Real Valued Actions In Tangled Program Graphs

Ryan Amaral
Faculty of Computer Science
Dalhousie University
Halifax, Canada
ryan.amaral@dal.ca

Caleidgh Bayer
Faculty of Computer Science
Dalhousie University
Halifax, Canada
caleidgh.bayer@dal.ca

Alexandru Ianta
Faculty of Computer Science
Dalhousie University
Halifax, Canada
aianta@dal.ca

Robert Smith
Faculty of Computer Science
Dalhousie University
Halifax, Canada
robert.smith@dal.ca

Malcolm Heywood
Faculty of Computer Science
Dalhousie University
Halifax, Canada
mheywood@cs.dal.ca

***Abstract*—Tangled Program Graphs (TPGs), a modular genetic programming algorithm, has been shown in the past to perform well in reinforcement learning environments which assume discrete actions (e.g. Atari console games). It is the goal of this research to expand the capability of TPG into the continuous control domain, where real valued actions are required, as well as to showcase the potential of using real valued action generation to aid in discrete action tasks. In addition to giving TPG real value capabilities, we also explore the impact of diversity maintenance through occasionally introducing new genetic material for TPG to work with during evolution, as well as a method of "speeding up" evolution through repeated mutations with a process called rampancy. With this in mind, a 2D bipedal walker control task will be assumed in which multiple real-valued control actions have to be specified per state, as well as 3D tasks through ViZDoom which accepts discrete actions.**

***Index Terms*—genetic programming, reinforcement learning, complex control**

I. Introduction

A. Overview

In previous work, TPG has always made use of discrete actions [6]–[8], [16], [17]. This limits the environments in which TPG can act to those that accept exclusively discrete actions. Attempting to replace real-valued actions with discrete values (discretization) results in too many discrete actions (i.e. an example of the curse of dimensionality). In this work an implementation of TPG is showcased that is capable of producing a vector of real valued actions of arbitrary size.[1]

This paper also explores the impact of methods of diversity maintenance on a TPG population. The primary motivation for this is that in prior experiments, TPG tends to lose diversity without explicitly working diversity maintenance into evolution. This is a problem because as diversity decreases, the amount of possible directions in which a population can search is reduced, increasing the risk of being trapped in a local minima (or an example of the exploration-exploitation trade-off).

One proposed method of diversity maintenance (as a contribution of this work) is to occasionally evolve a sub-population of Symbiotic Bid Based (SBB) individuals, of which some elite SBB individuals will be integrated into the TPG population. SBB is a pre-cursor to TPG in which candidate solutions may only take the form of single teams of programs. TPG generalizes this to the self-organization of graphs of teams of programs. Having these SBB individuals introduced into TPG provides new genetic material, increasing diversity.

Rampancy will also be discussed, which is a method of repeating mutations in a controlled fashion in order to create greater differences between original and newly mutated individuals.

B. Outline

The rest of this paper is organized as follows: First, section II Background will introduce the concepts of reinforcement learning, genetic programming, evolutionary algorithms, SBB, and TPG. Following that in section III we describe some previous works making use of ViZDoom and bipedal walking environments. After that, the method of producing real valued actions in TPG in this work will be introduced in section IV Real Valued Tangled Program Graphs. Next section V Diversity Maintenance introduces the concepts used to increase the level of diversity in TPG, namely intermittent SBB subpopulations and rampancy. Section VI ViZDoom Experiments and section VII Bipedal Walker Experiments describe the experiment setup and results for each experiment set. This is all followed by the Conclusion, summarizing the work and discussing limitations and future directions to take this work.

II. Background

A. Reinforcement Learning

Reinforcement Learning (RL) is a type of Machine Learning (ML) algorithm, where a given problem is posed as an

[1]https://github.com/Ryan-Amaral/PyTPG/tree/new-tpg is used in the bipedal walker task, another implementation in Java is used for the ViZDoom tasks.

interaction between an agent (instance of an ML algorithm) and an environment [18].

The environment first produces an observation, the representation of the current state of the environment (s_t). The agent takes this observation to produce an action (a_t). The environment is then updated based on the action, producing a new observation for the agent to use (s_{t+1}). In addition to each observation, the agent also gets a reward based on the current state of the environment (r_{t+1}). The ultimate goal of the agent is to maximize total cumulative reward or $\sum_{t=1} r_t$. Thus, an episode is the full set of interactions with the environment, from the start state ($t = 0$) to the last ($t = n$), when the environment ends and a terminal reward is received by the agent.

Often in RL, results of an agent are averaged over multiple episodes. Overall results are often reported as the mean performance over multiple runs with the same parameterizations (often displaying the standard deviation as well).

B. Genetic Programming

Genetic Programming (GP) is a type of ML algorithm based roughly on how genes behave in biological evolution [5]. GP's can be composed in different structures, in this work we use what is called linear GP [1].

In linear GP, individuals are represented as sequences of instructions, along with a set of registers that the program can make use of to store intermediary or final results. The register values are also used as return values when executing a program, for example just returning the value stored in the first register after execution for a single real valued output. Programs have $prog_{regs}$ registers.

Each instruction (as used in all algorithms in this work) is comprised of four components, a mode, operation, destination, and source. The mode represents whether the instruction will take an input from the environment's observation or from one of its registers, it is Boolean valued. The operation selects a single math operation (from a finite set of operations) to perform on the value currently stored in the output register and/or the input. The destination represents which register to store the result of the given instruction in, and can take on a value up to $prog_{regs}$. The source is the index in the observation or register set to pull a value from, this value can range up to $max(size(observation), prog_{regs})$.

Marking the value in the destination register as x and the value as specified by source and mode as y, the following are all of the operations used in all algorithms in this work (*op_set*): $x+y$, $x-y$, $x \times y$, $x \div y$ (only if $y \neq 0$), $x = x \times -1$ (only if $x > y$), and $cos(y)$.

To mutate a program, 4 parameters are used: $inst_{del}$, $inst_{add}$, $inst_{swp}$, $inst_{mut}$. Each representing the probability to delete an instruction, add an instruction, swap two instructions, and mutate an instruction, respectively. Mutating an instruction changes one of its components to any other value within the valid range. In this work programs have a max size during initialization ($prog_{maxInit}$), but through mutation there are no limits aside from being at-least one instruction long.

C. Evolutionary Algorithms

GP is typically used with an Evolutionary Algorithm (EA), which itself is roughly based on biological evolution [4]. The general process of an EA is shown in Algorithm 1, which is used with all of the algorithms tested in this work.

This specific formulation represents a 'breeder' in which all individuals have their performance evaluated and then the worst performing gap portion of individuals are deleted. Variation operators facilitate adding back the gap portion (based on a fixed target population size) of new individuals through a process that inherits material from the $1 - gap$ portion of survivors. G represents the number of generations to run evolution for, N is the population size, gap represents the portion of the population to delete (and add) each generation, and e is the number of episodes to run each agent for. Regardless of algorithm used (e.g. GP, neural network), the general flow remains the same. There are many more parameters that may be needed based on what algorithm is being used.

Algorithm 1 EvolutionaryAlgorithm: Performs evolution of a population in an environment.

INPUT: Evolutionary parameters (e.g. G, N, gap, e, mutation parameters).
OUTPUT: Varies, could be the set of $1 - gap$ individuals representing the survivors at generation G or the single best performing individual.

```
1)  pop = InitializeRandomPopulation(size = N)
2)  For (g In 1..G)
3)      For (individual In pop)
4)          individual.fitness = Perform(individual, e)
5)      survivors = GetFittest(from = pop, portion =
        1 − gap)
6)      pop = survivors
7)      For (i In 1..(N × gap))
8)          child = CloneRandomSurvivor(survivors)
9)          child.Mutate(mutateParams)
10)         pop.Add(child)
```

D. Symbiotic Bid Based Genetic Programming

Symbiotic Bid Based (SBB) GP is a GP algorithm, where multiple programs are grouped together in what is called a team [12]. Each program has a fixed action associated with it, historically, a single discrete action. These programs are no longer used to provide the (real valued) actions directly, but are instead used to "bid" on which program should have its action used (the action being fixed, typically integer valued). So unlike plain GP, SBB is relegated to tasks which require only discrete actions, aside from potential discretization of real valued action spaces (an approach that does not scale).

There is a type of SBB called hierarchical SBB, in which SBB is grown in layers, each layer using the previous layers as potential actions [3], [9]. 1-layer hierarchical SBB would be comprised of teams whose programs only refer to discrete atomic actions. 2-layer hierarchical SBB would be comprised

of teams whose programs only refer to 1-layer hierarchical SBB individuals. Each layer would be evolved in distinct phases. In the top most level each team is referred to as a root-team, only the root-teams count as agents to an environment.

When evolution starts, teams are created one by one. Each team is given at-least $team_{min}$ learners, each with randomly generated programs and atomic actions (or team actions if a higher layer of SBB). These initial teams will have between $team_{min}$ and $team_{maxInit}$ learners, selected uniformly.

Throughout evolution, at each generation, all of the root-teams are eligible to be cloned into a new individual. The newly cloned individual will initially share all of the same learners as the parent before mutation. The variables $lrnr_{del}$, $lrnr_{add}$, and $lrnr_{mut}$ define the probability of learner deletion, addition, and mutation, respectively.

First learner deletion happens, a learner is deleted with probability $lrnr_{del}$, if that is successful (and other conditions are met), another learner gets deleted with probability ${lrnr_{del}}^2$, and so on (with probability ${lrnr_{del}}^3$...). Deletion can happen as long as there are more than $max(2, team_{min})$ learners.

Learner addition then follows the same pattern as learner deletion. Any learner in the entire population (of the current layer) can be added to a team provided that the learner is not already in the team. Learners can be added without restriction to team size unless $team_{max}$ is defined, which sets the maximum number of learners a team can have.

Then learner mutation happens. Each learner is given $lrnr_{mut}$ probability to mutate. When a learner mutates, both the program and the action have a chance of mutating, based on $prog_{mut}$ and act_{mut} respectively, at-least one of these must occur. An action is mutated by swapping out the current action for an eligible one, either a random (different) discrete action if a first layer SBB, or a (different) team from the previous layer for higher layers of SBB.

Execution starts at a root-team, a team of the highest current layer. Each of that team's programs bid, the highest bidder has its action selected. If that action is another team, the same process is repeated on that team. If that action is an atomic action, that value is returned as the action.

E. Tangled Program Graphs

Tangled Program Graphs (TPG's) are very similar to hierarchical SBB, any SBB individual is a valid TPG individual. Unlike hierarchical SBB, TPG is not evolved in distinct phases, and it is also capable of having learners' actions be teams of any "layer" (the concept of layers doesn't matter to TPG). TPG thus produces graphs with arbitrary links, as the term "tangled" suggests.

When a TPG population initializes, teams are created just like in SBB. Mutation and execution are different.

Being composed as a graph rather than a tree (as in SBB) could possibly introduce infinite loops during execution. This is overcome by not allowing the same team to be visited more than once in each environment interaction, as well as by ensuring that each team has at-least one learner with an atomic action. When adding learners during team mutation, an additional constraint is that an added learner must not have its action be a team action that refers to the team itself (the team being currently mutated).

Additional changes are made to the learner mutation process. The variable act_{atom} is added, representing the probability that when a learner's action mutates, it becomes an atomic action (vs a team action). If the action mutates into a team action (whether or not it already was a team action), the new team cannot be the team that the learner being mutated belongs to.

During execution, the process is very similar for a TPG individual when compared to an SBB individual. The only change is that as each node (team) of the graph is visited during a given environment interaction, it is added to a list of visited nodes. Before taking any learners' team action, the team is compared to the visited node list, if it is already in the list then the next best learner's action is taken (the next learner following the same constraint). This along with ensuring that each team has at-least one atomic action satisfies the before mentioned constraint that each team will not be visited more than once per action selection, doing away with any possibility of an infinite loop, and ensures that an action will always be returned.

III. Related Work

A. ViZDoom

The ViZDoom environment has seen use in competition, [20], further promoting its use in the research community. The competition sets machine learning based bots against each other in "deathmatches" with the goal of obtaining the most kills. In the first two iterations of the competition (2016-2017) submissions all made use of some deep learning architecture (though a few submissions use unspecified algorithms).

The agents were capable playing the game at a reasonable skill level, though lacked much strategy and intelligence. For example, in the 2016 iteration, one solution simply stayed crouched, and no other agents were used to that so it died far less often. In addition to this, agents exhibited a lack of memory by losing track of enemies once they go off screen. Ultimately, a human skilled at the game was able to consistently beat the agents.

B. Bipedal Walking

To our knowledge there are no published GP results on the OpenAI bipedal walker task, which is a major contribution of this work. However there are results on the task which make use of neural networks, as well as GP results on different bipedal walking environments.

For GP results, there is [15] in which the authors evolve control policies for a 3D bipedal walker task. They use GP to create a model of the nervous system to sync movement of the different joints, each joint being controlled by a neural oscillator model (a basic rhythmic generator). Only up to 4 steps of walking were generated by their model, though this is a more complex environment then the one used in this work

as it controls 3 times as many joints (12 vs 4) and exists in a 3D environment.

Another bipedal locomotion attempt by [19] also makes use of GP. This work also makes use of more complex 3D environments with more joints to control, using two different environments. In one environment they were capable of obtaining slow movement with small steps, much slower than a human would move. In the other environment the biped was capable of at-most 2 steps before falling over.

In [13] an older version of the bipedal walker environment used in this work is used. They compare a method of optimizing neural net weights based on distributions (UMDAc) to an EA based neural net. In general results showed that the UMDAc outperformed EA, with UMDAc coming out on top with a probability of 0.821, though with a relatively large variance.

Finally, [11] generates symbolic (mathematical) policies by distilling deep neural nets. These final symbolic policies are similar to GP programs, in fact depending on the operations allowed in GP the results could come out exactly the same. They show that the distilled symbolic policies outperform other deep learning methods in general by showing that the average rank of their approach for each task is better than that of any other approach they tested.

IV. Real Valued Tangled Program Graphs

An implementation of TPG which uses real valued programs is used in this work, so that discretization of the environment is not required, to decrease any potential limitation imposed by a fixed discretization.

This new implementation of TPG produces real valued actions through action programs, which are the same as any other program. TPG's programs have single actions attached to them, if it's an atomic action it was previously defined as a discrete value, this new implementation replaces the discrete value with a program. An action program is ran when an atomic action is needed, the values in the registers represents the output vector.

Action programs have their initial max size defined by $actProg_{max\_init}$, this value will also be used for GP examples in this work since GP will only make use of action programs, no bidding programs. The amount of registers used by action programs is specified by $actProg_{regs}$ (also used by GP in this work).

During evolution of TPG with real actions, the only difference from regular TPG is that when an action is mutated, in addition to potentially changing the action as a whole (new program or team), there is a 50% chance of the action program being mutated.[2]

This new version of TPG (only as used in the bipedal walker task) also makes use of a concept called hitchhiker removal. Every g_{hh} generations hitchhiker removal is ran, which removes any learners attached to any team, if those learners are never used over the course of an evaluation of that team on the environment.

[2]As happens in the Python implementation which was used in the bipedal walker task. A different Java version is used for the ViZDoom tasks which conditionally mutates the action program only if the corresponding bidding action was first mutated.

V. Diversity Maintenance

A. SBB Sub-Populations

In an effort to improve TPG performance in the bipedal walker task [2], a method of diversity maintenance making use of SBB subpopulations is used. A run of this type is referred to as TPG+SBB. Essentially, there will be a single on-going TPG population, but every once in a while, an SBB subpopulation will be evolved, which will have its best individuals integrated into the TPG population.

When to switch between the TPG population and an SBB population is decided by either a max number of generations on the SBB population (G_{sbb}), or a max number of generations without improvement (g_{fail}). The TPG population will continue on without limit to its generations (until completing the final generation of the run, G), but will trigger the start of an SBB sub-population when it fails to improve its best fitness over g_{fail} generations in a row.

B. Rampancy

Another method, rampancy, is used for attempting to increase diversity. Rampancy is used in the mutation process at the team level, and defines how many times to repeat the mutation process, to increases changes imposed by mutation. This method will be used for the bipedal walker and ViZDoom tasks, and takes places on a per-team basis.

Rampancy is defined as a 3-tuple $rampancy = (freq, r_{min}, r_{max})$, where $freq$ is how frequently to perform rampancy in terms of generations, so with $freq = 5$, rampancy will take place every 5 generations. r_{min} and r_{max} define the range in which a random number of rampancy iterations will take place. If r_{min} and r_{max} are equal, then rampancy occur for that many iterations.

VI. ViZDoom Experiments

A. Overview

The ViZDoom platform consists of multiple different environments, each with similar base mechanics, but different goals [10]. Four environments are used: Defend the Center, Defend the Line, Health Gathering, and Take Cover. All environments share the same observations space, a $160 \times 120 \times 3$ RGB image, each component being 8 bits (each colored pixel consists of 24 bits). The action space and reward differs among the different environments, which will be specified ahead.

In Defend the Center the agent must survive in a room with five re-spawnable monsters. The agent can only turn left or right and shoot, ammo is limited. A reward of 1 is given for each monster killed, and the episode ends when the agent dies.

Defend the Line is similar to Defend the Center, except the monsters come back stronger each re-spawn.

In Health Gathering the agent must collect health packs, and constantly looses life, there are no monsters. The agent can only move forward and rotate left or right. A reward of 1

TABLE I
PARAMETERS FOR THE TPG VIZDOOM RUNS. $*$ IN THE VERSION OF TPG USED IN THESE RUNS, LEARNER MUTATION IS ATTEMPTED AS EACH LEARNER IS ADDED FROM A PARENT TO THE NEW TEAM AND PROGRAM MUTATION IS ALSO ATTEMPTED BUT NOT GUARANTEED.

Parameter	NRAP	RAL	RAP	RAPF	RAPS
N	120				
G	10,000				
g_{flip}	0			3,000	0
gap	0.5				
e	5				
$rampancy$	(1,5,5)				
$team_{min}$	12				2
$team_{max}$	12				4
$team_{maxInit}$	12				2
$lrnr_{del}$	0.7				
$lrnr_{add}$	0.7				
$lrnr_{mut}$	?*				
$prog_{mut}$	?*				
act_{mut}	0.2				
act_{atom}	0.5				
$prog_{maxInit}$	128				
$prog_{regs}$	8				
$actProg_{maxInit}$	128				
$actProg_{regs}$	7	0	7		
$inst_{del}$	0.5				
$inst_{add}$	0.5				
$inst_{swp}$	1.0				
$inst_{mut}$	1.0				

is given for each frame survived, up to a maximum of 2100 frames.

In Take Cover monsters are spawned in repeatedly and can shoot projectiles at the player. The agent can only move left and right. A reward of 1 is given for each frame.

B. Parameterization

Five different TPG run types are considered for the ViZDoom tasks. NRAP (no rampancy, action programs), RAL (rampancy, action labels (discrete actions)), RAP (rampancy, action programs), RAPF (same as RAP but doesn't allow learners to reference teams until generation $3,000$), RAPS (same as RAP, but smaller team sizes). TPG is parameterized as shown in Table I.

C. Results

Results are shown in Figure 1. These are early results, so how the runs will end up after ten thousand generations is yet to be seen, though we can observe the early characterizations of each run type.

In all environments except for Defend the Line, RAL runs seem to perform poorly relative to the rest, especially in Defend the Center and Health Gathering. RAL is not too far below the others in Take Cover, and even on average outperforms others on Defend the Line, though on these environments the differences between scores are relatively minuscule when compared to the other two environments. This could suggests that action programs provide superior performance to action labels (discrete actions), though perhaps only in certain types of tasks.

To assess the impact of Rampancy we can look at NRAP (no rampancy, action programs) and RAP (rampancy, action programs) as they only differ in use of rampancy. Though at this time in the runs not much can be said conclusively about the difference between the methods.

VII. BIPEDAL WALKER EXPERIMENTS

A. Overview

We use the environment Bipedal-Walker-v3, which is implemented in the Box2D physics engine, and is released as an OpenAI Gym environment [2]. There is a normal version which features a relatively flat yet uneven terrain, and a hardcore version which contains additional obstacles. At this time, we only have results for the normal version, so those will be presented. Also finalized runs will contain results from GP, SBB, TPG, and TPG+SBB. Only the GP and TPG+SBB runs were finished in time to present in this work.

The action space consists of activity levels of four joints, two hips and two knees (which a boxy hull/body sits on top of). The observations space consists of the hull angle, angular velocity, and horizontal and vertical speeds, the positions and angular speeds of the four joints, whether each leg is touching the ground, and ten range finder measurements. This observation space is extended in our runs by adding $\{sin(t)/k \mid k\ in\ 1..3\}$ and $\{cos(t)/k \mid k\ in\ 1..3\}$, to further motivate the adoption of cyclical/sinusoidal motion which has been shown to be beneficial for bipedal locomotion [14], [19]. This brings the total observation space size to 30. Reward is given for distance moved forward, -100 points is given if the agent falls over, and a small penalty is continually applied for energy use.

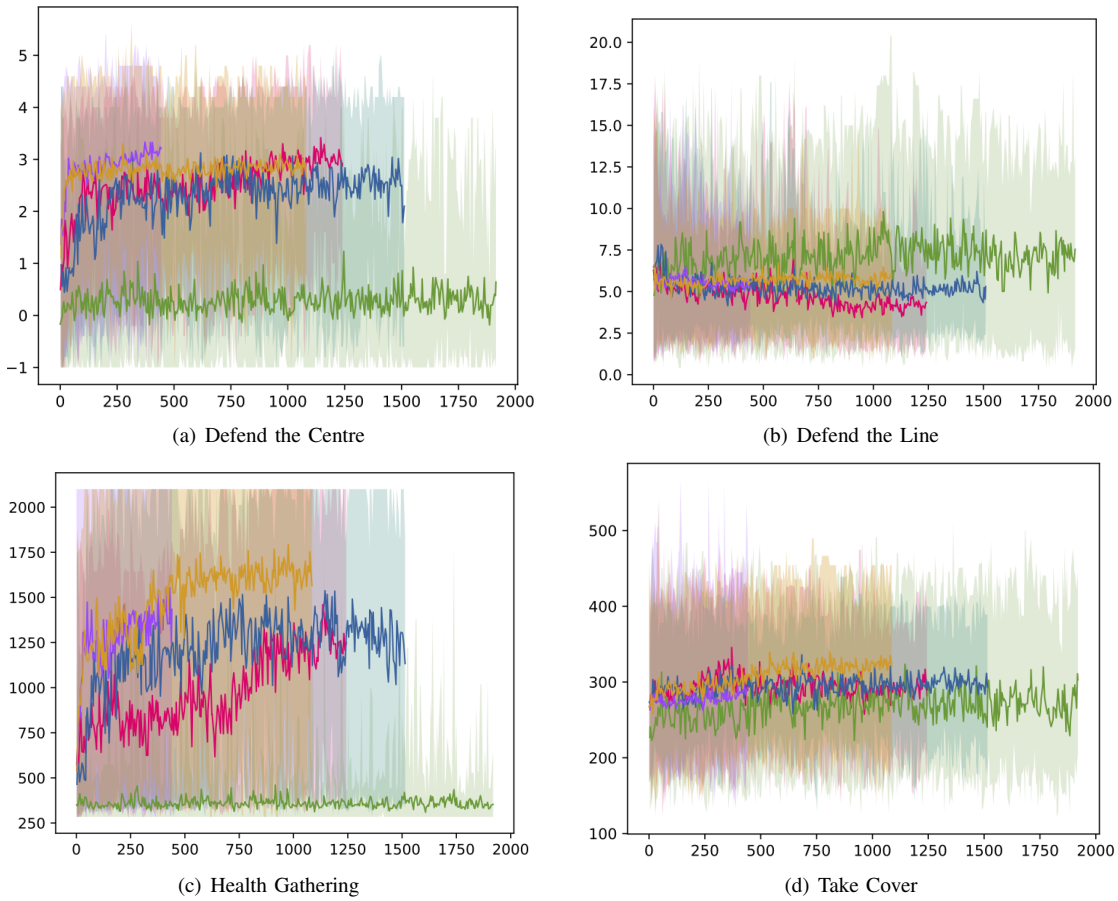
B. Parameterization

The specific parameterizations for the GP and TPG+SBB runs on the bipedal walker runs are shown in Table II.

C. Results

The results throughout evolution for 5 GP and 5 TPG+SBB runs can be seen in Figures 2 and 3 with the overall mean and standard deviation of each run type shown in Figure 4. In the end, TPG+SBB achieves a higher mean performance and a lower standard deviation.

The final agents obtained from GP and TPG+SBB are compared in Figure 5 in regards to evaluation over 1000 episodes. Table III shows the mean and standard deviation obtained by each champion over the evaluation episodes. The GP agents show a bi-modal distribution due to one exceptionally well performing champion creating a separate higher peak. TPG+SBB runs consistently produced champions capable of high mean scores, however, some champions performed inconsistently over their one thousand evaluation episodes, ending up with a slightly higher standard deviation. Although TPG+SBB produced better performing champions on average, the single highest performing champion was by GP with a relatively low standard deviation (GP 4).



(a) Defend the Centre

(b) Defend the Line

(c) Health Gathering

(d) Take Cover

Fig. 1. Fitness curves during training. NRAP (no rampancy, action programs) in red. RAL (rampancy, action labels) in green. RAP (rampancy, action programs) in blue. RAPF (rampancy, phase flip) in purple. RAPS (rampancy, small initial teams) in orange.

TABLE II
PARAMETERS FOR THE GP AND TPG+SBB BIPEDAL WALKER RUNS.

Parameter	**GP**	**TPG+SBB**
N	360	360
G	10,000	10,000
gap	0.5	0.5
e	5	5
g_{hh}		100
G_{sbb}		500
g_{fail}		100
$rampancy$	(5,5,5)	(5,5,5)
$team_{min}$		2
$team_{max}$		4
$team_{max\_init}$		2
$lrnr_{del}$		0.3
$lrnr_{add}$		0.2
$lrnr_{mut}$		0.7
$prog_{mut}$		0.5
act_{mut}		0.7
act_{atom}		0.8
$prog_{max\_init}$		64
$prog_{regs}$		8
$actProg_{max\_init}$	512	64
$actProg_{regs}$	8	8
$inst_{del}$	0.5	0.5
$inst_{add}$	0.5	0.5
$inst_{swp}$	0.5	0.5
$inst_{mut}$	0.5	0.5

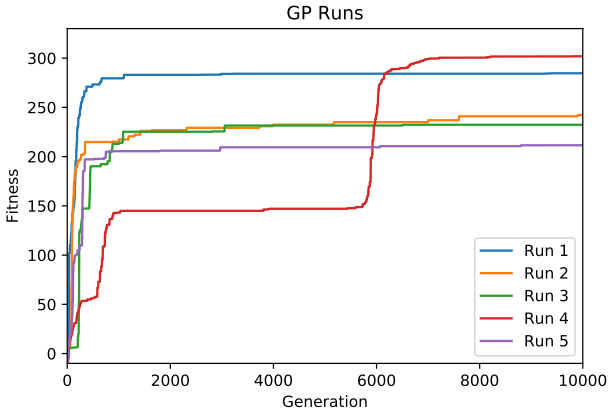


Fig. 2. GP champion scores throughout evolution of 5 different runs.

It is likely that relatively simple programs are sufficient to perform well, given that GP alone is capable of performing so well. This is supported by Figure 6, which shows that programs used by GP and SBB have roughly the same amount of instructions. TPG+SBB probably had higher success in general due to the constant influx of SBB individuals, cre-

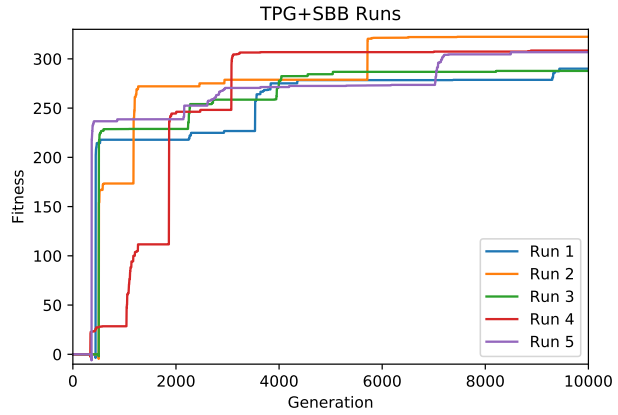


Fig. 3. TPG+SBB champion scores throughout evolution of 5 different runs.

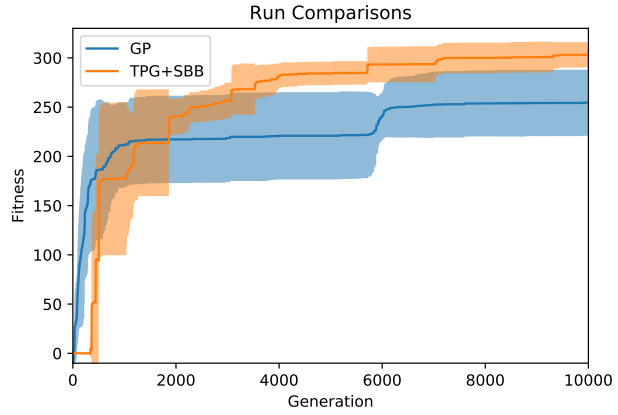


Fig. 4. Mean and standard deviation of champion scores throughout evolution, across 5 GP runs and 5 TPG+SBB runs.

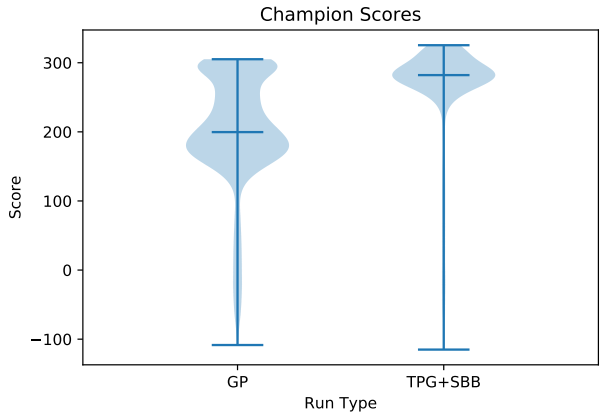


Fig. 5. Score distributions from champion agents of each run type with median bars. Each agent ran for 1,000 episodes, 5 agents for GP, 5 agents for TPG+SBB.

TABLE III
THE MEAN AND STANDARD DEVIATION OF SCORES OVER 1,000 EVALUATION EPISODES FOR THE 5 GP AND TPG+SBB RUNS.

Run	Mean	SD
GP 1	222.6	85.9
GP 2	142.6	96.7
GP 3	178.4	39.9
GP 4	**283.6**	**59.1**
GP 5	169.2	27.9
GP Mean	**199.3**	**61.9**
TPG+SBB 1	230.9	100.1
TPG+SBB 2	264.1	110.3
TPG+SBB 3	272.6	50.9
TPG+SBB 4	277.8	55.8
TPG+SBB 5	256.6	48.3
TPG+SBB Mean	**260.4**	**73.08**

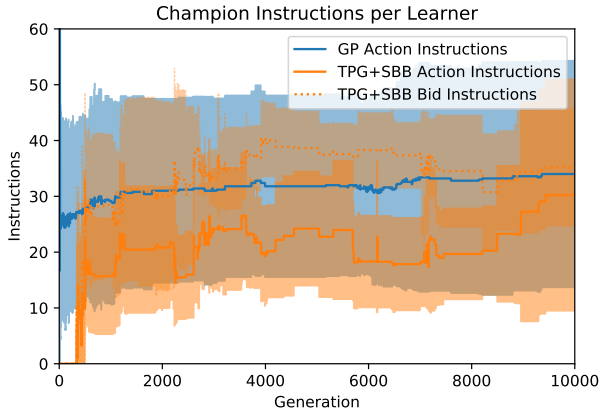


Fig. 6. Amount of instruction found per learner/program in for action and bid programs.

ating a relatively high baseline of diversity, and the multiple evolutionary restarts.

Figure 7 shows the amount of SBB subpopulation teams and learners found in the TPG population at each generation. The amount of teams remains somewhat consistent throughout evolution. Whenever an SBB team in the TPG population is mutated that new mutated team is not associated with the SBB subpopulation (though the original still is). So naturally as TPG evolves, the SBB teams will be replaced, and get filled back in consistently, so the level remains mostly neutral.

The amount of SBB learners in the TPG population is on a constant rise unlike the teams. This suggests that the TPG population is making good use of the SBB subpopulation learners, as they tend to contribute to teams that succeed in selection. Also naturally more learners get referenced per generation than teams, by the nature of how TPG works. The number of learners in the population naturally increases over evolution of TPG anyway, as opposed to the number of teams which remains mostly static. Of note is that SBB subpopulation learners make up on average about 62% of the total learner population by the end of ten thousand generations of evolution.

In Figure 8 we can see the probability that a given in-

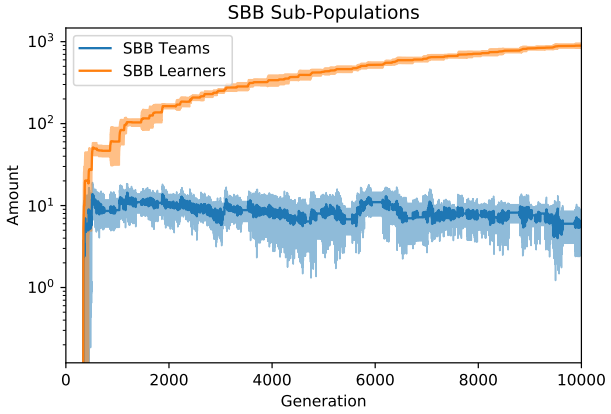


Fig. 7. Amount of teams and learners originating from SBB subpopulations used in the TPG population throughout evolution.

struction will use each index from an observation for each program type (from a single exemplary individual of each run type). Values from the upper half of the range are deemed significant here, opposite for values in the lower half of the range. The GP solution made significant use of the hull related variables, whereas the TPG+SBB solution hardly made any use of such variables. The lidar variables hardly saw any significant use, only $lidar6$ by TPG+SBB action programs. The most significant use of any limb related variable was $knee\ 2\ angle$ by TPG+SBB action programs, which was also fairly significantly used by TPG+SBB bid programs. Other limb related variables were mainly only significantly used by TPG+SBB action programs and the only significantly used limb variable by GP was $hip\ 2\ angle$. The added sin and cos variables saw significant use by each program type, mostly by TPG+SBB action programs using $cos/5$, $sin/20$, and $cos/20$, but also by TPG+SBB bid programs using $sin/20$, and GP programs using $sin/10$.

Figure 9 shows which actions are made through an episode. GP and TPG+SBB solutions both show cyclical properties, as is likely in locomotion tasks, with GP opting for a higher frequency of motion. Both solutions show heavy use of one hip and the opposite knee, hinting at a lunging gait with one leg kept forward doing the pulling and the hind leg keeping the walker upright, potentially providing push.

We can see that this lunging gait is what is being used in Figure 10. As suggested by the evidently quicker frequency used by GP in moving, Figure 10 (a) shows the GP walker making more abbreviated steps, not fully extending its leg out. This is contrasted by TPG+SBB in Figure 10 (b) which still uses the lunging gait, but makes use of larger steps, even sometimes getting some air.

VIII. Conclusion

A. Summary

TPG has been updated to enable the production of real valued vector outputs. This allows TPG to participate in domains which require real valued actions, and potentially improves upon the previous method of action labels in discrete action environments.

A new method of evolution called TPG+SBB makes use of SBB subpopulations, which provide champions to a main TPG population, which appears to improve average results in the test case examined (bipedal walker).

Rampancy, a method of increasing the iterations of mutation, seeks to increase the speed of evolution. Results remain inconclusive in early tests, but it does not seem to harm performance at-least.

B. Future Work

The ViZDoom and bipedal walker tests still require more time to be fully complete to provide full results. At the time of writing, tests for the bipedal walker are still in progress for the individual SBB and TPG runs. These runs are needed to properly analyze the influence of TPG+SBB beyond regular TPG. It is also desired to run similar experiments in the bipedal walker domain, but in the ”hardcore” version for a greater challenge and availability of comparisons to other work. Also further methods of diversity maintenance could be looked at.

References

[1] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

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

[4] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing, Second Edition*. Natural Computing Series. Springer, 2015.

[5] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3:95–99, 1988.

[6] Stephen Kelly and Malcolm I. Heywood. Emergent tangled graph representations for atari game playing agents. In *European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 64–79. Springer, 2017.

[7] Stephen Kelly and Malcolm I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 195–202. ACM, 2017.

[8] Stephen Kelly and Malcolm I. Heywood. Emergent solutions to high-dimensional multitask reinforcement learning. *Evolutionary Computation*, 26(3):378–380, 2018.

[9] Stephen Kelly, Peter Lichodzijewski, and Malcolm I. Heywood. On run time libraries and hierarchical symbiosis. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[10] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE Press, 2016.

[11] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5979–5989. PMLR, 18–24 Jul 2021.

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

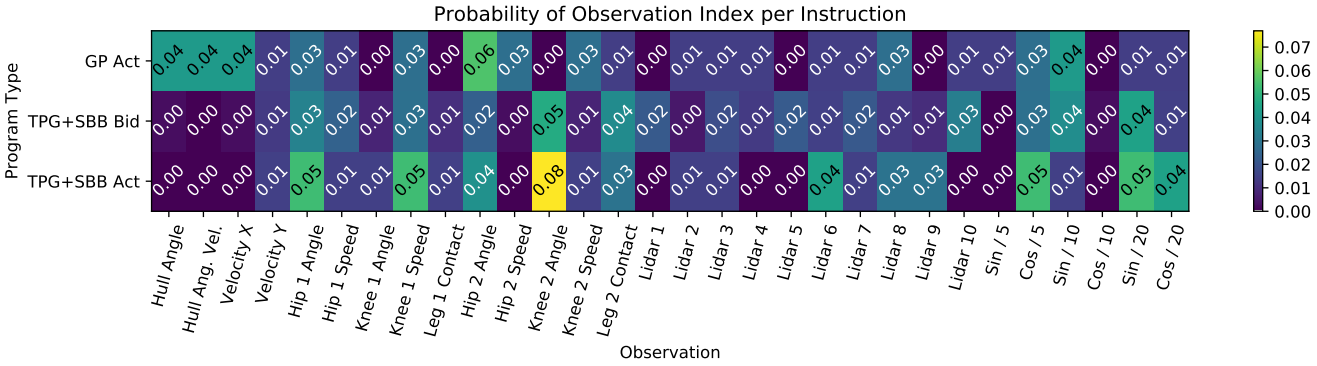


Fig. 8. Probability of an instruction to index the given observation. Probabilities written in white represent probabilities in the bottom half of the range, in black, the upper half.

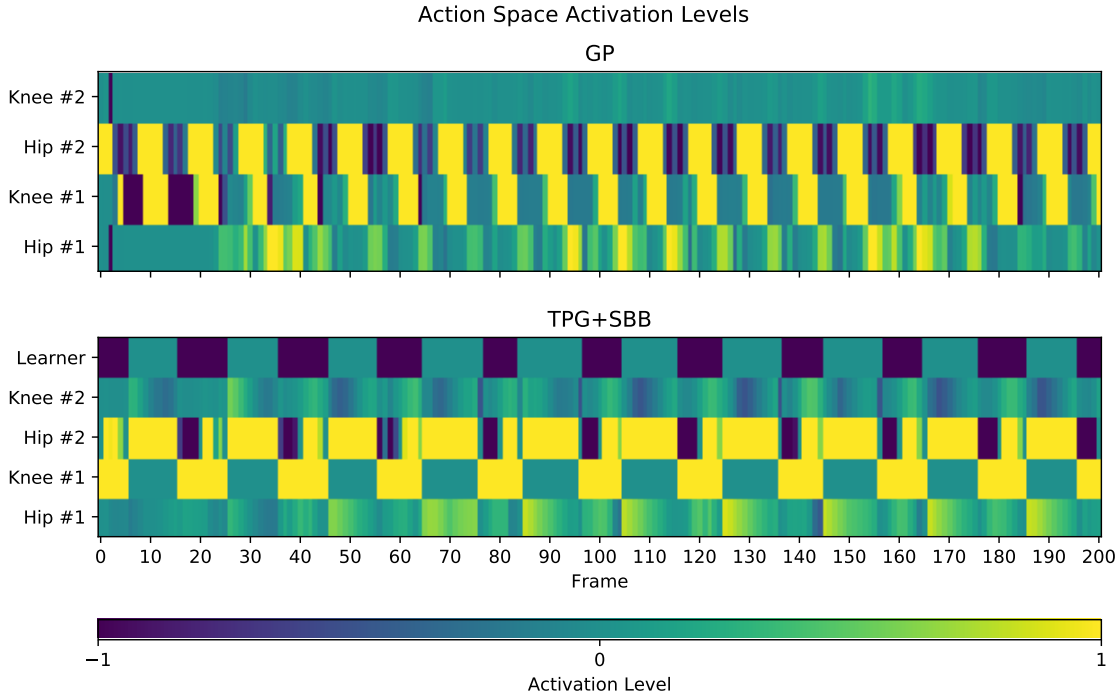
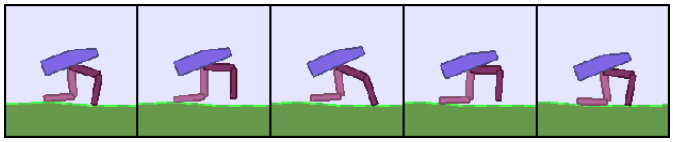
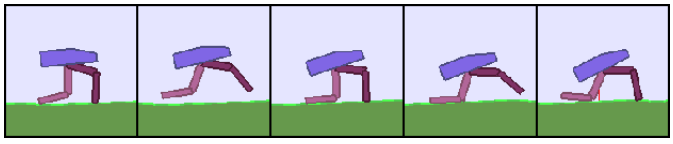


Fig. 9. Activation level of each action/joint at each time step in the first 200 time steps of a run. Learners in use shown where applicable, representing which learner is being selected for its action program.



(a) GP Gait



(b) TPG+SBB Gait

Fig. 10. Renders of the gaits used by GP and TPG+SBB. Each showing two full steps from front foot lifting to landing.

[13] Mikel Malagon and Josu Ceberio. Evolving neural networks in reinforcement learning by means of umdac, 2019.

[14] Jared M. Moore, Catherine L. Shine, Craig P. McGowan, and Philip K. McKinley. Exploring Bipedal Hopping through Computational Evolution. *Artificial Life*, 25(3):236–249, 2019.

[15] S. Ok, K. Miyashita, and K. Hase. Evolving bipedal locomotion with genetic programming - a preliminary report. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 2, pages 1025–1032 vol. 2, 2001.

[16] Robert J. Smith and Malcolm I. Heywood. Scaling tangled program graphs to visual reinforcement learning in vizdoom. In *European Conference on Genetic Programming*, volume 10781 of *LNCS*, pages 135–150. Springer, 2018.

[17] Robert J. Smith and Malcolm I. Heywood. Evolving Dota 2 Shadow Fiend bots using genetic programming with external memory. In *Genetic and Evolutionary Computation Conference*, pages 179–187. ACM, 2019.

[18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[19] Krister Wolff and Mattias Wahde. *Evolution of Biped Locomotion Using Linear Genetic Programming*. 10 2007.

[20] Marek Wydmuch, Michal Kempka, and Wojciech Jaśkowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 11:248–259, 2019.