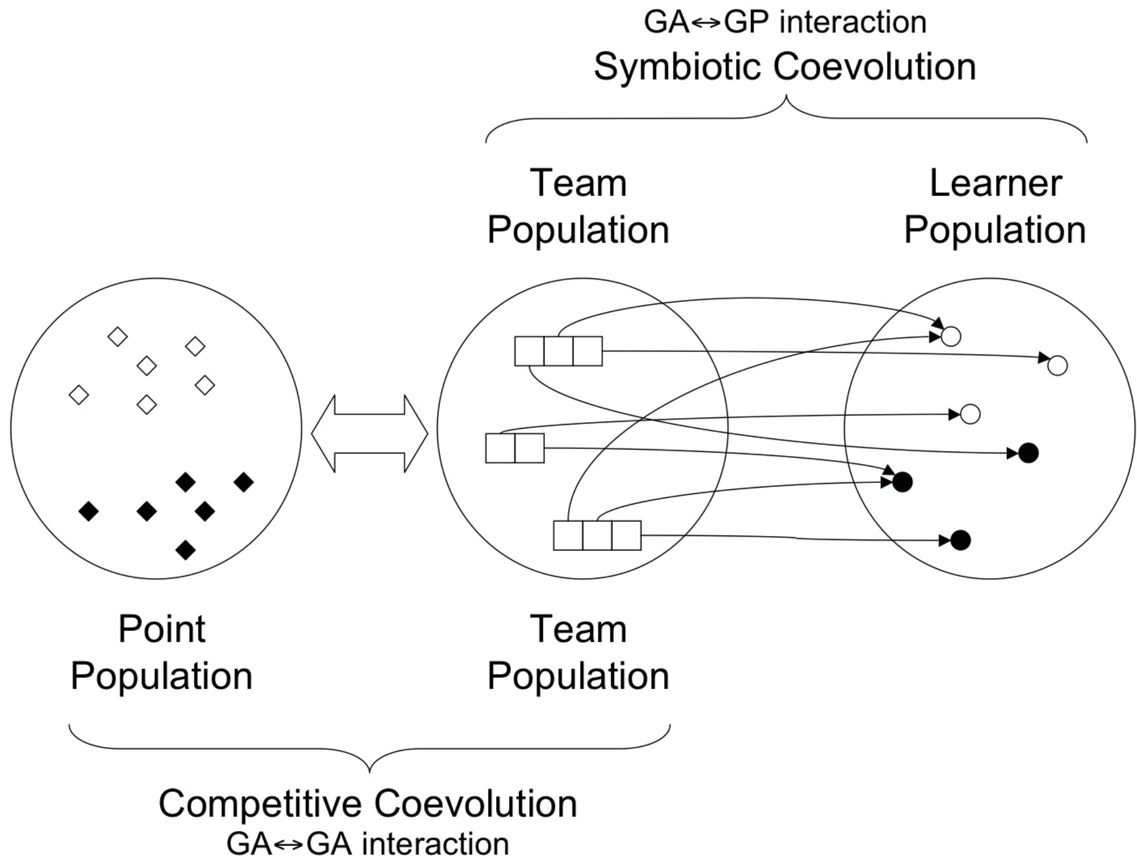


Figure 2.1: Basic architecture of SBB. Team population defines 'teams' of learner programs, e.g. $tm_i = \{s_1, s_4\}$. Fitness is evaluated relative to the content of the Point population, i.e. each Point population member, p_k, defines an initial state of for the Cube.

2.6 Symbiotic Bid-based GP

As noted above, several works have previously deployed SBB in various reinforcement learning tasks. In the following we therefore summarize the properties that make SBB uniquely appropriate for task transfer under the Rubik's Cube task. A total of three populations appear in the original formulation of SBB [28, 5, 22] as employed here: point population, team population and learner population, Figure 2.1.

The **Point population** (P) defines the initial state for a set of training scenarios against which fitness is evaluated. At each generation some fraction of Point population individuals are replaced, or the 'point gap' (G_P). In the Rubik's Cube task Point individuals, p_k, represent initial states for the Cube. For simplicity, the Point

population content is sampled without replacement (uniform p.d.f.) from the set of training Cube initial configurations (Section 4.1), i.e. no attempt is made to begin sampling with initial Cube states 'close' to the goal state.

The **Team population** (T) represent a variable length[4] GA that indexes some subset of the members of the (Learner) Program population (S). Each team defines a subset of programs that learn how to decompose a task through an inter-program bidding mechanism. Fitness is only estimated at the Team population and a diversity metric is used to reduce the likelihood of premature convergence. This work retains the use of fitness sharing as the diversity metric (discussed below). As per the Point population, a fraction of the Team individuals are deterministically replaced at each generation (G_T).

The **Learner population** (L) consists of bid-based GP individuals that may appear in multiple teams [27]. Each learner l_i is defined by an action, $l_i.(a)$, and program, $l_i.(p)$. Algorithm 1 summarizes the process of evaluating each team relative to a Cube configuration. Each learner 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 team populations, the size of the Learner population floats as a function of the mutation operator(s) adding new learners. Moreover, after G_T team individuals are deleted, any learner that does not receive a Team pointer is also deleted. There is no further concept of learner fitness, i.e. task specific fitness is only expressed at the level of the teams.

Note that while the source task is under evaluation there is only one level to a policy, thus Algorithm 1 Step 2.(d) is never called. During target task evaluation a new Point, Team and Learner population are evolved in which learner actions now represent pointers to teams evolved under the source task. In this case, Step 2.(d) is first satisfied resulting in a pointer being passed to the previously evolved team. A second round of learner evaluation then takes place relative to the learners of the previously evolved team. The learners of this team all have atomic actions (one of 12 possible quarter turn twists), thus the 'winning' learner updates the state of the

[4]Teams are initialized with a learner compliment sampled with uniform probability over the interval $[2, ..., \omega]$.

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 3.1) and t is the index denoting the number of twists applied relative to the initial Cube state.

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

Algorithm 2 Breeder style model of evolution adopted by Symbiotic Bid-Based GP.

```
1: procedure Train
2:     t = 0
3:     Initialize point population P^t
4:     Initialize team population T^t (implicitly initializes learner population L^t)
5:     while t ≤ t_max do
6:         Generate G_P new Points and add them to P^t
7:         Generate G_T new Teams and add them to T^t
8:         for all tm_i ∈ T^t do
9:             for all p_k ∈ P^t do
10:                evaluate tm_i on p_k
11:            end for
12:        end for
13:        Rank P^t
14:        Rank T^t
15:        Remove G_P points from P^t
16:        Remove G_T teams from T^t
17:        Remove learners without a team
18:        t = t + 1
19:    end while
20:    return best team in T^t
21: end procedure
```

Cube, Step 2.(c).

The overall evolutionary process assumes a 'breeder' formulation in which G_P points and G_T teams are added at each generation, Steps 6 and 7 of Algorithm 2. Fitness evaluation applies all teams to all points (Steps 8 through 12, Algorithm 2) in order to rank points and teams, after which the worst G_P points and G_T teams are deleted (Steps 15 and 16, Algorithm 2). Any learner not associated with a team are also deleted (resulting in a variable size learner population).

2.6.1 Coevolution

As mentioned above, SBB is based around the concept of coevolution. Under a traditional single-population GP model, a population of learners would act on some environment and a fitness measure would be defined. In the case of SBB, two GP-task interactions are present, or competitive coevolution and co-operative coevolution [12].

The interaction between Point and Team population assumes a Pareto archive formulation for competitive coevolution [27, 6]. This implies that individuals are first marked as dominated or not, with dominated Teams prioritized for replacement. Points are rewarded for distinguishing between Teams [3]. However, the number of non-dominated individuals is generally observed to fill the population, necessitating the use of a secondary measure for ranking individuals, or diversity maintenance, where an (implicit) fitness sharing formulation [32] was assumed in the original formulation of SBB [27]. Thus shared fitness, s_i, of team tm_i takes the form:

$$s_i = \sum_k \left(\frac{G(tm_i, p_k)}{\sum_j G(tm_j, p_k)} \right)^{\alpha} \tag{2.1}$$

where $\alpha = 1$ is the norm and $G(tm_i, p_k)$ is the interaction function returning a task specific distance.

In short, GP deployed without diversity maintenance would eventually maintain a population of teams with very similar characteristics as the best individuals would steadily fill the population with their 'offspring'. SBB enforces diversity maintenance by comparing a team's effectiveness on a particular cube initialization, p_i against the entire team population's performance. If a majority of the teams in the population do well against a particular point in the point population, then an individual team's contribution is weighed less heavily in its fitness calculation. However, if a single

Team V point	p1	p2	p3	fitness
tm1	0	1	0	1
tm2	0	0	1	1
tm3	1	1	0	2

(a) Original outcome vector

Team V point	p1	p2	p3	fitness
tm1	0	0.5	0	0.5
tm2	0	0	1	1
tm3	1	0.5	0	1.5

(b) Outcome vector with fitness sharing

Figure 2.2: Pareto archive of outcomes for three teams tm_i and three points p_i.

team does well against a particular point and the rest of the population does poorly, their fitness is weighed more heavily in its individual fitness calculation. Figure 2.2 provides a simplistic summary of a Pareto archive with and without fitness sharing. Without fitness sharing, team 3 is prioritized, but teams 1 and 2 are indistinguishable. With fitness sharing team 2 is also prioritized.

Many mechanisms are also available for discounting point fitness. In order to properly represent fitness in the context of this work, the standard outcome model had been modified to allow a greater breadth of fitness levels. As will become apparent later (Section 3.1), the Rubik's Cube performance functions are based on minimization, whereas Equation 2.1 assumes maximization. With this in mind, the range of the application performance functions will be reversed using their associated maximums (or worst possible fitness), then normalized to the unit interval. In this work a simple linear weighting is assumed for the fitness sharing function, or Equation 2.1 with $\alpha = 1$.

Co-operative coevolution is achieved through the use of the Symbiotic relationship between Team and Learner populations [13, 27]. Specifically, the variable length representation assumed by the Team population enables evolution to conduct a search for 'good' team content. This is facilitated by the definition assumed for Learners, i.e. programs identify context (the bid) while only the successful learner (from a team) suggests an action at any state. Task decomposition is a function of the interaction between learns within each team, as well as from the diversity maintenance enforced through implicit fitness sharing. Benefits that appear when adopting a co-operative coevolutionary framework include variation operators that only effect the 'module' they were applied to [39]. This clarifies the credit assignment process and enables variation operators to operate on multiple 'levels'. Moreover, modular solutions are

easier to reconfigure under objectives that switch over the course of evolution [18], where this could be a property of the point population or the fitness function.

2.6.2 Code Reuse and Policy Trees

In order to leverage previously learned policies SBB can be redeployed recursively to construct 'policy trees' in a bottom up fashion [5, 22, 20, 19]. Thus, following the first deployment of SBB in which no ultimate solutions need necessarily appear, teams from Phase 1 can be reused by teams from Phase 2 (Figure 2.3). In the Phase 2, a new set of SBB populations (Point, Team, Learner) are initialized and evolution repeated. The only difference from Phase 1 is that actions for each Learner in Phase 2 now take the form of pointers to Teams previously evolved in Phase 1. Thus, the goal of Phase 2 is to evolve the root note for a Policy Tree that determines under what conditions to deploy previously evolved policies. Moreover, the ultimate goal is to produce a Policy Tree that is more than the mere sum of its Phase 1 team compliment.

Evaluation of a Policy Tree is performed top down from the (Phase 2) root node. Thus, evaluating a Phase 2 team, tm_i results in the identification of a single learner with maximum output (Step 2.(b), Algorithm 1). However, unlike Phase 1 evolution, the action of such a learner is now a pointer to a previously evolved team (Step 2.(d), Algorithm 1). Thus, the process of team evaluation is repeated, this time for the Phase 1 team identified by the root team learner (as invoked by the 'GOTO' statement in Algorithm 1). Identifying the learner with maximum output now returns an atomic action (Step 2.(c), Algorithm 1) because Phase 1 learners are always defined in terms of task specific actions.

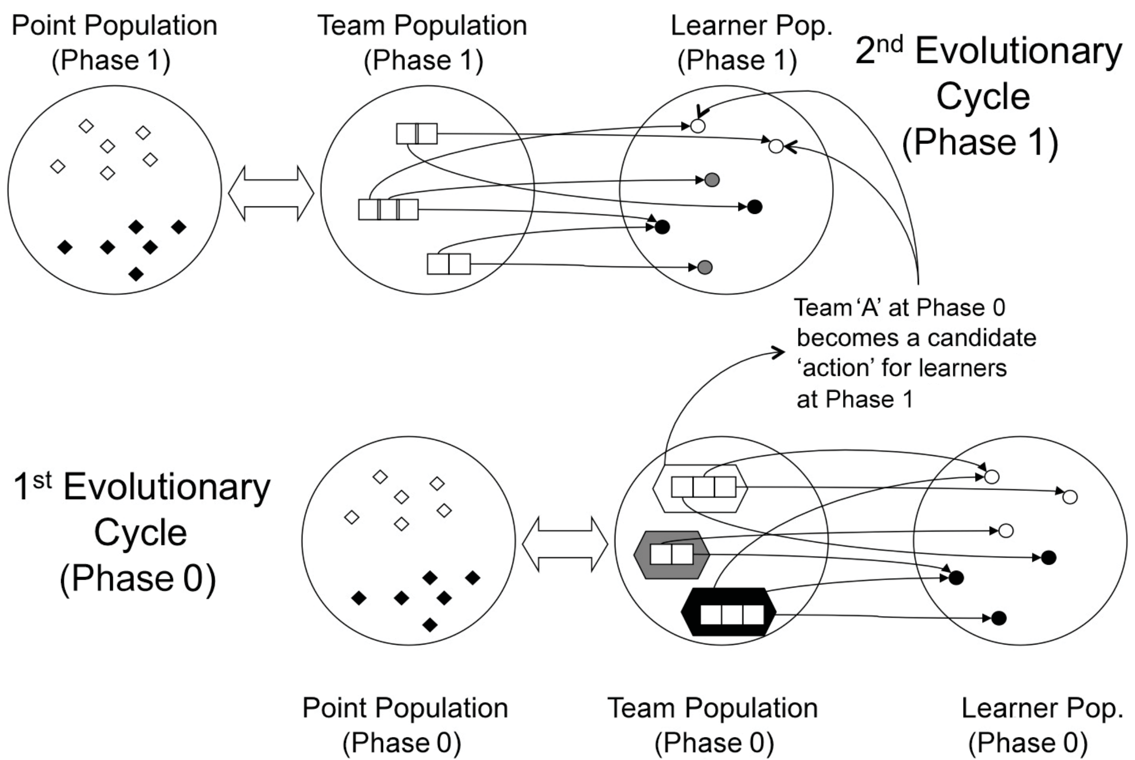


Figure 2.3: Phased architecture for code/policy reuse in SBB. After the first evolutionary cycle has concluded, the Phase 1 team population represent actions for the Phase 2 learner population. Each Phase 2 team represents a candidate switching/root node in a policy tree. Teams evolved during Phase 2 are learning which previous Phase 1 knowledge to reuse in order to successfully accomplish the Phase 2 task.

Chapter 3

Expressing the Rubik's Cube task for Reinforcement Learning

As noted in Section 2.4, El-Sourani et al. identify a sequence of four fitness functions corresponding to the consecutive subgroups associated with Thistlethwaite's Algorithm [7]. Each subgroup represents the incremental identification of invariances appropriate for moving the Cube into the solved state. Given a scrambled Cube, a GA was deployed to find a twist combination that satisfied each subgroup, the solution taking the form of a specific sequence of moves.

However, in limiting themselves to a GA, each Cube start state would require a completely new evolutionary run in order to return a solution, i.e. there was never any generalization to a policy. In this work, I assume a similar approach to the formulation of fitness functions, but with the goal of rewarding the identification of *policies* transforming between consecutive subgroups. In short, in assuming a GP formulation, I am able to evolve policies that generalize to solving multiple scrambled Cubes. Moreover, in assuming SBB in particular, I have a very natural mechanism for incorporating previous policies as evolved against differing goals. Finally, I will also investigate the ability to reduce the number of subgroups actually used, thus being less prescriptive in how to identify invariances.

In summary, SBB will be deployed in two independent phases to build each level of the policy tree under separate objectives, thus synonymous with the task transfer approach to reusing previous policies under different contexts. Moreover, the second phase of evolution needs to successfully identify the relevant policies for reuse / transfer from the first cycle, i.e. a switching policy is used to select between a set of previously evolved policies.

3.1 Formulating fitness for task transfer

The first three subgroups for the Rubik's Cube task under TWA (e.g., [7]) will take the form of two objectives: the source task objective and the target task objective. These two objectives will be considered for fitness in an iterative learning process through which our GP generates policy trees. The base learning run utilizes a five-twist space with the source task acting as a target objective, while the second iteration uses the source task objective as a seed with the target task being subgroup 2. Once these two iterations are complete, I will have policy trees which represent strategies for solving Rubik's Cubes relative to the tasks below.

3.1.1 Subgroup 1 - Source task

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

3.1.2 Subgroup 2 - Target task

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

In order to move the Cube from the Target task to the final solved state, only half twists are necessary. In this work I concentrate on the source and target tasks as defined above as this represents the majority of the search space and constitutes actions defined in terms of quarter twists alone. Assuming that I can solve for the above to tasks, solving for the final objective is much easier and would constitute a Policy specifically evolved for this task alone.

3.1.3 Ideal and Approximate Fitness Functions

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

1. Sample scrambled Cube configurations that conform to the source task.

2. Construct a database of finite depth d exhaustively enumerating moves reaching each sampled instance of the source task, i.e. there are as many database trees as there are source task configurations sampled in step '1'.

3. Extend each database further to identify optimal paths to the Target task for each source task.

Such a database approach obviously limits the number of twists applied to scramble a Cube in order to provide optimal paths between Source and Target tasks. My motivation is to provide a baseline to evaluate the effectiveness of the non-database approach. That is to say, any performance function (other than a database) used to measure the 'distance' to the Source and Target tasks will be an approximation. I want to know what the impact of such an approximation is. In the following I assume a database depth of $d = 10$, which limits the (ideal) path between each subtask to *five* twists. That is to say, in the pathological case, an SBB policy might make five moves that are completely in the *wrong* direction, thus a total of *ten* twists from the desired Cube state. The database(s) need to be able to 'trap' any Cube configuration that SBB policies suggest (relative to a finite sampling of goal states).

In detail, the process assumed for achieving this has the following form:

1. Start with a Rubik's Cube in the final *ultimate* solved state and construct a database consisting of *all* 1 through 10 quarter twist Cube configurations. Such a database consists of $\approx 7.5 \times 10^9$ states [31].

2. Query the database to locate the Cube configurations conforming to Subgroup 1 (source task). Valid solutions to the source task must be to one of these states.

3. Relative to the configurations of the source task (Subgroup 1), query the database to identify all Cube configurations that lie 1 through 5 quarter twists away from

this subgroup. There are over 50 million such (5 twist) configurations alone, which is too large to query during training without significant overhead. Moreover, as the number of twists increases there are increasing numbers of duplicate states. My resulting database consists of $17,675,698$ *unique* Cube configurations that include 1 through 5 quarter twists associated with this subgroup.

4. Target task (subgroup 2) is a further database search for the subset of the $17,675,698$ configurations that satisfy the additional constraint for perfect position of the 12 edges, but with the 8 corners now limited to being in their respective orientations.

5. Relative to the configurations of the target task (Subgroup 2), query the database to identify all Cube configurations that lie 1 through 5 quarter twists away from this subgroup. As subgroup 2 is more constrained, there are a smaller number of such Cube states.

The above process enables us to explicitly identify Cube states that satisfy subgroup 1 and 2 given a five twist limit from the respective goal states. An ideal fitness function would therefore be the count for the number of twists away from subgroup 1 and 2. In general this is would not be feasible, however, it does provide a baseline for what the ideal fitness function performance would be. Hereafter, such a fitness function is referred to as the **ideal fitness function**. That said, the implementation of such an ideal fitness function in practice is not trivial if the implementation is to be efficient. The Appendix summarizes the approach used to design such a database using a Hash map in Java, thus native to the SBB implementation (also built by the author).

A second formulation for fitness is defined in terms of the count of the number of edges and / or corners that fail to match each subgroup [7]. In the case of the Source Task (Section 3.1.1), I adopted the following:

$$G_{source}(tm_i, p_k) = \min_{t=0,\ldots,T}(10 \times o_{edge}|tm_i, p_k, t) \tag{3.1}$$

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

In the case of my Target Task (Section 3.1.2), I adopted the following:

$$G_{target}(tm_i, p_k) = \min_{t=0,\ldots,T}(40 \times o_{corner} + 10 \times p_{edge} | tm_i, p_k, t) \quad (3.2)$$

where o_{corner} is the count for the number of corners that are not oriented and p_{edge} is the count of the number of edges that are positioned incorrectly. Likewise, fitness is expressed relative to an initial Cube configuration, p_k, and up to a maximum of $T(= 5)$ moves identified interactively by candidate policy tm_i. Hereafter, fitness based on G_{source} and G_{target} are denoted the **approximate fitness function**.[1] Needless to say, this latter fitness function does not require database queries, but is unable to provide as much quantification of state as the ideal fitness function. In order to adhere to the fitness sharing methodology presented in Equation 2.1, these values are inverted (turning this task into a fitness maximization task to better fit the SBB fitness structure) and then normalized across a unit interval (to better support SBB's inherent outcome storing and fitness sharing system).

3.2 Representing the Rubik's Cube

The Rubik's Cube represented in this thesis is a $3 \times 3 \times 3$ Rubik's Cube. The cube consists of 6 sides, each of which is represented by a 3×3 grid of 9 colours. Of the 9 colours on each side, the centre grid space never changes and thus defines that side's colour. The individual sides may be rotated left or right. This thesis assumes the quarter-twist metric whereby a single action may be a 90 degree rotation to the left or the right. From an implementation standpoint we have foregone the standard Singmaster notation for the Rubik's Cube group and instead represent sides as $x \in \{B, G, O, R, Y, W\}$ and rotation actions of left and right by 90 degrees as $y \in \{L, R\}$. Thus we can represent each atomic action as an xy tuple, the full set of which can be found in Table 3.1.

The state of a Rubik's Cube is expressed as a vector of 54 integers. Each integer expresses the facelet colour at a specific location. The relationship between an unfolded $3 \times 3 \times 3$ Cube and the vector is summarized by Figure 3.1. Note that cells

[1] As noted in Section 2.6.1, Equations 3.1 and 3.2 are subject to a linear transform in order to re-express them as a minimization objective.

Table 3.1: The Rubik's Cube group is defined as $(G, \cdot)$ where G represents the set of all possible actions which may be applied to the cube and the $\cdot$ operator represents a concatenation of those actions.

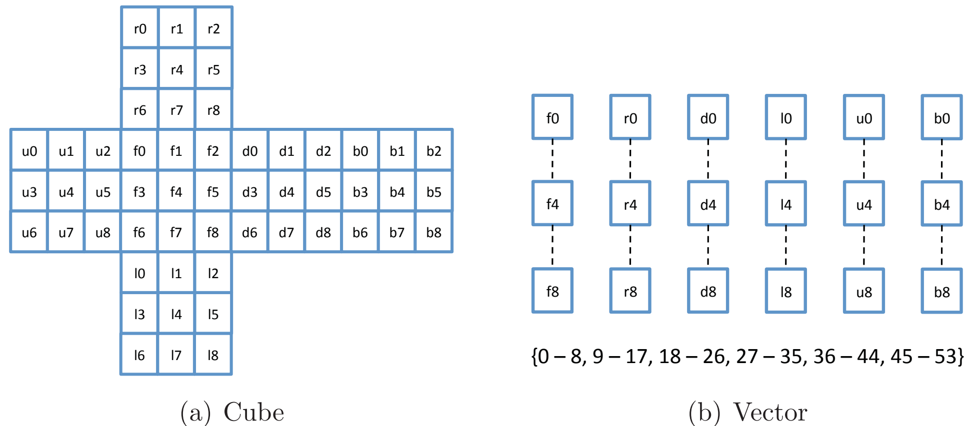
Counterclockwise 90°	Clockwise 90°
BL (blue face twisted left)	BR (blue face twisted right)
GL (green face twisted left)	GR (green face twisted right)
OL (orange face twisted left)	OR (orange face twisted right)
RL (red face twisted left)	RR (red face twisted right)
WL (white face twisted left)	WR (white face twisted right)
YL (yellow face twisted left)	YR (yellow face twisted right)

corresponding to $\{f4, r4, d4, ..., b4\}$ denote the centre facelet for each face and therefore the integer indicating their respective colour never changes. Such information is retained in order to provide the basis for GP to establish the correct colour for each face. Note also that this information essentially represents the state of the game in a form similar to that experienced by a human, i.e. facelets. Conversely, applying machine learning to gaming tasks often requires considerable effort to identify game specific attributes/features/sensors that encode useful properties of the game.[2]

3.3 Policy tree structure

Policy trees are generated through through the two Phase model as shown in Figure 2.3. The Phase 1 SBB individuals are evolved relative to the Source task. Once the individual teams have completed training they will cumulatively understand how to solve a certain percentage of the Source task cubes. From previous results, I have observed that the top 20 teams will typically be capable of solving a majority of the cubes and thus I assume those teams to be the actions for Phase 2, evolving root nodes solving the Target task. At this second phase, SBB leverages the previous knowledge obtained in the first phase by learning which previous team to query for a solution. My insight is that, as Source and Target tasks are related, then SBB should also be able to identify Cube state and relate it to the appropriate source policies while generalizing this to a Policy tree that solves both Source and Target task.

[2]See for example, the state representation used for GP under the 'Rush Hour' puzzle [11], or sensors required for Monte Carlo Tree Search as applied to Ms. Pac-Man behaviours [30].



(a) Cube

(b) Vector

Figure 3.1: 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.

This method of policy tree construction will create trees with a depth, D, equal to the number of learning tasks being performed, which is 2 in the case of this work. Additionally, the number of leaf nodes in a single policy tree is equal to ω^D. Regarding these experiments, ω is a parameter which represents the maximum number of learners a team is allowed to contain. For the experiments in this thesis, an ω value of 9 was used. My insight here is that although there are 12 atomic actions available, Phase 1 policies will appear that only solve subsets of the Source task. That is to say, teams at Phase 1 will solve the Source task for different subsets of initial (scrambled) Cube start states, hence any single team does not need to represent *all* atomic actions. There is no need to predefine which subset of up to 9 atomic actions each team should to use, this is also an evolved property. During Phase 2, teams are again free to determine what combination of the 20 best teams from Phase 1 to index. Given that some subset of the Phase 1 teams will collectively cover the entire set of 12 atomic actions, adopting $\omega = 9$ is now sufficient for building Policy trees that cover the entire set of atomic actions.

Chapter 4

Evaluation Methodology

In the following, specific parameterizations for SBB are summarized. Moreover, the SBB framework for evolving policy trees assumes several inter-related components (Section 2.6) including: Policy diversity maintenance, Point fitness (competitive co-evolution). My evaluation will also assess their respective contributions by explicitly turning each component off and comparing the resulting performance to the fully specified system.

4.1 Parameterization

The publicly available jSBB distribution provided the code base from which modifications were made to support policy tree discovery under the Rubik's Cube task.[1] The parameters for SBB are summarized in Table 4.1. Note that the gap size is defined such that 100 individuals in the Team and Point population are common to sequential generations, and a lower number replaced per generation ($G_P = 50$ and $G_T = 20$ for Point and Team populations respectively). Given that there are $50,000$ generations performed, then the total number of unique Cube configurations encountered during training is $2,500,100$.[2] This means that at after the first generation, the 20 *new* teams are evaluated on all 150 Cube configurations, whereas 100 Teams already have fitness evaluated on 100 Points, thus this subset of Teams only need evaluation over the 50 new Points. This corresponds to $8,000$ evaluations per generation or 4×10^8 per training cycle.

After training is complete for both source and target task, all possible 5 twist configurations conforming to the union of the subgroups are used for test purposes or $17,675,698$ (of which 14% were encountered *once* during training).[3] My motivation

[1] `http://web.cs.dal.ca/~mheywood/Code/`

[2] The same subset of Cube configurations are employed for all runs.

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

Table 4.1: Generic SBB parameters. t_{max} generations are performed for each task or $2 \times t_{max}$ generations in total. Team specific variation operators P_D, P_A pertain to the probability of deleting or adding a learner to the current team. Learner specific variation operators P_m, P_s, P_d, P_a pertain to the probability of mutating an instruction field, swapping a pair of instructions, and deleting or adding an instruction respectively.

Parameter	Value
Max. Learners per team (ω)	9
Population size (H_{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
Learners	
Max. Num. instructions	64
P_m, P_s, P_d, P_a	0.1

for revisiting 'training' configurations during test is that there is only a $\frac{1}{2}$ chance of a Cube configuration being retained *per generation*. This compounds to a $\frac{1}{2}^t$ chance of survival as measured per generation t.

There are a total of eight operators as utilized (by learners). This is unchanged from previous work, i.e. $\{+, -, \times, \div, \cos, \exp, \log, \text{cond}\}$ where 'cond' is a conditional operator that switches the sign of a operand if taken [27, 6], i.e. no attempt is made to craft task specific operators. Each policy is allowed a maximum budget of 5 twists to solve a Cube configuration.

4.2 Qualifying experimentation

In addition to the main experimental task, this thesis describes alternative techniques for determining the effects of various algorithm modifications in an attempt to qualify SBB-related features. The alternative SBB parameterizations are defined and discussed in this section.

4.2.1 Disabling Policy Diversity

SBB uses (implicit) fitness sharing to maintain diversity in the policies discovered by teams (Section 2.6.1). The motivation was to ensure that different teams learn

to solve different aspects of a task; hence, the Team population does not favour the reproduction of the single 'fittest' team. In disabling fitness sharing I anticipate that Phase 1 evolution will fail to provide a sufficiently diverse range of policies for solving the Source task. This in turn will limit the ability of Phase 2 to generalize to solving multiple Cube configurations. With this functional change, the new equation for individual team fitness becomes (compare to Eq. (2.1)):

$$s_i = \sum_k G(tm_i, p_k) \tag{4.1}$$

4.2.2 Random Selection of Points

SBB also utilizes competitive co-evolution between data points and teams in order to scale to a task with large cardinality (Section 2.6.1). That is to say, it is too expensive to conduct fitness evaluation against all possible training configurations. Instead, competitive coevolution is employed to identify 'more informative' Cube configurations while simultaneously introducing previously unseen ones. When operating under normal conditions SBB will evolve the point population based on how well the team population evaluated: points which are too difficult or too easy are removed in order to promote learning among the team population. This mechanism prevents teams from entering a state of stunting learning and prevents any teams from overfitting, effectively forcing the team population to learn on points which are always challenging, but never inhibitive.

The alternative experiment performed for the purpose of measuring the contribution of competitive co-evolution selects points for replacement completely at random. Thus, at each generation G_P points are replaced with uniform probability (no concept of 'point fitness').

Chapter 5

Results

5.1 Standard 5 Twist Model

Figure 5.1 summarizes the performance of the resulting policy on the subgroup 2 – target task (Eq. (3.2)) – in terms of both the average individual-wise and the population-wise performance. Specifically, after the last generation on the target task, all individuals from the population are evaluated across all $17,675,698$ Cube configurations. We then rank each team in descending order of the number Cube configurations solved. This provides a monotonically decreasing curve summarizing the strength of individuals in the population after evolution. However, population diversity implies that individuals do not necessarily solve the same subset of Cube configurations. Thus, relative to the individual-wise ordering of teams, we also build a count for the total number of unique Cube configurations solved, or the monotonically ascending curve in Figure 5.1. The third curve represents a population of random policies (content at generation zero).

Figure 5.2 presents the individual-wise and cumulative curves for Ideal fitness function and the Approximate fitness function in more detail. Under the Ideal fitness function the cumulative performance is such that all 5 runs resulted in 100% of the test cases being solved by some subset of the teams. The Approximate fitness function approaches this, with three of the 5 runs solving 100% of the test cases and the two remaining cases solving over 96% of the test cases (across the population of teams). In short, the Approximate fitness function is able to approximate the Ideal fitness function sufficiently well, while being feasible to deploy in practice, i.e. the Ideal fitness function requires an exhaustive construction of multiple databases of twist states, thus considerable prior effort to efficiently construct.

Figure 5.3 summarizes the architecture of a solution post training (corresponds to one of the first ranked policies on the target task). Team 'H1' is the switching team evolved during the target task and consists of four learners. Each H1 learner

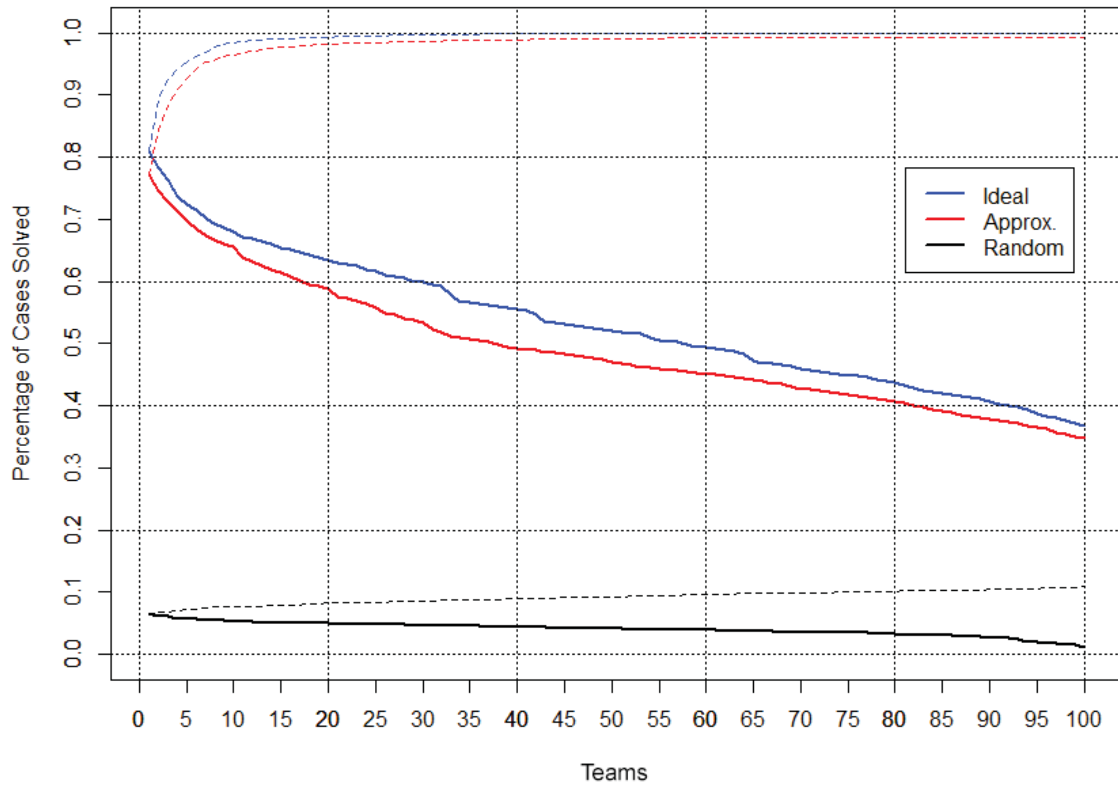
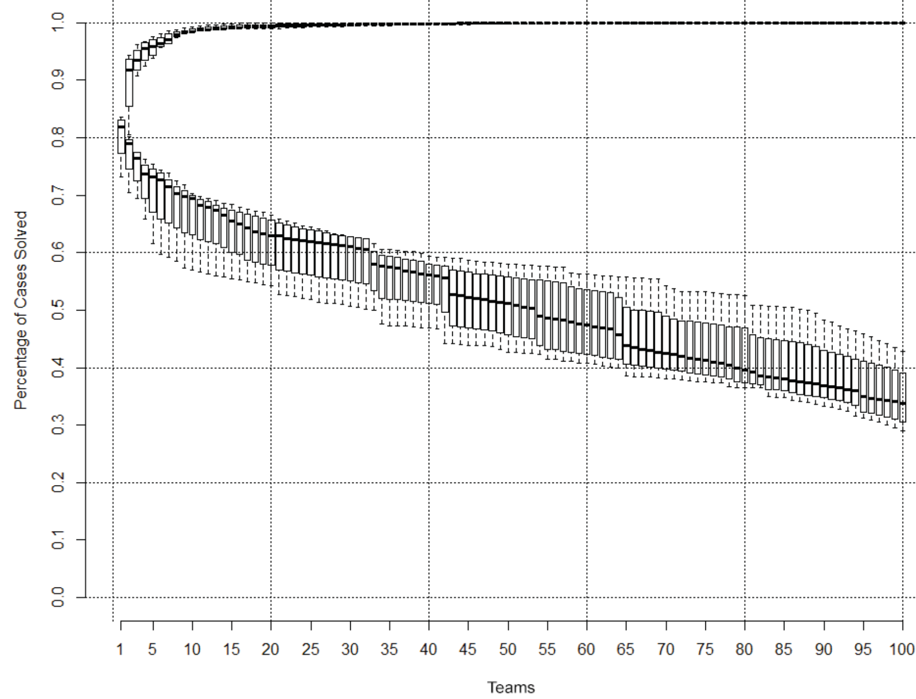
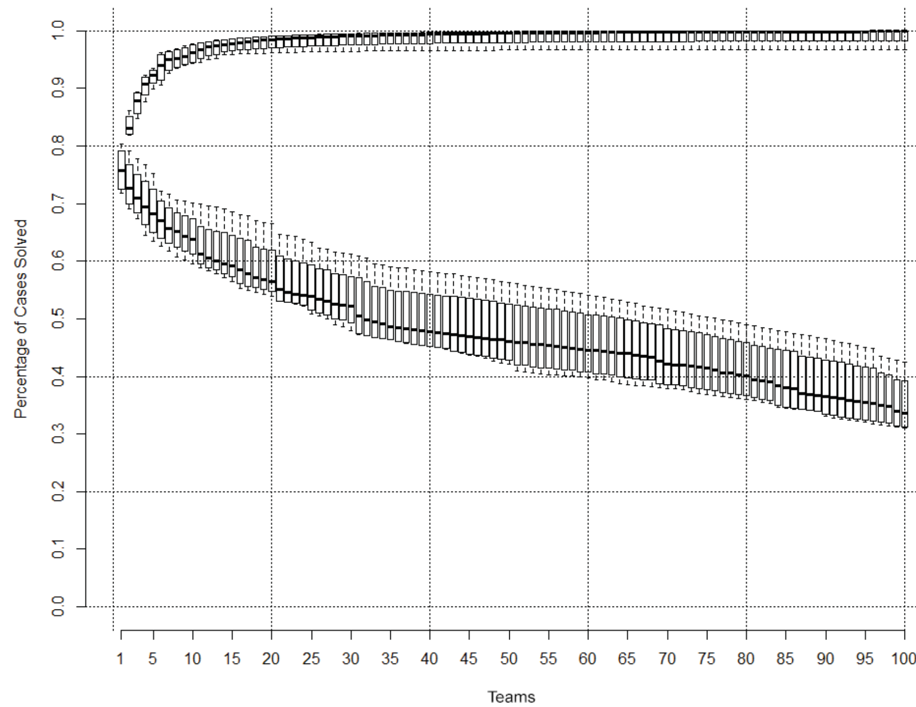


Figure 5.1: Average number of Cube configurations solved at subgroup 2 (target task) by SBB. Descending curves (solid) represent average individual-wise performance. Ascending curves (dashed) represent cumulative performance. The y-axis represents the percent of $17,675,698$ unique scrambled Cube configurations solved.



(a) Ideal fitness function



(b) Approximate fitness function

Figure 5.2: Percent of 17, 675, 698 Cube configurations solved at the Target subgroup. Individual-wise ranking (descending) and cumulative ranking (ascending). Distribution reflects the variation across 5 different runs per experiment.

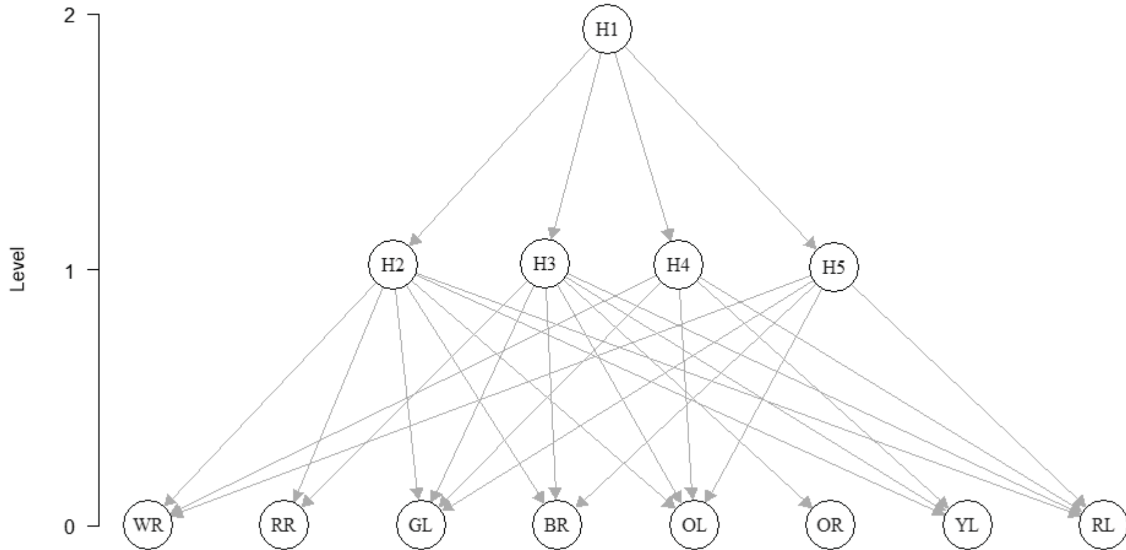


Figure 5.3: Policy tree solving 80% of the Cube configurations under the Target task. Level 0 nodes represent atomic actions. Level 1 nodes represent teams indexed as actions by learners from the single phase (level) 2 team. Each atomic action is defined by an xy tuple in which $x \in \{B, G, O, R, Y, W\}$ denote one of six colour Cube faces, and $y \in \{L, R\}$ denote left (counter clockwise) or right (clockwise) quarter turns.

uses a unique action in the form of a reference to a team evolved during the source task (labelled H2 through H5). H1 is therefore the 'switching' node for the overall policy tree. Given the current state of the Cube, the four H1 learners execute their programs to resolve which branch to take, i.e. identifies *one* phase 1 team (H2 through H5). This implies that only learners associated with one team at each phase are ever executed *per state* in order to determine an action, *not* the entire tree. The selected phase 1 team executes its learner programs (on the same Cube state as the switching team), so identifying a specific atomic action (level 0). In this particular case, the (phase 1) Teams typically employ 5 to 7 learners. Out of a total of 12 available atomic actions, 8 are actually used by this particular individual.

5.2 Disabling Policy Diversity

Figures 5.4 and 5.5 summarize the percentage of scrambled Cube configurations solved when (implicit) fitness sharing was disabled (Section 4.2.1). In short, teams typically performed worse individually by up to $\approx 15\%$ and cumulatively by up to $\approx 30\%$. This

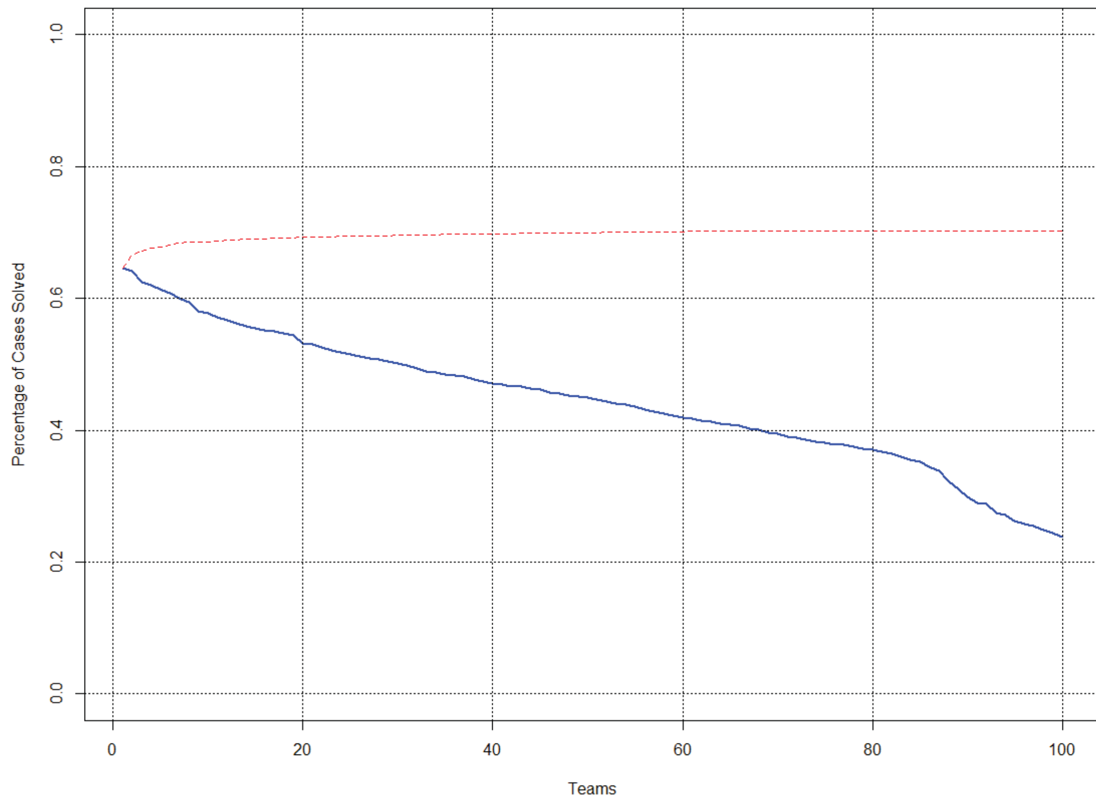


Figure 5.4: Mean solution rate for five team populations across a test set against a 2nd subgroup target task without diversity maintenance. Individual-wise ranking with an average best team solving approximately 64% of all cases.

provides an excellent argument for maintaining Policy diversity in the SBB algorithm.

The flat slop of the cumulative curve indicates that teams are effectively doing the same thing and the population has therefore been dominated by a single phenotypic trait. Thus, not only is the single fittest team much weaker under no diversity maintenance, but the population as a whole has collapsed. In short, none of the possible advantages of assuming a population based framework are evident. Naturally, any advantage that conducting evolution over multiple phases is also largely lost as there is little in the way of different behaviours from Phase 1 that can be leveraged into Phase 2.

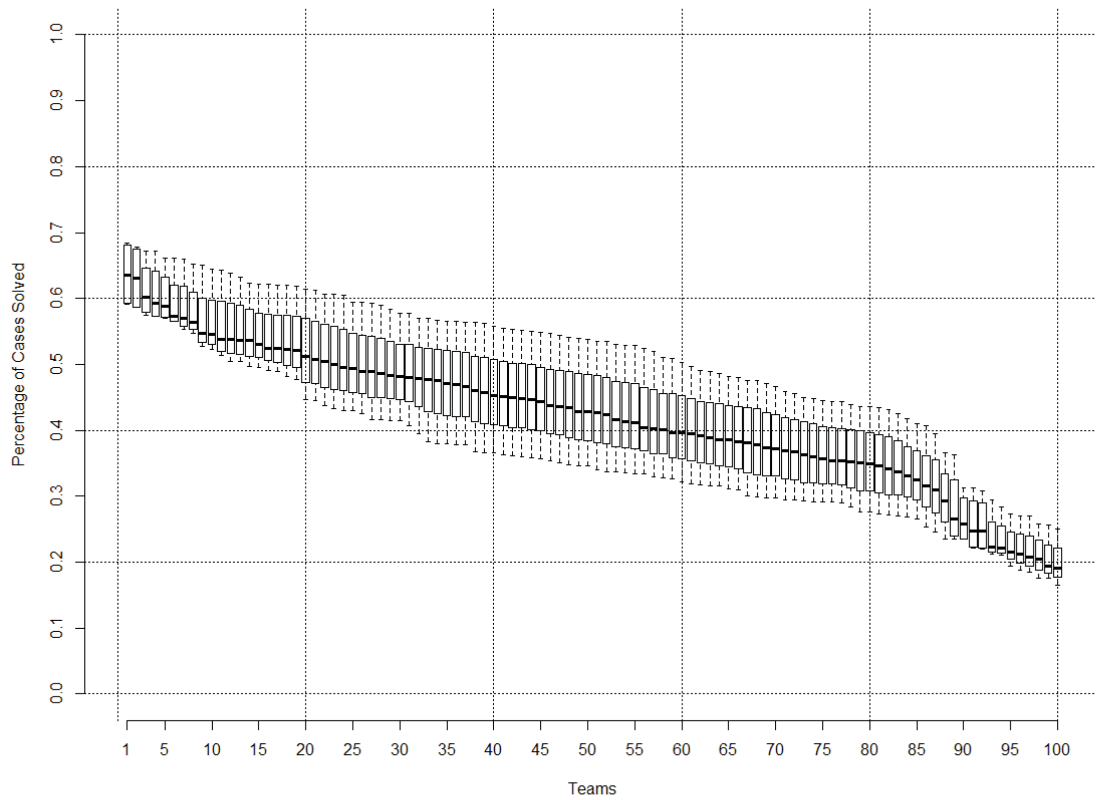


Figure 5.5: Distribution of solution rates for five team populations across a test set against a 2nd subgroup target task without diversity maintenance. Individual-wise ranking with the median best team solving approximately 64% of all cases.

5.3 Random Selection of Points

The concept of random selection in this experiment implied that, rather than prioritizing points for replacement based on the relative challenge that the individual points present, random selection would instead have us simply remove G_P random individuals from the point population and then insert the same number of randomly chosen new points (Section 4.2.2).

Figures 5.6 and 5.7 summarize the effects of random point selection. Teams perform very poorly with a tendency to only solve $\approx 25\%$ more than a static sampling of policies (no evolution).[1] Without the ability to retain Points (Cube configurations) that identify one Team as more effective than another, Teams are effectively being replaced at random, reducing the search process to little more than a random walk.

In short, both Policy diversity maintenance *and* retention of useful training scenarios are necessary to successfully guide evolution. Competitive coevolution already has several pathologies (forgetting, disengagement). Random replacement of useful training scenarios potentially magnifies these effects. Likewise, without Policy diversity we are left with limited opportunity to reuse policies from an earlier phase of evolution.

5.4 Phasic task generalization

Based on the conclusions drawn about the main 2-phase task solutions, it was suggested that due to the structure of the problem space it should be possible for SBB to build its policy tree knowledge base from the same task (rather than iterative tasks) during each phase. Since SBB leverages diversity in the team population we know that the amount of overlap in solutions provided by a final team population should be minimized. Our previous results show that the the whole team population is capable of solving a very large number of Cube states ($\approx 98\%$) after the twenty best teams are included, thus we can conclude that subsequent phases of evolution will allow us to take advantage of that knowledge by creating a mechanism for choosing which of the teams to use to solve a given task. Thus the initial phase allows the teams to learn how to solve nearly all cubes and the second phase allows a new population of

[1] The solution rate of a random population can be seen in Figure 5.1.

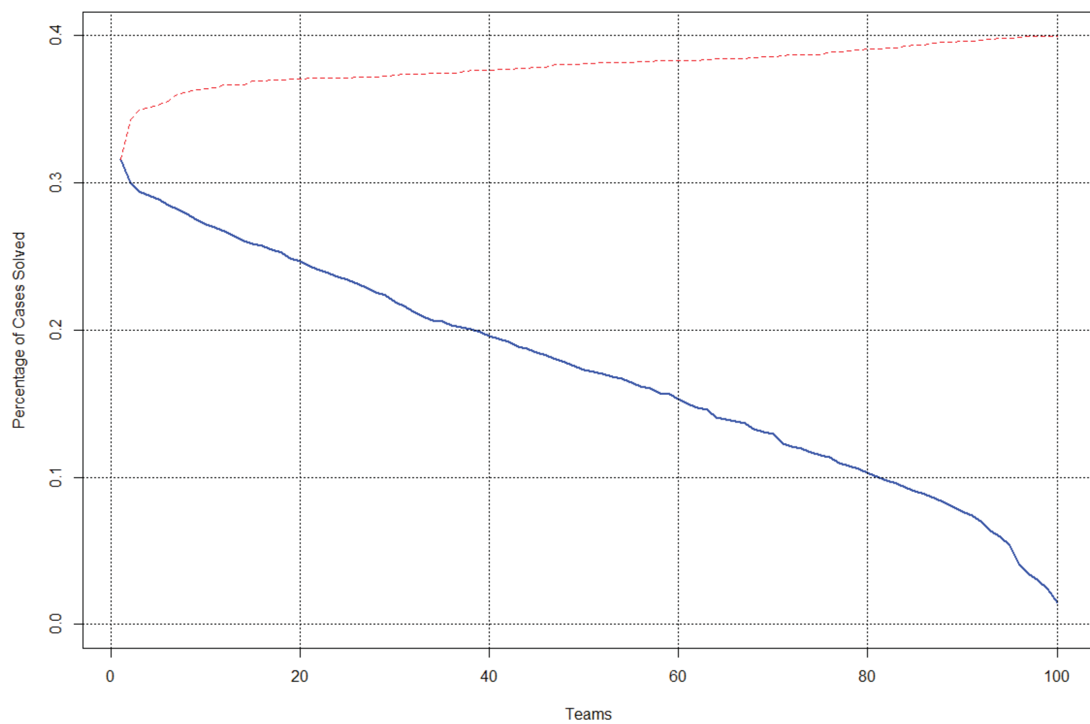


Figure 5.6: Mean solution rate for five team populations across a test set against a 2nd subgroup target task using random point selection. Individual-wise ranking (descending) and mean cumulative ranking (ascending) with an average best team solving approximately 32% of all cases.

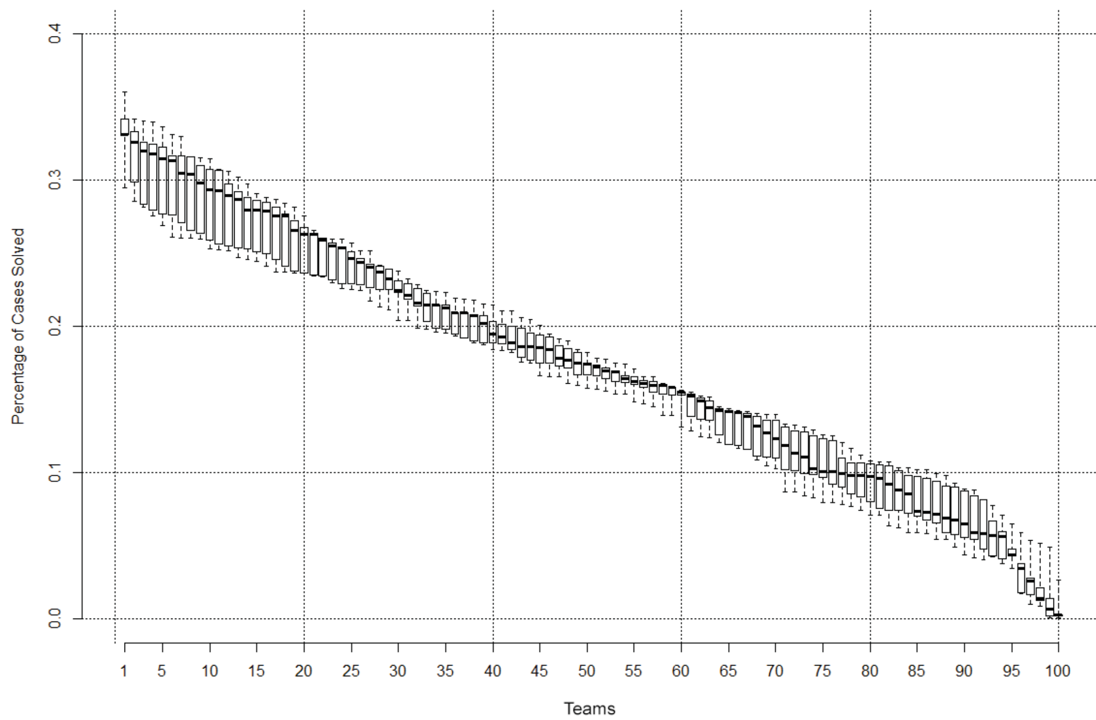


Figure 5.7: Distribution of solution rates for five team populations across a test set against a 2nd subgroup target task using random point selection. Individual-wise ranking with a median best team solving approximately 33% of all cases.

teams to match Cubes to previous understanding.

In Figure 5.8 we can see the effects of phasic task generalization. In these experiments the phase 2 results show a similar learning trend to the approximate fitness function under the source/target subgroup tasks as presented above. This figure shows the distribution of individual team outcomes with a best team achieving $\approx 83\%$ solve rate. The team populations as a whole contain less effective 'worst' individuals and their cumulative fitness does not grow as quickly when compared to the previous experiments. However, there are only 5 repetitions in each case, so this is likely to be due to the underlying stochastic artifacts, e.g. drift. In short, it appears that it is possible to construct equally effective solutions using the target objective alone. Thus, potentially making it easier to apply to solving Cubes scrambled with more than 5 twists as we will no longer need to first build solutions for all twists under the source objective and then repeat for the source objective, i.e. in order to scale to an arbitrary number of twists I would anticipate requiring more than one phase of evolution.

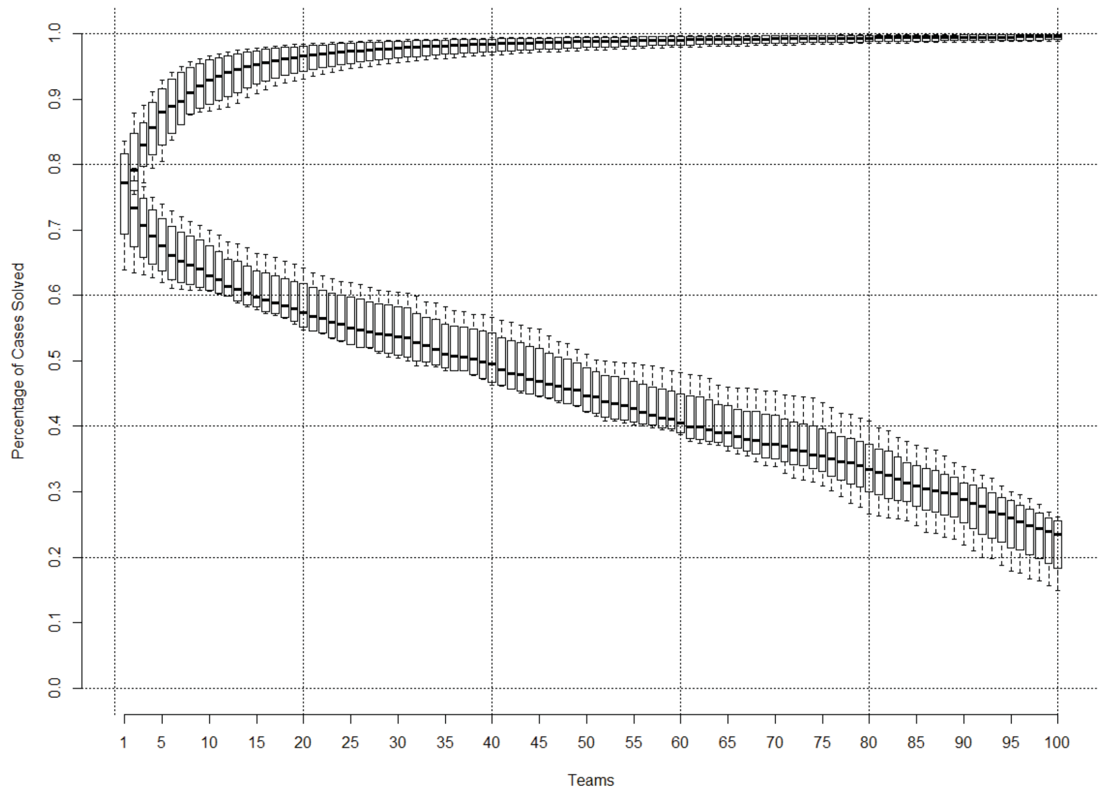


Figure 5.8: Phasic task generalization. Distribution of fitness for five team populations across a test set against a 2nd subgroup target task using the target task as a goal for 2-phase populations. Individual-wise ranking (descending) and cumulative ranking (ascending) with an average best team solving approximately 78% of available cases.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Machine learning approaches for solving the Rubik's Cube have to date been limited to performing a search conducted relative to a specific scrambled Cube configuration, i.e. solutions are *not* capable of solving multiple Cube configurations. This work builds on two recent developments in order to provide a starting point for evolving policies that solve *multiple* configurations of the Rubik's Cube. The first development is the formulation of an approximate fitness function that corresponds to the subgroups of Thistlethwaite's Algorithm. The second provides a framework for the piece-wise hierarchical construction of GP individuals under policy search or SBB. The combination of the two leads to knowledge transfer between policies associated with consecutive subgroups. In this work we demonstrate the feasibility of this scheme for two subgroups that span the first three (of four) subgroups associated with Thistlethwaite's Algorithm. We are able to show that the approximate fitness function provides a very close approximation to performance of an ideal fitness function based on an exhaustive database of Cube configurations, albeit limited to 1 through 5 twists away from the subtask and sampled from the subset of Cube states up to 10 twists away from the solved Cube. Naturally the latter is too expensive to deploy in practice, but can be used under the restricted task considered here (policies subjected to no more than 5 twists).

In addition to the results found relating the ideal fitness function to the approximate fitness function, we can also show the affects of diversity as they relate to this implementation of the Rubik's Cube domain. We are able to show that Policy diversity has a positive net effect on both individual and population solution rates through experiments in which reduced diversity decreased both measures of success. In addition, competitive co-evolution has also been identified as a major contributor to the accuracy of teams as they are given the appropriate amount of time to create

policies for specific cube outcomes, while simultaneously not teaching them redundant information.

In the case of phasic task generalization, we see that the way we utilize underlying knowledge can impact the outcomes in future layers. These experiments show that we can iteratively build policy trees by beginning with a complex task and allowing the algorithm to improve systematically through the use of evolutionary phases. Assuming our initial phases contain team populations capable of solving a majority of the test cases collectively, but adding subsequent evolutionary phases we can then learn how to best use the specialists in the population in order to achieve a better single team fitness in future evolutionary cycles.

6.2 Future Work

The work reported here is limited to a 5 twists in order to facilitate comparison to an ideal fitness function. Given the results of the comparison of the two methods we can say that the approximate fitness function in this case is accurate enough in a 5 twist model to attempt larger numbers of twists and thus more varied configurations of Rubik's Cubes. In this section we discuss some future experiment foundations which could allow us to potentially solve 5 twist Rubik's Cube configurations and expand into even more twist space.

6.2.1 5 Twist Completion

Based on the previous work by El-Sourani et al. [7] we know that once cubes have reached a state satisfying our target task (edge position/orientation and corner orientation) then the complete number of unique cube configurations relating target task to the solved Cube is $\approx 663,000$.[1] This is a much smaller search space than currently addressed and can be solved using half-twists alone. A single Policy tree would likely be sufficient to solve this part of the task.

[1] See discussion of the $|G_3|$ state space in [7].

6.2.2 Twist Expansion

Given the success achieved under a 5 twist model, extending the framework to scrambled Cubes with more than 5 twists is also possible under the approximate fitness function. Since we would no longer need the database for the ideal function due to the accuracy of the approximate fitness function, we would no longer be burdened with overhead from database generation and queries.

Twist expansion can be approached in several ways. First, I could simply allow the program to run from the beginning using a new parameterization. This method would allow us to statically assign the boundaries of the optimization task similar to the experiments presented above. This would yield comparable results to the current work and would likely provide insight into the limitations of operating under an N-twist model. An alternative way of attempting to solve higher twists is to add an iterative phasing function to the overall policy tree generation approach. In such a design space I would allow the algorithm to proceed with a 5 twist space and decide based on fitness measurement (either single best team or cumulative fitness score) when to reorganize with higher twist cube states and greater action budget (that is, allowing the program to perform 6 twists instead of 5). This style of iterative learning could provide insight into the type of Rubik's Cube solution algorithms SBB is creating over time.

The foundation of human Rubik's Cube solving is based around the formulation of cubie-shifting algorithms and the cascading solution process whereby a human solves one side, then the equator, then the opposite side of the first. To perform this solution method humans use a series of algorithms (often written in Singmaster notation) designed to move and rotate individual cubies without changing the previously completed sections of the cube. Since our current work only describes cubes within a 5 twist model, the goal of twist expansion would allow us to better examine general solutions and determine whether or not SBB is creating algorithms similar to those used by humans to solve cubes from a variety of orientations. Since all twists are performed first by rotation a particular side, we could standardize the side and convert our current notation into Singmaster notation for the purpose of understanding the patterns being generated by the policy tree.

6.2.3 Complexification of Policy Trees

Another technique which SBB can attempt to execute is iterative policy tree generation against a single task. Rather than continue to expand into new task domains as the current work shows, we can allow SBB to continue making additional policy tree levels against a single task in order to refine the decision-making process and improve fitness through exploitation of previous knowledge.

Under this policy, we could provide a fitness goal and allow SBB to attempt to achieve it. Since we currently have a best team solving approximately 80% of the target task states based on refining previous knowledge, I hypothesize that the number of solutions for a best individual will increase in subsequent executions.

This would work by taking advantage of the diversity of the team population. My results for a single random initial SBB population indicate that the best individual is capable of solving $\approx 6.5\%$ of the 17 million test Cube configurations. Moreover, this is as measured relative to the target task. In short, we already have some basis for engagement which under fitness directed selection could conceivably imply that repeating the process relative to the target task alone is sufficient for discovering the necessary Policy tree.

6.2.4 Rubik's Cube as a reinforcement learning benchmark

Section 2 noted that the AI community has only considered finding solutions to the Rubik's Cube task using different forms of search (IDA* in particular). Conversely, from a reinforcement learning (RL) perspective there are no previous results. This is surprising because the task has many properties that could potentially be discovered by RL algorithms. At the time of writing Stephen Kelly had attempted to deploy the well known Sarsa credit assignment scheme with a Radial Basis Function (RBF) neural network representation to the source task [34]. The Sarsa–RBF combination failed to converge for the source task, without which solving for the target task would not be possible.

Some of this is likely a function of different representational biases, i.e. the RBF tiling explicitly indexes all states, whereas GP does not. This might provide more robustness to state changes that result in $\approx 37\%$ of the state variable values changing.

Alternatively, neuroevolutionary schemes might be adopted that have been demonstrated to match the performance of deep learning while utilizing a fraction of the model complexity [9].

Appendix A – Constructing the 10 twist database

The SBB distribution assumed in this work takes the form of jSBB, thus a Java implementation (also developed by the author). Given that the efficiency of the fitness evaluation will have most impact on the run-time of the ideal fitness function, I also implement the database in Java in order to achieve as tightly a coupled pairing as possible (no socket interface). The database implementation was created in Java (the language in which SBB is also, which required the consideration of several important factors were judged to have a direct impact on the quality of the resulting database.

First, Java's default hash functionality is limited to integers. This means my hash period would be limited to $2^3$2 different values, which is obviously not sufficient for any large twist count. In order to mitigate this, we instead moved to a hash function working entirely with long integers, which gives us a period of $2^6$4 which is almost sufficient for handling the largest state value by twist. Since the Java hash functions are built in (and thus immutable), we were then required to handle hashing separately from the default implementation by creating custom methods.

Second, a list implementation might be based purely on if-based comparisons and linear searches. Instead of relying on lists, I would instead use hashed maps to mitigate the need for constant linear searching. Additionally, maps would give us the ability to remove duplicates with very few conditionals, where the identification of duplicates represents an increasing cost as the size of the database increases.

In order to minimize the amount of conditional-related computational overhead in the program, I used two separate hashing algorithms: one for cube faces and one for cubes as a whole. These two algorithms perform different tasks in order to maximize the amount of time spent creating cubes and minimize the amount of time checking for equivalency.

The hashing algorithm used for the cube faces was not formulaic. Instead, we note that there are six faces on a Rubik's Cube and each face has nine individual colours. Since there are six faces, there will be six colours total and thus we need three bits to represent all colours. If we know there are nine colours per face and

each is represented by three bits, we can see that we simply need $3 \times 9 = 27$ bits to uniquely hash every single possible face combination. This hash can still be stored in an integer, which increases the speed of comparison and each face is therefore uniquely hashed.

The second hashing algorithm is a very simple prime-additive method similar to the one typically used for hashing strings. Since there are far more possible states for cubes than faces, we did not attempt to maximize the bit combinations in our hash. Instead, we simply supplied a greater range of values and left the hash function as normal. It should be noted that both the faces and cubes only calculate a new hash if the object state has changed since the last hash. This minimizes the amount of time spent calculating integers and longs in favour of occasionally flipping bit flags.

These two hashes are important for the hash map, which is used both when constructing a database and calculating fitness later. When a hash map is implemented, it uses the hashes of the cubes to determine where to store the cubes in memory and uses the following rules for determining which cubes should be stored and how:

- If the hash provides a value with no currently stored cube, we store the cube in this location. This is an $O(1)$ operation.

- If the hash provides a value and at least one cube is stored in this location, we compare the new cube against all the cubes stores in this hash index using an 'equals()' method. If we find that the new cube is equal to a cube already stored here, we discard the new cube. If there is no such match, we store the new cube at the end of the stored linked list. This is potentially an $O(n)$ operation (where n is the total number of cube states), but in practice we only starting getting hash collisions at approximately 10.5 million cubes, thus making this step much less time-consuming in a practical sense.

Bibliography

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

[2] J. Bongard. Behavior chaining – Incremental behaviour integration for evolutionary robotics. In *Proceedings of the International Conference on the Synthesis and Simulation of Living Systems*. MIT, 2008.

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

[4] E. D. Demaine, M. L. Demaine, S. Eisenstat, A. Lubiw, and A. Winslow. Algorithms for solving Rubik's Cubes. In *ESA*, volume 6942 of *LNCS*, pages 589–700, 2011.

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

[6] J. A. Doucette, P. Lichodzijewski, and M. I. Heywood. Symbiotic coevolutionary genetic programming. *Genetic Programming and Evolvable Machines*, 13:71–101, 2012.

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

[8] R. Miikkulainen F. Gomez, J. Schmidhuber. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[9] C. Fernando, D. Banarse, M. Reynolds, R. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra. Convolution by evolution – differentiable pattern producing networks. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 109–116, 2016.

[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 *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 955–962, 2009.

[12] M. I. Heywood and K. Krawiec. Solving complex problems with coevolutionary algorithms. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (Companion)*, pages 547–573, 2015.

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

[14] Alexander Irpan. Exploring boosted neural nets for rubiks cube solving. As of this writing, paper may be found at `http://www.alexirpan.com/public/research/nips_2016.pdf`, 2016.

[15] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artif. Intell.*, 175:2075–2098, 2011.

[16] W. Jaskowski, M. Szubert, P. Liskowski, and K. Krawiec. High-dimensional function approximation for knowledge-free reinforcement learning: A case study in SZ-Tetris. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 567–574, 2015.

[17] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Machine Learning Research*, 4:237–285, 1996.

[18] N. Kashtan, E. Noor, and U. Alon. Varying environments can speed up evolution. *Proceedings of the National Academy of Sciences*, 104(34):13711–13716, 2007.

[19] S. Kelly and M. I. Heywood. Genotypic versus behavioural diversity for teams of programs under the 4-v-3 keepaway soccer task. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2014.

[20] 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.

[21] S. Kelly and M. I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis. In *ACM Genetic and Evolutionary Computation Conference*, pages 1143–1150, 2015.

[22] 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.

[23] 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.

[24] Richard E. Korf. Sliding-tile puzzles and rubik's cube in AI research. *IEEE Intelligent Systems*, pages 8–12, 1999.

[25] D. Kunkle and G. Cooperman. Solving the Rubik's Cube: Disk is the new RAM. *Communications of the ACM*, 51(4):31–33, 2008.

[26] P. Lichodzijewski. *A symbiotic bid-based framework for problem decomposition using genetic programming*. PhD thesis, Faculty of Computer Science, Dalhousie University, 2011.

[27] P. Lichodzijewski and M. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *ACM Genetic and Evolutionary Computation Conference*, pages 363–370, 2008.

[28] 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.

[29] D.E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Machine Learning Research*, 11:241–276, 1999.

[30] T. Pepels, M. H. M. Winands, and M. Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in games*, 6(3):245–257, 2014.

[31] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. God's number is 20. `http://cube20.org`, 2010.

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

[33] D. G. Shapiro, H. Munoz-Avila, and D. Stracuzzi. Special issue on structured knowledge transfer. *AI Magazine*, 32(1), 2011.

[34] R. J. Smith, S. Kelly, and M. I. Heywood. Discovering Rubik's Cube subgroups using coevolutionary GP – a five twist experiment. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 789–796, 2016.

[35] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

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

[37] R. R. Sutton and A. G. Barto. *Reinforcement Learning: An introduction*. MIT Press, 1998.

[38] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via intertask mappings for temporal difference learning. *Journal of Machine Learning Research*, 8:2125–2167, 2007.

[39] Gunter P. Wagner and Lee Altenberg. Perspective: complex adaptations and the evolution of evolvability. *Evolution*, 50:967–976, 1996.