Evolving Multimodal Networks for Multitask Games

Jacob Schrum and Risto Miikkulainen

Abstract—**Intelligent opponent behavior helps make video games interesting to human players. Evolutionary computation can discover such behavior, especially when the game consists of a single task. However, multitask domains, in which separate tasks within the domain each have their own dynamics and objectives, can be challenging for evolution. This paper proposes two methods for meeting this challenge by evolving neural networks: 1) Multitask Learning provides a network with distinct outputs per task, thus evolving a separate policy for each task, and 2) Mode Mutation provides a means to evolve new output modes, as well as a way to select which mode to use at each moment. Multitask Learning assumes agents know which task they are currently facing; if such information is available and accurate, this approach works very well, as demonstrated in the Front/Back Ramming game of this paper. In contrast, Mode Mutation discovers an appropriate task division on its own, which may in some cases be even more powerful than a human-specified task division, as shown in the Predator/Prey game of this paper. These results demonstrate the importance of both Multitask Learning and Mode Mutation for learning intelligent behavior in complex games.**

I. Introduction

Video games often feature computer controlled opponents, Non-Player Characters (NPCs), which human players must defeat to do well in the game. Creating intelligent behavior

for such NPCs traditionally requires extensive hand-coding and troubleshooting by programmers. However, there has been much interest, and some success, in the academic community in evolving NPC behavior for games [1, 9, 16, 20]. These approaches each treat the game as a single task, and optimize the NPC behavior for that task.

However, many entertaining games consist of multiple tasks. Even the classic game of Pac-Man involves two tasks: the NPC ghosts chase after Pac-Man and try to catch him, but as soon as he eats a power pellet the task switches, and the ghosts must run from Pac-Man or be eaten.

Recently, the multitask nature of games has been noted, and a few methods have been developed to deal with them. One such approach is interactive evolution, as demonstrated in NERO [12], a game in which agents can learn many different behaviors because a user constantly interacts with the system by adjusting the fitness function and changing the agents' environment. However, this approach is prone to forgetting old behaviors when learning new tasks, which is a problem requiring considerable user effort to overcome [4].

Another approach to dealing with multiple tasks is evolution of a subsumption architecture composed of neural networks [15, 17]. This approach requires the programmer to divide a domain into its constituent tasks, and develop

Jacob Schrum and Risto Miikkulainen are with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712 USA (email: {schrum2,risto}@cs.utexas.edu)

978-1-4577-0011-8/11/$26.00 ©2011 IEEE
effective training scenarios to evolve separate network controllers for each task. These controllers are combined into a hierarchical controller which is also evolved.

A third approach, Multitask Learning [2], is similar in that the task division must be known, but different in that only a single network controller is learned. This controller has separate output modes corresponding to each task, and uses knowledge of the current task to choose which mode to use in a given situation. This approach is typically used in a supervised learning setting; Multitask Learning is rarely combined with evolution, and so far has not been aimed at evolving agent behavior. Therefore, multitask evolution for games is one approach evaluated in this paper.

While it may be easy to divide a game into its constituent tasks in some cases, games are typically complex enough that an appropriate division is difficult to achieve. Not all games have a clear task division like Pac-Man. One approach to multitask domains that addresses this problem directly is Mode Mutation, introduced in [10], which lets evolution decide how many modes to have and when to use them. Unlike the subsumption and Multitask Learning approaches, Mode Mutation can be used without knowledge of the domain's task division. Interestingly, as shown in this paper, even if such a division is available, Mode Mutation may

sometimes discover one that is better. In this paper, the original Mode Mutation is evaluated, as well as an enhanced version that makes such discovery possible.

Both Multitask Learning and Mode Mutation create multimodal networks, i.e. networks with access to multiple modes of behavior. These approaches are evaluated in two multitask games designed for this paper. After these games are reviewed, the evolutionary method is described, along with specific details on Multitask Learning and Mode Mutation. The method section is followed by a description of experiments, results, and ideas for future work.

II. Multitask Games

This section defines multitask games and describes two such games designed for the experiments of this paper.

A. *Definition*

In multitask games, NPCs perform two or more separate tasks, each with their own measures of performance. In the extreme case, performance in one task is unrelated to performance in other tasks. This extreme view makes it easy to analyze task performance independent of other tasks, and is therefore the basis of the domains in this paper. However, multitask games are only interesting if it is desirable to have NPCs capable of performing all tasks. Therefore, all tasks

presented are related in that they place NPCs in the same environment with the same sensors.

There is an important distinction between the multitask games of this paper and other games with multiple tasks. In a multitask game, tasks are isolated and it is always clear what the current task is. In Pac-Man, the task division is clear because eating a power pellet causes a task switch, but the tasks are not isolated because the positions of agents at the task switch affect the performance in the next task. A game like Unreal Tournament also has multiple tasks, but there is no strict separation between them; NPCs choose which task to perform when, such as gathering items and fighting, and may even do multiple tasks simultaneously. Note that an NPC's *internal* state is irrelevant for determining whether the game has separable tasks; the multitask nature of a game is a property of the game, and not of the NPCs programmed to play it. This paper concerns itself solely with games where the task division is clear and performance in each task is unaffected by behavior in other tasks, though it will be possible to apply the methods in this paper to games with less strict task divisions in the future.

Although the task division is clear, the NPCs may not have access to this knowledge. Therefore, this paper deals both with agents that have a priori knowledge of the task they face, *and* agents that must figure out how to behave

despite not knowing which task they face.

To assure that these domains are challenging, they are designed to involve tasks in which good behavior in one task is bad behavior in another task. The separate tasks still have underlying similarities, but different behaviors are needed across tasks to be effective NPCs in the game as a whole.

All tasks in this paper are multiagent tasks designed using the simulation environment BREVE [6]. In each task, evaluation begins with a team of NPCs surrounding the player. Task evaluations have limited duration and are independent from each other. All agents can move forward and backward and can turn left and right with respect to their current heading.

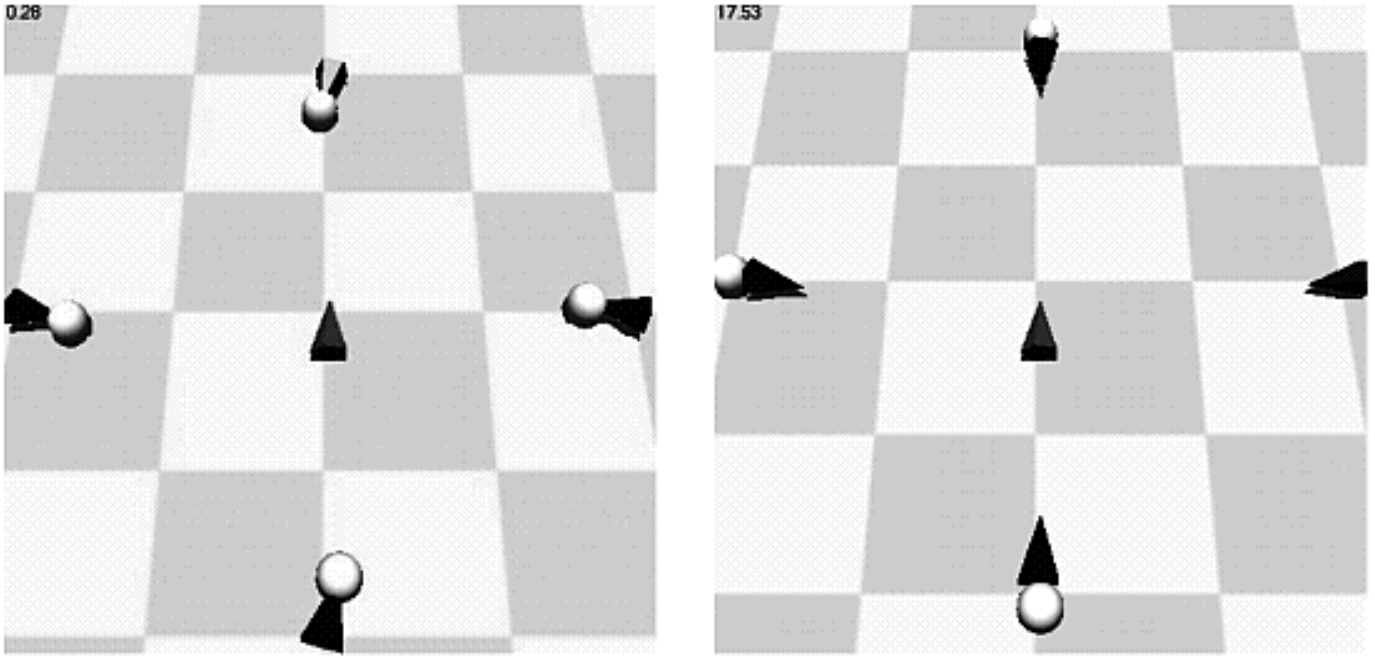
Although the player stands for a human player in principle, a scripted, task-dependent agent was used in the experiments instead to make a large number of evaluations possible. This "player" agent will be referred to as the "enemy" throughout this paper. The initial heading of the enemy is random in each evaluation to make evaluation noisy, which requires NPCs to learn situational behavior and prevents blind memorization of enemy trajectories from succeeding. Informal experiments show that NPC behavior evolved against the scripted enemy is still interesting and challenging for humans to overcome.

The multitask games designed for this work are Front/Back Ramming (FBR), which requires NPCs to be alternately aggressive with and protective of different parts of their body depending on the task, and Predator/Prey (PP), which contrasts attacking the enemy with running away from it.

Each domain is explained in full detail next.

B. Front/Back Ramming Game

This game requires both offensive and defensive behavior from NPCs in each task to be successful, but these behaviors



(a) Front Ramming task (b) Back Ramming task

Fig. 1: Front/Back Ramming. Fig. 1a shows the start of a Front Ramming task and Fig. 1b shows the start of a Back Ramming task. In both tasks the NPCs start pointed at the enemy in the center. The rams are depicted by white orbs attached to the NPCs. In the Front Ramming task, NPCs can start attacking the enemy immediately, but in the Back Ramming task they must turn around first. Learning which behavior to exhibit is difficult because different behavior is needed in the different tasks, even though the NPCs' sensor readings would be the same in each of the above situations.

are needed under different circumstances in each task.

Each NPC is equipped with a battering ram, and is therefore called a rammer (Fig. 1). If a ram hits the enemy, then the enemy is damaged, but if the enemy hits a rammer, then the rammer takes damage. All agents start with 50 hit points, and every hit removes 10 hit points. If the enemy dies, all agents are reset to their starting locations around the enemy before the enemy comes back to life, thus giving the evolving NPCs a chance to accrue additional fitness. When NPCs die, they are dead for the rest of the evaluation.

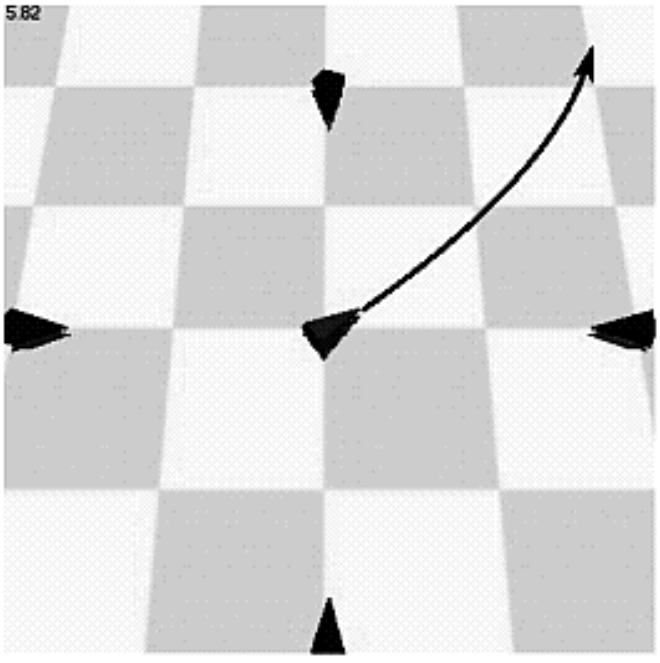
This game consists of Front Ramming and Back Ramming tasks, the only difference being where on the rammer the ram is located. When Front Ramming, rammers start with their rams pointed at the surrounded enemy. When Back Ramming, the rams start facing away from the enemy, and rammers must first turn in order to ram the enemy.

Enemy behavior is essentially the same in both tasks: It will try to circle around the rammers to hit them from the unprotected side if possible, but if threatened by the rams, it will prefer to run and avoid damage.

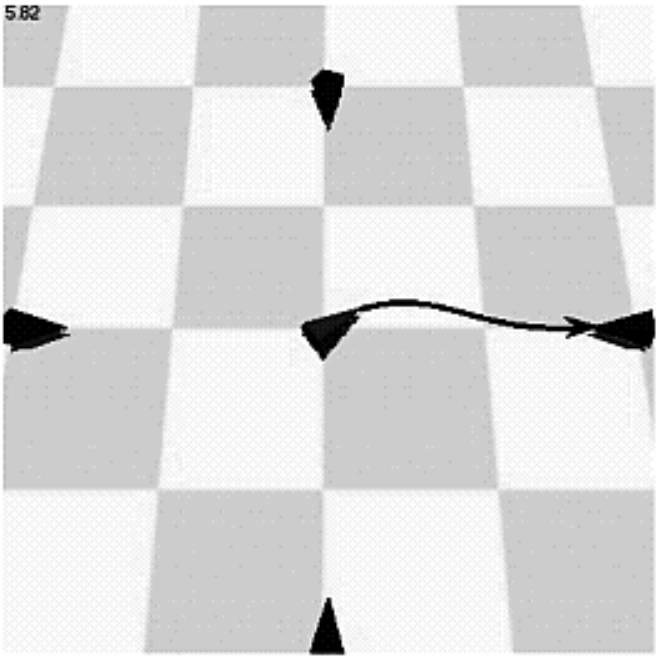
This game has six objectives. Each task has its own instance of the same three objectives: deal damage to the enemy, avoid damage from the enemy, and stay alive as long as possible. Damage dealt to the enemy is a group score shared by NPCs on a team. The damage avoidance and staying alive objectives are assessed individually, and the average across team members is assigned to the team.

Although damage received and time alive are both affected by taking damage, each one provides valuable feedback when the other does not: if all NPCs die, then time alive indicates how long each avoided death, but if no NPCs die, then damage received indicates which team performed better.

The need to be alternately offensive and defensive in each task and the large number of fitness objectives makes this game very challenging. The next game has fewer objectives, but is nonetheless challenging in its own right.



(a) Predator task



(b) Prey task

Fig. 2: Predator/Prey. Both the Predator and Prey tasks look the same. Fig. 2a has an arrow showing the movement path of the enemy in the Predator task: It tries to escape through the nearest gap between two NPCs. Fig. 2b shows how the enemy would behave in the same situation in the Prey task: It pursues the nearest NPC prey in front of it. Both situations are the same to the NPCs,

but because the environmental dynamics and enemy behavior are different, different behavior is needed to succeed.

C. Predator/Prey Game

In contrast to FBR, offensive and defensive behaviors are needed in separate tasks within this game. NPCs are either predators or prey depending on the task, and the enemy takes on the opposite role (Fig. 2). The dynamics of the environment and the behavior of the enemy change depending on the task.

In the Predator task, NPCs are predators and the enemy is prey (Fig. 2a). The enemy tries to escape by moving through a gap between two predators. When a predator hits the enemy, the enemy sustains damage and is flung away from its attacker. All agents move at the same speed, which means predators must avoid crowding the enemy, since hitting it can knock it so far away that it is impossible to catch. Therefore, evaluation ends prematurely if the enemy is no longer surrounded. This task is the same as the "Flight" task in [10], but the enemy's escaping behavior is more intelligent because it explicitly seeks gaps through which it can escape.

The Prey task reverses the dynamics of the Predator task, such that the enemy deals damage to NPCs, who are now the prey (Fig. 2b). This task is fairly simple since NPCs still start surrounding the enemy, and can avoid it by just running away. The enemy's behavior consists of moving forward towards

the closest NPC. Thus, the PP game is challenging because a single evolved controller must be able to execute essentially opposite behaviors depending on the task.

PP has three objectives. In the Predator task, the only objective is to maximize damage dealt to the enemy. This amount is shared across NPCs as in FBR. The Prey task has two objectives: minimize damage received, and maximize the time spent alive. As in FBR, these amounts are averaged across team members to get the team score.

As in FBR, the amount of damage per hit is 10 hit points, and all agents have 50 hit points. Furthermore, the domain is reset to starting conditions if the enemy dies (only possible in Predator task). The next section explains the evolutionary methods used to handle these games.

III. Evolutionary Methods

Evolutionary multiobjective optimization was used to handle the many objectives across tasks. The evolved individuals were neural networks, and special methods were used to evolve them for multitask games.

A. Evolutionary Multiobjective Optimization

Multitask games are by their very nature multiobjective, since at least one objective is needed in each task. The above domains have multiple objectives *per* task, which makes evolving in them even more challenging. Therefore a principled way of dealing with multiple objectives is needed.

The concepts of Pareto dominance and optimality provide such a framework[1]:

Pareto Dominance: Vector $\vec{v} = (v_1, \ldots, v_n)$ dominates $\vec{u} = (u_1, \ldots, u_n)$, i.e. $\vec{v} \succ \vec{u}$, iff

1. $\forall i \in \{1, \ldots, n\} : v_i \geq u_i$, and
2. $\exists i \in \{1, \ldots, n\} : v_i > u_i$.

Pareto Optimality: A set of points $\mathcal{A} \subseteq \mathcal{F}$ is Pareto optimal iff it contains all points such that $\forall \vec{x} \in \mathcal{A}$: $\neg \exists \vec{y} \in \mathcal{F}$ such that $\vec{y} \succ \vec{x}$. The points in $\mathcal{A}$ are non-dominated, and make up the non-dominated Pareto front of $\mathcal{F}$.

The above definitions indicate that one solution is better than (i.e. dominates) another solution if it is strictly better in at least one objective and no worse in the others. The best solutions are not dominated by any other solutions, and make up the Pareto front of the search space. The next best individuals are those that would be in a recalculated Pareto front if the actual Pareto front were removed first. Layers of Pareto fronts can be defined by successively removing the front and recalculating it for the remaining individuals. Solving a multiobjective optimization problem involves approximating the *first* Pareto front as best as possible; In this paper this goal is accomplished using the Non-Dominated Sorting Genetic Algorithm II (NSGA-II [3]).

NSGA-II uses $(\mu + \lambda)$ elitist selection favoring individuals in higher Pareto fronts over those in lower fronts. Within a given front, individuals that are more distant from others in objective space are favored by selection for the sake of

exploring diverse trade-offs.

Applying NSGA-II to a problem results in a population containing an approximation to the true Pareto front. This approximation set potentially contains multiple solutions, which must be analyzed in order to determine which solutions fulfill the needs of the user. However, NSGA-II is indifferent as to how these solutions are represented. For all domains in this paper, NSGA-II was used to evolve artificial neural networks to control the NPCs. The process of evolving these networks is called neuroevolution.

B. Neuroevolution

Neuroevolution is the application of evolution to neural networks. All evolved behavior in this paper was learned via constructive neuroevolution, meaning that networks start with

[1]These definitions assume a maximization problem. Objectives to be minimized can simply be multiplied by -1.

minimal structure and become more complex from mutations across several generations. The initial population consists of networks with no hidden layers, i.e. only input and output neurons. Furthermore, these networks are sparsely connected in a style similar to Feature Selective Neuro-Evolution of Augmenting Topologies (FS-NEAT [19]). Initializing the networks in this way allows them to ignore any inputs that are not, or at least not *yet*, useful.

Three mutation operators were used to change network behavior. Weight mutation perturbs the weights of existing network connections, link mutation adds new (potentially recurrent) connections between existing nodes, and node mutation splices new nodes along existing connections. Recurrent connections transmit signals that are not processed by the network until the following time step, which makes them particularly useful in partially observable domains. An environment is partially observable if the current observed state cannot be distinguished from other observed states without memory of past states [14]. Recurrent connections help in these situations because they encode and transmit memory of past states; a property that could help a network determine which of several tasks it currently faces.

These mutation operators are similar to those used in NEAT [13]. Though NEAT provides a method for performing crossover on these arbitrary topology networks, crossover is not used in this paper because preliminary experiments showed that it often had no effect, and in some cases even decreased performance.

The form of neuroevolution described so far has been used to solve many challenging problems [5, 8, 11, 13], but this approach does not directly target multitask domains. The next section describes some enhancements to neuroevolution for dealing with multitask domains.

C. Multitask Evolution

Two approaches to dealing with multitask games are evaluated: Multitask Learning, which translates work from [2] into a neuroevolution framework, and Mode Mutation, which was introduced in [10], but is enhanced in this work.

1) Multitask Learning: Multitask Learning assumes that evolving agents are always aware of the task they currently face. Each network controller is equipped with a complete set of output neurons per task (Fig. 3a). Therefore, if two outputs are required to define the behavior of an NPC in any task, and the NPC is required to solve two tasks, then the controlling networks would have two outputs for each task, for a total of four outputs. When performing a given task, the NPC bases its behavior on the outputs corresponding to the current task, and ignores the other outputs.

2) Mode Mutation: Mode Mutation does not provide NPCs with knowledge of the current task. It is a mutation operator that adds a new output mode to a network. As a result, networks can have many different output modes, often even exceeding the number of tasks in the domain.

There is no mode-to-task mapping, therefore a way of choosing a mode to define NPC behavior each time step is needed. Mode arbitration depends on output neurons called preference neurons. Each mode has one preference neuron in addition to enough neurons to define agent behavior, i.e. policy neurons. Every time step, the output mode whose preference neuron value is highest is the mode that defines agent behavior. So if two neurons are needed to define agent

behavior, Mode Mutation adds three neurons to the output layer: two policy neurons and one preference neuron. Two methods of Mode Mutation are evaluated.

The first method is Mode Mutation Previous (MM(P); Fig. 3c), the original version from [10]. Neurons for new modes start with one input synapse each. Each input comes from the corresponding neuron of the previous output mode. These connections are lateral, from left to right in the same layer, but are treated as feed-forward connections (they transmit on the same time step). The weights of these connections are set to 1.0, thus assuring that newly created modes start out similar to a preexisting mode, making them unlikely to cause a large drop in fitness. However, future mutations can differentiate the new mode from its source mode such that both modes exhibit distinct behavior.

However, such differentiation is not guaranteed to occur. The second method of Mode Mutation, new in this paper, was invented to address this problem. With Mode Mutation Random (MM(R); Fig. 3d), each neuron in a new mode receives one input with a random synaptic weight from a random source in either the hidden or input layer. This approach is risky since a new mode could cause fitness scores to plummet, but it has the advantage of more quickly introducing *distinct* modes of behavior.

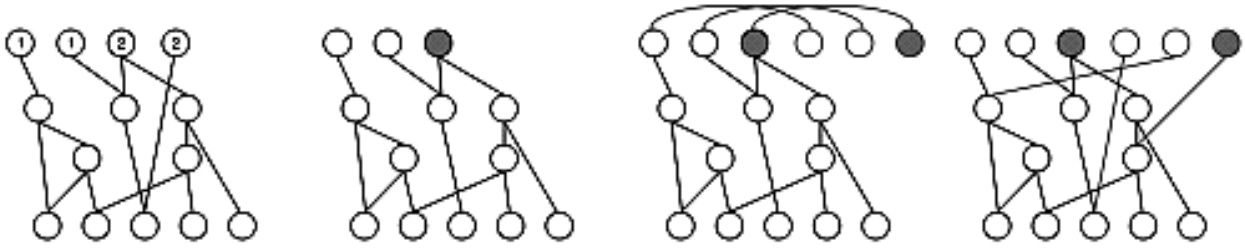
MM(R) also makes it feasible to delete output modes. Deleting a mode when using MM(P) is often infeasible, because the modes are tightly interconnected and a deletion

would often disconnect output modes from the network. However, modes can be safely deleted in MM(R) networks without modes becoming disconnected. In fact, preliminary experiments indicated that the ability to delete modes is very important to the success of MM(R), so whenever using MM(R), the following mode-deletion mutation is also used.

Throughout evaluation, the number of times each mode is used is tracked. If a mode-deletion mutation occurs on a network with multiple modes, then this data is used to choose for deletion the output mode that was used the least in the previous evaluation. If multiple modes are tied for least usage (usually meaning they were not used at all), then the oldest of these modes is deleted. This procedure removes unimportant, dead-end modes and allows the other mutation operators to focus on refining the remaining useful modes.

IV. Experiments

These approaches to solving multitask games were applied to the games from Section II. Experiments in both games were run in a similar manner. All experiments used constructive neuroevolution with a weight-mutation rate of 0.4, link-mutation rate of 0.2, and node-mutation rate of 0.1. These and other parameters used are similar to those used in previous works [9–11].



(a) Multitask network (b) One-mode network (c) Network after MM(P) (d) Network after MM(R)

Fig. 3: Networks for playing multitask games. Fig. 3a shows a multitask network with two modes, each consisting of two outputs. The network always knows which of the two tasks it is performing, and picks the appropriate outputs accordingly. Fig. 3b shows a network with only one output mode containing a grey preference neuron. Fig. 3c shows how this network would be modified by MM(P) to create a network whose new output mode receives inputs from the previous mode. The new lateral connections all have weights of 1.0 to assure similarity to the previous mode. Fig. 3d shows how the one-mode network would be modified by MM(R). In this case, the new mode is connected by randomly weighted synapses to random nodes in the hidden and input layers, thus making the new mode likely to be very different from preexisting modes. For both types of Mode Mutation, further mutations can change the behavior of these new modes, and result in the addition of more modes beyond the two shown. The benefit of these architectures is that they allow networks to have multiple policies for different tasks, while still allowing shared information about underlying task similarities in the hidden layer.

In the results below, `Control` represents networks in which a single output mode was used in both tasks of each game. `Multitask` represents networks with one mode for each task in the given game. These networks always knew which task they were facing, and used the appropriate output mode accordingly. Both Mode Mutation conditions, `MM(P)` and `MM(R)`, had initial populations containing networks with only a single mode each. New modes could be added by the appropriate Mode Mutation, whose rate was 0.1 in each case. These networks chose which mode to use based on the values of their preference neurons, as described in Section III-C2. Additionally, `MM(R)` made use of a mutation to delete the least-used output mode at a rate of 0.1.

NPCs were evolved 20 times for 500 generations for each experimental condition in each game. NSGA-II was used with a population size of $\mu = \lambda = 52$ to evolve neural network controllers. Each controller earned scores by being evaluated in multitask games. To earn scores for a single

task, a network was copied into each of the four members of a team of NPCs. Such homogeneous teams tend to be better at producing teamwork because the altruistic behavior of individuals is not punished if it contributes to greater team scores [18]. Because of random starting conditions, evaluations are noisy. Therefore every network was evaluated three times in each task, and their final scores in each objective were the averages across evaluations. The maximal evaluation time for each task was 600 time steps. Networks were evaluated separately in each task, and since the games in this paper consist of two tasks each, each network was evaluated a total of six times.

On each time step of the simulation, the enemy acts according to scripted behavior (Section II), and the evolving agents act according to their neural networks. On each time step, the NPCs' sensors provide inputs to the network, which are then processed to produce outputs, which then define the behavior of the NPC for the given time step.

In each game, evolving NPCs had the following 31 sensors: a constant bias, difference between NPC and enemy headings, angle between the NPC's current heading and enemy's location, brief signals whenever the NPC deals or receives damage, signals for when any teammate deals or receives damage, a sense for when the enemy is being knocked back from being hit, a sense for whether the NPC is in front of the enemy, differences in headings between the NPC and each of its teammates, angles between the

NPC heading and each of its teammate's locations, individual signals for when each teammate deals damage to the enemy, as well as an array of five ray sensors in front of the agent that provide different signals for when the enemy or other teammates are sensed. Though each team member is controlled by a copy of the same network, each member senses the environment differently, and can therefore take action in accordance with its particular circumstances. Additionally, each NPC's network has its own recurrent state corresponding to its history of senses and actions. The recurrent states of all NPCs are reset whenever the enemy respawns. This list may seem long, but recall that a feature selective approach [19] is used to evolve the networks, which allows for some of these inputs to be ignored or incorporated later if necessary.

In contrast to the long list of inputs, the list of outputs (per mode for multimodal approaches) is short: One output for the degree of forward vs. backward thrust, and another for left vs. right turning. However, complex behaviors can be produced from these outputs, as the results show.

V. Results

Performance in multiobjective domains is measured very differently from performance in single-objective domains. Therefore, methods for multiobjective performance assessment are discussed before moving on to the results.

A. *Multiobjective Performance Assessment*

A run of NSGA-II creates an approximation to the true Pareto front, i.e. an approximation set. Multiobjective performance metrics compare approximation sets from different runs. Individual objective scores and statistics based on them are misleading because high scores in one objective can be

combined with low scores in other objectives. Comparing approximation sets directly reveals whether one dominates another, but this approach does not scale to a large number of comparisons. Furthermore, if different approximation sets cover non-intersecting regions of objective space, it is still unclear which one is better. Multiobjective performance metrics help by reducing an approximation set to a single number that gives some indication of its quality.

The primary performance measure in this paper is the hypervolume indicator [21]. Hypervolume measures the region dominated by all points in an approximation set with reference to some point that is dominated by all points in the set. For example, if an approximation set consisted of a single solution with all positive scores, and the reference point were the zero vector, its hypervolume would be the product of all objective scores, i.e. the volume of the hypercube between the solution and the reference point. When more points are in the approximation set, hypervolume measures

the size of the union of the hypercubes between each solution and the reference point. The actual reference points used were (0,−50,0) for PP and (0,0,−50,−50,0,0) for FBR, where the zeroes are for the various damage dealt and time alive objectives, and each −50 is for one of the damage received objectives. Basically, each reference point was a vector of minimum scores for each objective.

Hypervolume is particularly useful because it is Pareto-compliant [21], meaning that an approximation set that completely dominates another approximation set will have a higher hypervolume. The opposite is not true: an approximation set with higher hypervolume does not necessarily dominate one with lower hypervolume, since each set could dominate non-intersecting regions of objective space.

In fact, it is provably impossible to construct a unary indicator that tells when one approximation set dominates another [22]. Despite this limitation, hypervolume is one of the best metrics available for multiobjective performance assessment. Other Pareto-compliant metrics are the multiplicative and additive unary epsilon indicators [7]. All hypervolume-based comparisons discussed below were also done using these epsilon indicators as well. The results of statistical tests using epsilon indicators tell the same story as tests performed using the hypervolume indicator. Therefore the analysis below will focus only on hypervolume.

B. Front/Back Ramming Results

The results for FBR conform to expectations of how the

different methods should perform: `Control` performed the worst, both `MM(P)` and `MM(R)` are better, and `Multitask` is the best. The hypervolumes (Fig. 4) support these conclusions. The differences in hypervolume between `Control` and both forms of Mode Mutation, as well as the differences between the Mode Mutation methods and `Multitask`, are significant ($p < 0.05$). There is no significant difference between the two forms of Mode Mutation.

The behaviors of NPCs from each condition are in line with these results. Movies of FBR behavior from each condition can be seen at http://nn.cs.utexas.edu/?multitask.

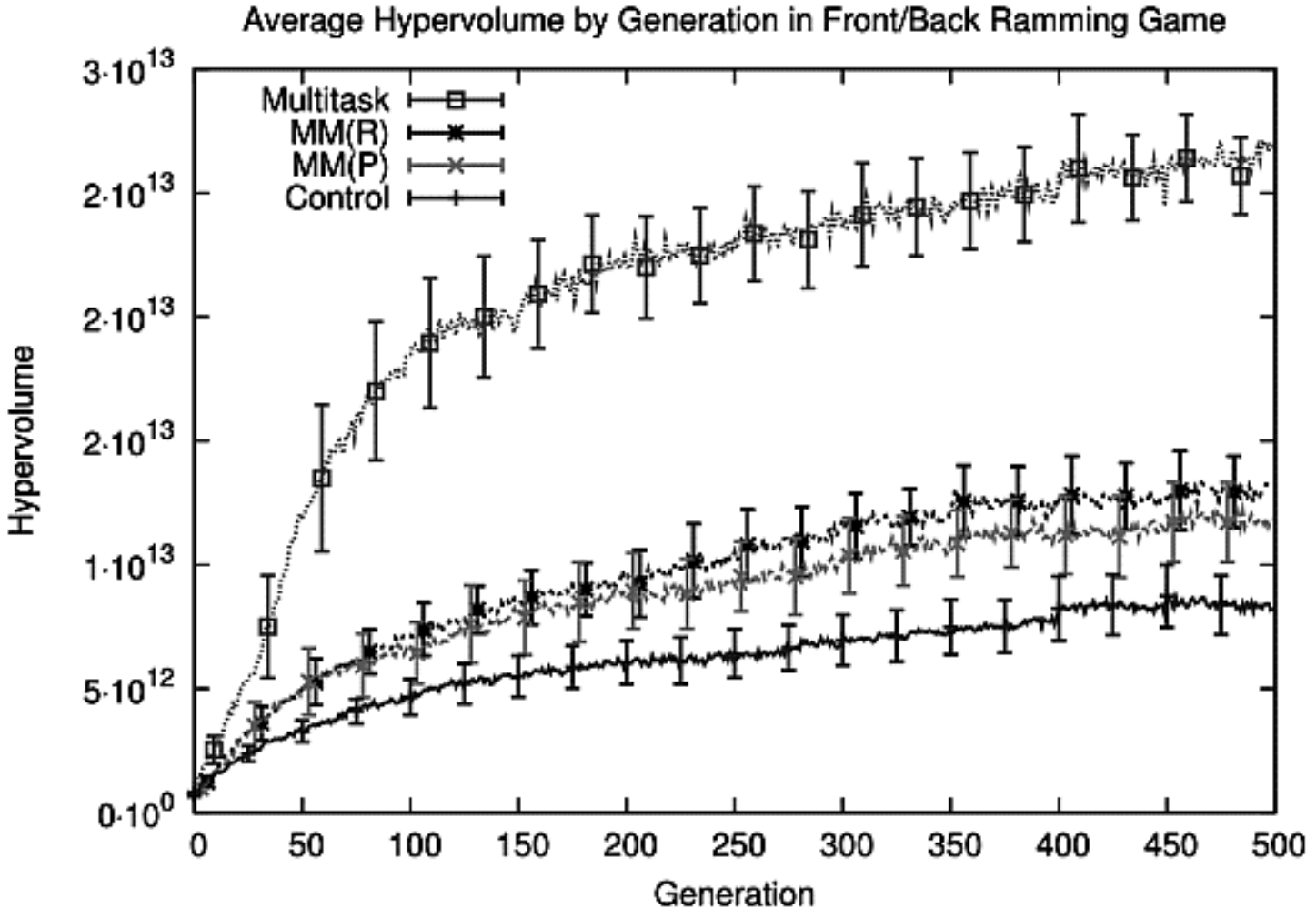


Fig. 4: Average hypervolumes in Front/Back Ramming game. For each experimental condition, average hypervolumes across 20 runs are shown by generation, along with the corresponding 95% confidence intervals. The figure indicates that `Multitask` quickly outpaces the other methods by achieving and maintaining approximation sets with significantly higher hypervolumes. Beneath `Multitask` are `MM(R)` and `MM(P)`, which are roughly equal to each other, but both significantly better than `Control`, thus demonstrating that multimodal networks have a significant advantage over generic neuroevolution in this multitask game.

In general, `Control` networks easily learn to perform well in one of the two tasks, but are rarely capable of performing both well. These networks often perform behavior that is successful for Front Ramming in the Back Ramming task (or vice versa), in which such behavior is detrimental.

In contrast, `Multitask` networks are almost always capable of performing both tasks well. Such behaviors are easy to learn since the networks have completely different policies for each task. In the Front Ramming task NPCs rush forward to ram the enemy, and in Back Ramming the same NPCs immediately turn around at the start of the trial so they can attack the enemy with the rams on their rears.

Mode Mutation networks, though lacking information available to `Multitask` networks, are significantly different from `Control` networks in an important way: they are capable of solving both tasks instead of just one. However, since Mode Mutation networks need to overcome the chal-

lenge of not knowing which task they are facing, their scores tend to be lower than `Multitask` networks.

Though in terms of performance metrics there is no significant difference between `MM(P)` and `MM(R)`, observation of evolved behaviors indicates that `MM(R)` networks tend to more clearly associate particular modes with particular behaviors, i.e. `MM(R)` behaviors are more transparent. `MM(P)` mode usage is often confusing, in that more thrashing between modes occurs, or multiple behaviors seem to be exhibited by a single mode. This confusion is likely caused by the interconnectedness of `MM(P)` modes; since each mode leads into the next, the behavior of a given mode might actually be more representative of one of the modes that precedes it. The resulting networks usually have the majority of hidden-layer connections leading into the oldest output

Average Hypervolume by Generation in Predator/Prey Game

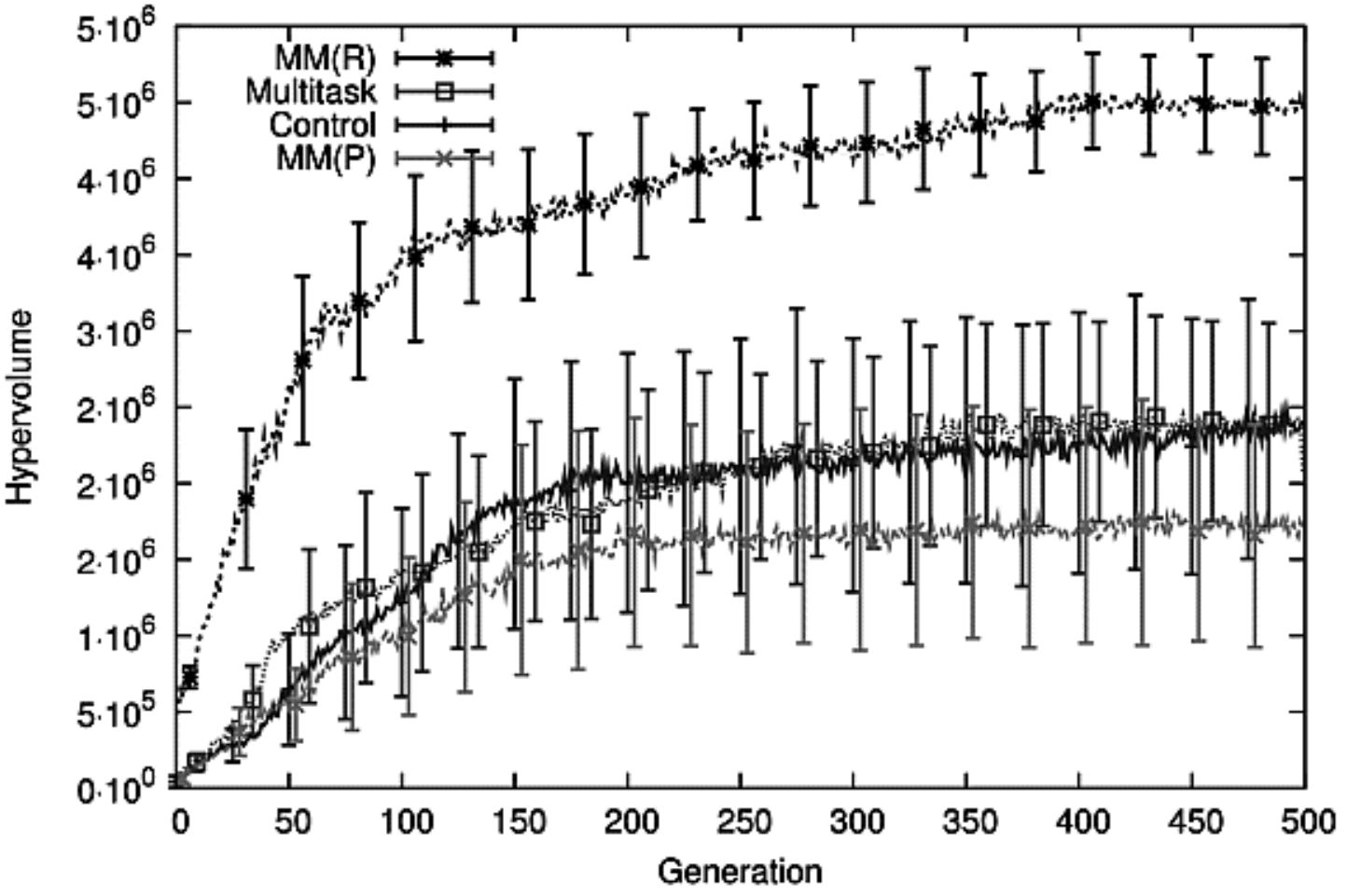


Fig. 5: Average hypervolumes in Predator/Prey game. For each experimental condition, average hypervolumes across 20 runs are shown by generation, along with the corresponding 95% confidence intervals. In contrast to the FBR task, `MM(R)` outperforms all other methods. Those other methods are roughly equal with overlapping confidence intervals, with `MM(P)` slightly below `Multitask` and `Control`. This domain demonstrates how evolving a task division with `MM(R)` outperforms the obvious task division used by `Multitask` and the lack of a task division used by `Control`.

modes, even though there are usually several newer output modes in the network as well (their only connection to the network is through previous modes).

One method that Mode Mutation networks found of overcoming their lack of task awareness is to start by turning the

NPC backs towards the enemy. In the Back Ramming task, this strategy is effective. In the Front Ramming task, this behavior will cause the NPCs to be hit, but this hit does two things: 1) when hit, NPCs are flung backwards with their front ram facing the enemy, and 2) NPCs sense being hit, and as a result switch network modes so they now attack with their front rams. Preference for the new attack mode is maintained by internal recurrent state. This multimodal behavior is a good example of how Mode Mutation networks can learn to overcome the challenges of a multitask game.

Though the results in FBR make sense given the resources and information available to each method, these results are clearly domain-dependent, as demonstrated via contrast with the results from PP, explained next.

C. Predator/Prey Results

The results in PP are unexpected, in that neither `Multitask` nor `MM(P)` performs better than `Control`, but `MM(R)` greatly outperforms all of these conditions. The hypervolumes (Fig. 5) support this conclusion: `MM(R)` is significantly better than all other conditions ($p < 0.05$).

The insights gleaned from the hypervolume values are further supported by observing the evolved behaviors of the NPCs. Movies of behavior from each condition can be seen at http://nn.cs.utexas.edu/?multitask.

`Control` networks tend to be good in only one of the two tasks, but because the Prey task is so easy, there are also `Control` networks that are successful at both tasks.

The Predator task is the more challenging task. Sometimes NPCs that take damage and die in the Prey task make it into the Pareto front because they deal a large amount of damage in the Predator task.

What is surprising is that `Multitask` networks do not do better in the Predator task. `Multitask` networks *always* master the Prey task because they start running from the Predator as soon as evaluation starts; not a single individual in any of the 20 Pareto fronts for `Multitask` networks in PP fails to get perfect scores in the Prey task. It is easy for `Multitask` networks to have one policy that makes the NPCs run away. However, it is unclear why `Multitask` networks do not always do well in the Predator task.

A possible explanation is that giving equal attention to each task, as `Multitask` networks do as a result of their architecture, is unnecessary and even detrimental in this game, because the relative challenge of the two tasks is so different. Good Prey behavior thus becomes over-optimized at the expense of good Predator behavior.

This trade-off in evolutionary search might also explain why `MM(R)` does so well: Evolution with Mode Mutation chooses how many modes to make, and how often to use each of them. This result indicates that the "obvious" task division may hinder evolution, but `MM(R)` overcomes this problem by finding its own task division.

However, this conclusion does not explain why `MM(P)` behavior is so erratic; sometimes mediocre in both tasks, and

sometimes spectacular in both tasks. Success with MM(P) seems to depend strongly on luck in this domain. When MM(P) succeeds, it tends to use few of its modes. It seems that the interconnectedness and similarity of MM(P)'s output modes make it difficult for networks to specialize modes for either task, so success for MM(P) mainly comes about when multiple modes are ignored. Successful MM(R) networks often, though not always, use only one mode as well.

Since the few quality MM(P) networks and the many quality MM(R) networks tend to favor only one mode, perhaps one mode is the ideal number for this game. Then why does MM(R) do so well? The mode-deletion mutation is likely the key. If a single quality mode is all that is necessary, then MM(R) is ideal because it both creates new, novel modes via mode mutation and deletes pointless, unused modes via mode deletion. In other words, MM(R) helps evolution find the right one mode for this game. In fact, modes found early on can serve as crutches until better modes are found, at which point the old modes are deleted. Switching behavior in this way is easier for evolution than incrementally changing the behavior of existing modes, as in the Control and Multitask cases.

VI. Discussion and Future Work

Interestingly, although Multitask Learning and Mode Mutation generally work well in the multitask games of this paper, results were very different for the two games. In order

to best exploit these methods in more challenging games with less extreme task divisions, some idea of when a given method will be successful is needed.

First, `Multitask` is restricted by needing to know the current task, whereas `MM(R)` is not. Since `MM(R)` does great in PP and better than `Control` in FBR, it is the ideal choice for multitask games in which the task division is not known. However, since games are artificial environments, programmers will usually be able to tell agents what the current task is. Therefore, in these cases the `Multitask` approach could be used.

However, Multitask Learning is only powerful when the task division *properly* splits the challenges of the game. As was shown by the PP game, when tasks are not equally difficult, separate dedicated modes may actually be detrimental to evolution. In such cases it is better to let `MM(R)` discover a good division into modes. Furthermore, `MM(R)` may also be applied to games where the task division is more dynamic or overlapping. For instance, `MM(R)` should work well even when the agents choose which task they perform, as in the Unreal Tournament example discussed in Section II.

`MM(R)` could be further improved by controlling bloat more intelligently, while still allowing new modes to take hold in the network. The delete-mode mutation helps control bloat, but `MM(R)` networks still contain unused modes. In

this paper, a Mode Mutation rate equal to the mode-deletion rate was used, which may not do a good enough job of pruning seldom used modes. Furthermore, it may be the case that new modes need some protection from deletion for a certain number of generations after being created. Exploring these and other ways of improving Mode Mutation is an interesting direction for future work.

VII. Conclusion

Two multitask games, Front/Back Ramming and Predator/Prey, were used to evaluate two methods of evolving multimodal networks: Multitask Learning and Mode Mutation.

Multitask Learning uses knowledge of the current task to pick which of a set number of output modes will control an NPC for an entire task. Mode Mutation, in contrast, does not know about the current task. Instead, it discovers a suitable task division by adding new output modes to a network, any of which can be used on any given time step depending on preference-node values.

In the Front/Back Ramming game, Where the task division is both obvious *and* balanced, Multitask Learning is the most effective, and Mode Mutation is second best. Both approaches are better than using networks with just one mode. In Predator/Prey a form of Mode Mutation, named MM(R), proved to be the most effective method by discovering a task division that is not obvious to a human designer. Multitask Learning and Mode Mutation thus allow evolving agents

to have multiple policies to fit different situations, which will make these approaches useful in developing intelligent behaviors for challenging games consisting of multiple tasks.

ACKNOWLEDGMENT

This research was supported in part by the NSF under grants DBI-0939454 and IIS-0915038 and Texas Higher Education Coordinating Board grant 003658-0036-2007.

REFERENCES

[1] D. Andre and A. Teller. Evolving Team Darwin United. In *RoboCup-98: Robot Soccer World Cup II*, pages 346–351. Springer Verlag, 1999.

[2] R. A. Caruana. *Multitask Learning*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 1997.

[3] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *PPSN VI*, pages 849–858, 2000.

[4] T. D'Silva, R. Janik, M. Chrien, K. Stanley, and R. Miikkulainen. Retaining learned behavior during real-time neuroevolution. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2005.

[5] F. Gomez and R. Miikkulainen. Active guidance for a finless rocket using neuroevolution. In *Genetic and Evolutionary Computation Conference*, 2003.

[6] J. Klein. BREVE: A 3D Environment for the Simulation of Decentralized Systems and Artificial Life. *ALife*, 2003.

[7] J. Knowles, L. Thiele, and E. Zitzler. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, TIK, ETH Zurich, Feb. 2006.

[8] N. Kohl and R. Miikkulainen. Evolving neural networks for strategic decision-making problems. *Neural Networks, Special*

issue on Goal-Directed Neural Systems, 2009.

[9] J. Schrum and R. Miikkulainen. Constructing complex NPC behavior via multi-objective neuroevolution. In *Artificial Intelligence and Interactive Digital Entertainment*, 2008.

[10] J. Schrum and R. Miikkulainen. Evolving Multi-modal Behavior in NPCs. In *Computational Intelligence and Games*, 2009.

[11] J. Schrum and R. Miikkulainen. Evolving agent behavior in multiobjective domains using fitness-based shaping. In *Genetic and Evolutionary Computation Conference*, July 2010.

[12] K. O. Stanley, B. D. Bryant, I. Karpov, and R. Miikkulainen. Real-Time Evolution of Neural Networks in the NERO Video Game. In *National Conference on Artificial Intelligence*, 2006.

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

[14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[15] T. Thompson, F. Milne, A. Andrew, and J. Levine. Improving Control Through Subsumption in the EvoTanks Domain. In *Computational Intelligence and Games*, pages 363–370, 2009.

[16] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber. Super Mario Evolution. In *Computational Intelligence and Games*, 2009.

[17] N. van Hoorn, J. Togelius, and J. Schmidhuber. Hierarchical Controller Learning in a First-Person Shooter. In *Computational Intelligence and Games*, 2009.

[18] M. Waibel, L. Keller, and D. Floreano. Genetic Team Composition and Level of Selection in the Evolution of Multi-Agent Systems. *Evolutionary Computation*, 13(3), 2009.

[19] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl. Automatic feature selection in neuroevolution. In *Genetic and Evolutionary Computation Conference*, 2005.

[20] G. N. Yannakakis and J. Hallam. Interactive opponents generate interesting games. In *Computer Games: Artificial Intelligence, Design and Education*, pages 240–247, 2004.

[21] E. Zitzler, D. Brockhoff, and L. Thiele. The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration. In *Evolutionary Multi-Criterion Optimization*, 2007.

[22] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. TIK Report 139, TIK, ETH Zurich, 2002.