Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Survey of evolutionary learning for generating agent controllers in synthetic environment

A. Taylor

Defence R&D Canada – Ottawa

Technical Memorandum
DRDC Ottawa TM 2011-212
December 2011

Canada

Survey of evolutionary learning for generating agent controllers in synthetic environments

A. Taylor

Defence R&D Canada – Ottawa

Technical Memorandum

DRDC Ottawa TM 2011-212

December 2011

Principal Author

Original signed by Adrian Taylor

Adrian Taylor

Approved by

Original signed by for Julie Tremblay-Lutter

Julie Tremblay-Lutter
Section Head, CARDS

Approved for release by

Original signed by for Chris McMillan

Chris McMillan
Chief Scientist, DRDC Ottawa

Abstract

Using evolutionary methods is a promising approach to overcoming limitations in developing traditional artificial intelligence (AI) controllers in synthetic environments. Traditional AI is difficult and expensive to create and maintain. As a result exercises and training simulations often require the participation of human players who exist solely to provide believable and realistic behaviour for allied, enemy, and neutral forces. Here we review recent results in evolutionary learning for achieving two goals: reducing the difficulty of creating AI, and creating highly capable and robust AI. Results are organized first according to their architectures; these range from hand-designed controllers with evolved parameters to neural networks that grow in complexity. Second we compare methods of guiding the evolutionary search process, necessary because of the large size of the controller search space. These methods, such as modularization and multiobjective evolution, help to reduce the designer's workload. We explain the need to identify practical applications for evolutionary methods by examining shortcomings in current client use of AI, and outline a research plan for developing evolutionary methods for application to these client requirements. Successful application of these methods will generate better AI, reduce cost, and reduce the human workload in executing distributed simulation exercises.

Résumé

Les méthodes évolutives offrent une démarche qui promet de surmonter les limites du développement de contrôleurs d'intelligence artificielle (IA) dans des environnements synthétiques. La création et le maintien traditionnels d'IA sont difficiles et dispendieux. Par conséquent, les exercices et les formations simulées exigent souvent une participation humaine pour assurer un comportement plausible et réaliste de la part des forces alliées, ennemies et neutres. Nous examinons des résultats récents en apprentissage évolutif pour étudier des méthodes pour simplifier la création d'AI et créer des IA plus complexes, fonctionnelles et robustes. Nous examinons aussi l'utilisation de fonctions de recherche dans diverses architectures d'AI, allant du réglage de contrôleurs conçus en grande partie manuellement aux contrôleurs dont la structure évolue pour atteindre une complexité suffisante. Nous comparons ensuite les méthodes pour guider le processus de recherche de manière à gérer de manière modulaire la complexité de l'espace de recherche du contrôleur et à réduire la charge de travail du concepteur au moyen d'une évolution à objectifs multiples. Nous expliquons le besoin d'identifier les applications pratiques des méthodes évolutives en examinant des lacunes dans l'utilisation actuelle de l'IA pour notre client, et nous traçons un plan de recherche pour le développement de méthodes évolutives applicables aux exigences du client. Une application réussie de ces méthodes produira de meilleures IA à moindre coût et réduira la charge de travail humaine requise dans l'exécution des exercices de simulation distribués.

This page intentionally left blank.

Executive summary

Survey of evolutionary learning for generating agent controllers in synthetic environments

A. Taylor; DRDC Ottawa TM 2011-212; Defence R&D Canada – Ottawa; December 2011.

Synthetic Environments (SEs) are increasingly being used by the Canadian Forces and its allies to conduct training and exercises. These environments are populated by a mixture of humans and computer-controlled agents. The computer agents are governed by artificial intelligence controllers – programs that define their behaviour according to how they sense their environment. However the agents used in current SEs are generally unable to behave with sufficient complexity and realism to be employed without human supervision. Furthermore, traditional methods of developing AI are expensive, and customizing AI for each separate engagement is prohibitively so. As a remedy, we seek to use novel methods in evolutionary learning to automatically generate AI. We have two goals: to reduce developer effort, and to produce AI that is capable of realistic and robust behaviour in uncertain environments. To support this effort, here we review recent results in the literature on computational intelligence in games and evolutionary robotics.

The core idea of evolutionary learning is to use a search algorithm to find parameters for an AI controller. Though any search algorithm can be used, in practice genetic algorithms are almost universally the preferred method because they are easy to implement and understand, and are effective at nonlinear search. The search is guided by feedback from the environment and/or designer in the form of fitness functions. We organize this review in two parts. First we categorize approaches by the amount of structure embedded in the controller. The controller's structure influences the size of the space to be searched. A rigid structure has few parameters, resulting in a small search space. A more flexible structures, such as a genetic program, produces a much larger search space. As the search space becomes larger, the search requires more guidance from the designer. The need for guidance leads to the second part of our review, in which we categorize methods of guiding the search. The search can be guided by different forms of feedback (fitness), or by modularizing the controller, or using multi-objective search. Fitness functions are usually based on measures of achieving some end goal, but other methods such as the imitation of human performances or pure novelty have also been used with some success. Modularizing the controller limits the size of the search space at the expense of both flexibility and the designer's workload. Multi-objective evolution provides the benefits of modularizing controllers less additional effort, and is the most promising approach. This progression of methods approaches two goals: architecting controllers with sufficient complexity to produce behaviour equal to the demands of the problem space, and finding ways of guiding the genetic algorithm through the large search space created by the complexity of such controllers.

Results in the literature using these methods are mixed. Some of the simpler methods reported here, in which hand-designed controllers are optimized with evolutionary methods,

produced controllers comparable to those optimized by humans. More ambitious methods such as genetic programming are promising but difficult to manage. However human comparable performance with genetic programming has been reported in other problem domains. Also, the complexity of controllers reviewed here is limited, but new ideas in machine learning such as deep learning neural networks may provide for more capable solutions. We recommend investigation into generating complex controllers using multiobjective search with evolving neural networks and genetic programming. Also, we will need near-term targets with which to apply these methods. While it is well-known that the use of AI has been limited in CF exercises, it will be helpful to identify specific flaws that are limiting their use. Thus we recommend investigation into the CF's execution of exercises to identify areas that can benefit from more effective controllers. In the near-term, applying evolutionary methods to optimize existing controllers may provide useful initial results. It is hoped that in the longer term, these methods will produce more effective controllers for agents in synthetic environments. These agents will reduce the human workload in distributed simulation, reducing cost and enabling new applications for wargaming.

Sommaire

Survey of evolutionary learning for generating agent controllers in synthetic environments

A. Taylor ; DRDC Ottawa TM 2011-212 ; R & D pour la défense Canada – Ottawa ; décembre 2011.

Les Forces canadiennes et leurs alliés utilisent de plus en plus les environnements synthétiques (ES) pour effectuer des formations et des exercices. Ces environnements sont peuplés par un mélange d'agents humains et informatiques. Les agents informatiques sont gérés par des contrôleurs d'intelligence artificielle : des programmes qui définissent leur comportement en fonction de leur perception de leur environnement. Toutefois, les agents utilisés dans les ES actuels ne se comportent généralement pas de manière assez complexe et réaliste pour pouvoir se passer d'une supervision humaine. De plus, les méthodes traditionnelles de développement d'IA sont coûteuses, et la personnalisation d'IA pour chaque engagement entraîne des coûts exorbitants. Nous tentons d'utiliser des méthodes novatrices d'apprentissage évolutif pour automatiquement générer des IA en simplifiant la tâche des développeurs et pour produire des IA qui peuvent adopter un comportement complexe et robuste dans des environnements incertains. À cette fin, nous examinons des résultats récents dans la littérature sur l'intelligence informatique relatifs aux jeux et à la robotique évolutive pour évaluer des méthodes qui permettraient d'atteindre ces objectifs.

Le concept central de l'apprentissage évolutif est l'utilisation d'un algorithme de recherche pour établir les paramètres d'un contrôleur d'IA. Bien qu'on puisse utiliser n'importe quel algorithme de recherche, en pratique les algorithmes génétiques sont la méthode préférée dans presque tous les cas puisqu'ils sont faciles à appliquer et effectuent les recherches non linéaires de manière efficace. La recherche est guidée par les rétroactions de l'environnement et du concepteur au moyen de fonctions d'adéquation. Nous classons les démarches en premier lieu selon la quantité de structure intégrée dans le contrôleur, puisque cette structure limite le nombre de paramètres libres qui définissent l'espace de recherche et en restreint la taille et la complexité. Nous examinons ensuite les méthodes pour guider la recherche, premièrement par type de rétroaction, puis par modularisation et par recherche à objectifs multiples. Ont été examinés des contrôleurs très structurés avec un nombre limité de paramètres optimisés, des réseaux neuronaux et des programmes génétiques construits au moyen de l'évolution artificielle. Un concept complémentaire lié à chacune de ces démarches est l'organisation des contrôleurs dans des hiérarchies ou des structures semblables en fonction des sous-tâches. L'évolution séparée de chaque module réduit leur espace de recherche et aide à créer des contrôleurs utiles, mais cela réduit l'adaptabilité de l'algorithme et sa capacité de trouver des solutions novatrices et augmente la quantité de travail et les coûts de conception. Les fonctions d'adéquation sont généralement fondées sur le succès dans un environnement de jeu ou sur la survie, mais d'autres méthodes (imitation du comportement humain, nouveauté) sont aussi été utilisées avec plus ou mois de succès. L'évolution à objectifs multiples, qui vise plus d'un objectif simultanément, fournit certains avantages des contrôleurs modulaires avec un effort réduit. La progression de ces démarches peut

être considérée comme la poursuite d'un seul ou des deux buts suivants : concevoir des contrôleurs suffisamment complexes pour produire un comportement qui correspond aux exigences de la problématique et déterminer comment guider l'algorithme génétique dans le vaste espace de recherche créé par la complexité de ces contrôleurs.

La littérature indique que les résultats obtenus au moyen de ces méthodes sont variables. Néanmoins, dans certains domaines, on a obtenu un comportement comparable à celui d'un humain au moyen de la programmation génétique. De plus, la complexité des contrôleurs examinés est limitée. Par contre, de nouvelles idées dans le domaine de l'apprentissage automatique, par exemple, l'apprentissage en profondeur, peuvent donner de meilleurs résultats. Nous recommandons d'enquêter sur ces concepts et sur la quantification de la complexité des contrôleurs et des environnements pour contribuer à l'évolution du domaine. Nous devrons aussi fixer des cibles à court terme pour l'application de ces méthodes. Nous savons que l'utilisation de l'IA dans les exercices des FC a été limitée, mais les faiblesses des contrôleurs qui limitent leur utilisation ne sont pas évidentes. Nous recommandons d'étudier l'exécution des exercices des FC pour cerner les secteurs dans lesquels nous pourrions tirer profit de l'apprentissage évolutif. À court terme, l'optimisation des contrôleurs existants peut être suffisante pour obtenir des résultats utiles. Nous espérons que, à plus long terme, ces méthodes produiront des contrôleurs plus efficaces qui permettront l'utilisation plus fréquente et plus productive d'environnements synthétiques.

Table of contents

1 Introduction

The use of distributed simulation using Synthetic Environments (SE) is becoming a cornerstone of training and experimentation in the Canadian Forces (CF). These synthetic environments are inhabited by both humans and non player characters, or agents. Agents are usually provided by Computer Generated Forces (CGFs) like OneSAF or JSAF. These software tools simulate agents, in a complex environment, with behaviour governed by Artificial Intelligence (AI) controllers. The AI controller is a computer program, specified at the abstraction level of the SE, designed to respond to all anticipated situations. This means that every possible situation must be considered in advance for the AI to be effective. In addition, agents must behave effectively and appropriately for the context – otherwise human trainees may learn the wrong lessons. For example, in an combat scenario involving an unorganized militia, the agents should not perform with the coordination and skill of soldiers. In addition, the agent must follow the script written by the scenario designers.

In practice, accounting for these factors and generating agent AI requires special expertise. But the CF does not have the resources to support the development of AI specific to each scenario. Instead they use humans to oversee the nonplayer agents. These participants are called pucksters, named after the soldiers responsible for moving game pieces around the table in traditional wargames. Their role is to ensure that the exercise or training simulation goes according to plan. But accepting simple AI and relying on humans to run exercises limits the potential of simulation. Simulation offers opportunities for exploration not possible in real world exercises, since it can in theory accommodate deviations from the intended course leading to learning and discovery not necessarily foreseen by the scenario designer [1]. Here we are motivated by two goals: making AI easier to develop, and making AI more robust. In support of this, we review methods for automatically generating agent controllers using evolutionary learning. Success application of these methods will reduce the need for pucksters in exercises, allow exploration of more advanced uses of simulation.

1.1 AI in commercial and serious games

Traditional development of AI in synthetic environments is both a design and programming problem. The designer's job is to specify the desired behaviour. The programmer supports the designer by implementing the behaviours directly, or providing a controller framework for the designer to use directly. Programmers have, over the years, explored various frameworks in hopes of improving the robustness and complexity of agent behaviour. Early methods included finite state machines and scripts. More recently, more expressive methods such as Behaviour Trees and Hierarchical Task Networks have grown in popularity [2]. Other aspects of AI, such as the use of A* search with navigation meshes for pathfinding, have become nearly standardized. However none provide revolutionary improvement in performance or reduction in development workload. Additionally, the military client has different constraints than the game industry. The goal of the AI in games is create an entertaining and empowering experience for the player. This means the AI is designed to present an interesting but surmountable challenge. It is not difficult to design a controller that fulfills these goals [3]. And in some applications, this might be appropriate for CF needs. For

example in training, where the focus is on learning a specific skill, the simulation will reward success at its practice and temporarily ignore failures outside of the trainee's control. But more often the military needs AI that behaves competently and realistically, something much different from entertainment AI.

More sophisticated AI capable of reducing human workload is achievable but very expensive. One expert system called TacAir Soar generates realistic pilot agents. The TacAir Soar agents are capable of responding to standard orders from flight controllers, working in a team, and reacting appropriately in combat situations [4]. The system is sufficiently robust to require just two humans to oversee the behaviour of 100 AI pilots. However, developing a TacAir Soar was a very large project. Its behaviour was designed following extensive interviews with pilots. Their responses were distilled into 5200 rules, 450 operators, and 130 goals encoded in the Soar cognitive architecture. Debugging and understanding the choices made by the AI required the creation of a tool to provide visibility into its internal state [5]. Given the current CF simulation exercise tempo, building and maintaining such controllers for every possible role in a simulation exercise is probably more expensive than employing humans to play those roles for individual trials. However this limits the execution of scenarios by the availability of trained operators.

Simulation AI research at DRDC has pursued traditional methods. Scientists working on the Virtual Maritime Systems Architecture at DRDC Atlantic, dissatisfied with the AI available in the CGF software JCATS, investigated the use of agent architectures as a replacement [6]. While their new architecture streamlined development, they were still unable to significantly reduce the workload of developing behaviours for all the anticipated situations in their simulation, and generating realistic behaviour for its background actors. The CAMX system is a DRDC CORA project that provides realistic behaviour for civilian entities in distributed simulations [7]. Using simple phenomenological models for civilian behaviour, it provides realistic background civilian traffic for synthetic environments. The designer can control behaviour by defining geographic regions – e.g. exclusion zones, or sinks to guide traffic. The agents follow these goals and also react realistically to dangerous situations. It has been highly successful in addressing that gap, but it has a narrow focus, and its creation was motivated by the high cost of commercial CGFs offering similar capabilities. DRDC Ottawa has also in recent years been investigating shortcomings in AI in CGFs [8]. Part of this research has focused on developing reusable and portable AI controllers. A key finding was that learning was not exploited in commercial, government, and gaming simulators.

1.2 Evolutionary design

Evolutionary design has the potential to create systems of greater complexity and robustness. It is a method of learning how to behave, using feedback from the environment and the human designer, without specifying of details of its implementation. Evolutionary search methods such as genetic algorithms and genetic programming have been around for decades. But recently they have been employed to create robot and game AI controllers in synthetic environments. For evolutionary robotics, synthetic environments offer a simplified environ-

ment with sufficient complexity to generate interesting behaviour, without the difficulty and expense of constructing robots or interpreting high-bandwidth sensors [9]. For game AI researchers, evolutionary learning is a way to create interesting and varied behaviour, and a tool for exploring search methods and metrics for automatically generated programs. We intend to leverage this work for military synthetic environments. What follows is a broad review of results in evolutionary design. We approach this from two directions. First we review controller architectures used in evolutionary design. Second we review methods of guiding the search process: different types of fitness feedback measures, modularization, and multiobjective search.

We begin with a short description of the general template for research using evolutionary design to generate controllers. In general, the basic problem is to automatically generate a controller. This effectively means searching the space of possible controllers to find good ones. Controllers may be highly constrained, in which case the search is optimizing parameter choices, or very loosely specified, in which case the search space may be very large. The controller is embodied in a synthetic environment. This could be a simple arcade game, or a rich simulation of the natural world. The search process requires some feedback measure to determine whether the evolved controller is successful. And the search process itself has many variations. Thus an experiment must define the controller architecture, environment, fitness, and search method:

- The environment: e.g. a classic game such as Pac-Man, a driving simulator, or a simple robot foraging task.
- The controller: a fixed architecture like a Finite State Machine (FSM), or an evolving one like a genetic program.
- The fitness measure: e.g. points scored, survival time, speed of the race, or an aggregate of these.
- The search method: almost always a genetic algorithm, but possibly with variations such as incremental learning.

Almost every method reviewed here uses a Genetic Algorithm (GA) to search. A GA is a method of searching inspired by biological evolution [10]. GAs operate on a genome; here the genome is usually a string of characters representing parameters of the controller. A collection of genomes is called a population. The algorithm operates on generations of populations. In the first generation, genomes for each individual are randomly generated. In this and subsequent generations, each genome is tested for fitness. In this context fitness is determined by measuring the agent's performance in a synthetic environment, e.g. rewarding it for inflicting damage on an opponent. The next generation is generated from the previous by combining pairs of genomes. For each offspring, two parents are selected with probability proportional to their fitness. Then with some crossover probability their genomes are swapped at a crossover point. Then with some small mutation probability each gene is randomly modified. The process is repeated until some criteria is met, for example the best or mean population fitness stops improving, or the population converges to a single genotype.

1.3 From simple games to serious simulations

Why focus on games? Most of the examples here mostly come from research into AI for commercial games. The academic community has investigated learning in games and it is thus a rich source of research results. Games are of interest because they have a number of practical benefits making them a good venue for research. They offer a simpler environment than the real world, but one that is by design compelling to humans. This is helpful for learning AI. It is easy to evaluate success, the world has already been abstracted, and competition – a vital part of evolution – is already built-in. Also it provides a venue for testing the results of the AI against human players. Games also can provide a wealth of player recorded data to train AI [11]. Some of the games described herein are very simple. Even these games are useful because they are easy to understand, and as such are a test bed for developing learning techniques that can then be directly applied to more sophisticated environments, e.g. serious games such as VBS2[1]. The CF currently uses VBS2 for land exercises and experiments.

Throughout, we assume basic familiarity with game genres and conventions.

[1]So-called serious games are SEs designed for purposes other than entertainment, such as training.

2 Controller Architectures

The architecture of the controller is fundamental to its performance and capabilities. A controller's design may contain implicit information about the problem it is solving. There are many different approaches. For the purposes of evolutionary design, we choose to divide controller architectures into two groups: fixed and evolving. Fixed controllers have a finite number of parameters that are tuned with the genetic algorithm. This approach can be fast and effective if the designer has enough foreknowledge to choose the right architecture. Evolving architectures are themselves modified by the genetic algorithm. These are theoretically more capable, but more difficult to develop because they create a much larger possibility space to search.

2.1 Fixed Controller Architectures

Search problems increase in difficulty with the number of parameters to be optimized. So the simplest application of search to control algorithms is writing the controller by hand, and optimizing a small number of its parameters. Here we consider three categories of fixed architectures. In the simplest, optimizing hand-designed controllers, the designer specifies all but a few parameters. With direct controller architectures, action-responses are encoded directly in a genome. Finally with fixed-topology neural networks, genetic algorithms are used to train neural networks to act as reactive controllers.

2.1.1 Optimizing Hand-Designed Controllers

Here we look at controllers that are largely designed by hand and then optimized using genetic algorithms. This is the simplest use of evolutionary design in this context. It may also be the most common, especially in production environments. As we will see it is powerful enough to produce human-comparable results with less effort than designing by hand.

Pac-Man is a classic arcade game that has become a popular platform for comparing AI controllers [12]. Gallagher designed a controller for Pac-Man game that was tuned using a GA algorithm called population-based incremental learning [13]. They used a simplified version of the game; Pac-Man must contend with only one ghost instead of four, and has no power pellets. The authors wrote a controller with two states: free roaming and flee. Its decision to switch the state is based on relative location of the ghost, and the agent's current junction in maze. There is some randomness in this behaviour, modulated by about 80 parameters in the genome. Fitness is aggregate of proportion of pellets eaten to max pellets and time survived. Using the GA to tune the parameters, the authors were able to match the performance of their hand-tuned controller. However, the controller architecture was not sufficiently flexible to discover novel or high-level strategies. More recently, Thawonmas tuned a rule-based controller for the full competition version of Ms. Pac-Man [14]. Here the controller relies on a pathfinding algorithm based on a tree search. It is guided by the current threat level to the agent using a subsumption [15] goal selector. Nine rules are interpreted in order, with the first one qualifying setting the goal. Parameters for the

goals were determined using evolutionary search with mutation (but not crossover). The evolved controller outperformed the hand-designed one. Laramee used a similar approach to evolve preferences for goals for an agent in a fantasy game setting [16]. The genome was a list of multipliers for goal desirability. His population quickly converged to an aggressive personality, and he notes this could be adjusted by changing the fitness weights.

Counterstrike is a popular multiplayer First Person Shooter (FPS). Cole et al tuned a hand-designed controller for Counterstrike with GA [17]. The parameters were preferences for each weapon and aggressiveness. Weapon choice in Counterstrike greatly changes the style of play, and aggressiveness affected the bots decisions about how to navigate the levels. Fitness was evaluated using the Counterstrike in-game reward table. Notably, the GA-tuned bots were just as effective as the hand-tuned ones. Overholtzer took a similar approach in evolving preferences in a designed AI routine for an open source FPS [18], [19]. Preferences for decision making were evolved using just mutation operators, leading to behaviour that was different and more effective than that using the designer-chosen parameters.

Unreal Tournament 2004 is a multiplayer FPS used for AI competitions [20]. A programming interface to the software gives bots full knowledge of the game environment. It also handles navigation, so the bots can focus on high-level decision-making. Esparcia et al used this to tune the built-in UT2004 fuzzy finite state machine controller's parameters [21]. Their goal was to explore the effects of different kinds of fitness functions on each controller's personality. They evolved personality traits based on approximately forty parameters affecting shooting accuracy, jump height, agressiveness, and preference for ranged or melee combat. They were able to produce differentiated personalities, although many of the parameters had a direct effect on the bots' performance (e.g. aiming accuracy). Using the same approach the team investigated selection for team play [22]. Despite the team focus, the fitness function rewarded individual performance for enemies killed and items retrieved. The team received the sum of individual fitness scores. Although the authors hoped to create team member specialization, the winning teams had identical members. This is possibly because their controller was not flexible enough generate diverse behaviour effectively given its free parameters. Cuadrado et al took part in a UT2004 competition with their own custom-built FSM [23], using GA to evolve just the dodge functionality of their bot. The module takes in trajectory, damage radius of incoming missile, and outputs a jump decision. The jump decision is discretized, based on incoming missile direction. The controller was trained with two bots: one shot, and one dodged and was rewarded for how long it survived. The evolved dodge bot lived significantly longer than their hand-tuned controller. However their overall controller did not win that year's competition, in part because their controller didn't account for environmental hazards or multiple threats.

From these examples we see that tuning simplifies the final steps of optimizing a controller for performance or personality. Its success however depends on the choice of parameters to tune, and whether the overall system provides enough complexity to address the designer's goal behaviour.

2.1.2 Direct Architectures

Genetic algorithms can be applied directly to a controller. The controller delineates regions of sensor inputs and enumerates the regions as discrete states. The genome contains an effector response for each enumerated world state. This approach limits the controller's flexibility to the designer's choice of world states, but constrains the complexity of the search problem and is suitable for reactive controllers.

A simple implementation of this idea for a collision avoidance controller is given in Schwab [24]. The game environment is a clone of the classic game Asteroids. The ship is given information about the nearest asteroid moving toward the ship:

- A direction, discretized into eighteen pie slices,
- a distance, with four possible values,
- and a velocity, with ten possible values.

This results in 720 possible input states for the nearest obstacle to avoid. The genome has a response for each state consisting of direction and thrust commands. Fitness is granted as long as the ship is alive and close to an asteroid. Example code was provided for this controller, and notably it was not found to perform as well as advertised. However it has served as a test bed for ongoing work investigating how to improve its performance as well as for exploring other methods.

Q-learning is a standard approach to reinforcement learning in AI [25]. McPartland et al used a Q-learning variant called Sarsa to develop bots for a simple FPS [26]. Notably this is the only example of learning not based on genetic algorithms. However the method is similar in principle to the asteroids controller described above. The authors created a custom, simple FPS environment to experiment with. Again, the input space was discretized. The bot is provided with information about enemies, walls, and items, in terms of near and far distances, in three wedges in front of the bot. This is an abstraction of a human player's view. The controller was assigned a reward based on the outcomes of each decision. Over the course of a session, the decision rules are adjusted based on what worked in the past. If the previous decision led to an unfavourable outcome, a novel decision is created. A challenge with this approach is the explosion of states; the monolithic controller created here had 137781 action-response pairs to learn. This was mitigated by creating separate controllers for individual tasks. Three tasks were defined: navigation, combat, and goal selection. Each were trained with different objectives, and had different inputs. This reduced the input space to each to about 40000 states each. The monolithic and hierarchical controllers were both evaluated against a human-designed bot. The hierarchical controllers were more effective than the monolithic one. The hand-designed state machine controller was more effective in combat, but less flexible; it had been developed only to pay attention to one opponent at a time, whereas the combat controllers could account for up to three opponents.

Priesterjahn used a similar but more sophisticated direct encoding method to evolve a Quake III bot [11]. Here the environment was simplified from the full game: only one weapon was

available, and the levels were limited to a flat plane so that all combatants were at the same elevation. The agent senses the world using an occupancy grid, centered in the agent's reference frame. The contents of the grid define a world state, and each grid configuration in the agent's database had an associated action. An important difference between this scheme and those used in the other examples in this section is that the controller does not have a comprehensive catalogue of grid states. Rather, the agent chooses the grid state available in its database that is closest to the measured state. The responses are the normal FPS controls: movement in four directions, turning, and, choosing whether to attack. The genetic algorithm evolved both the grid states and associated responses. Fitness is rewarded for inflicting damage to enemies and removed for losing one's own health. Notably the designers adjusted the weights of these two factors as the bots evolved to guide the personality of the agents. Too much emphasis on avoiding damage caused cowardly behaviour, and too much on aggression resulted in suicidal agents. Given the simplified environment, the agents were not tested against other kinds of controllers, but the resulting behaviour was convincing to the authors.

2.1.3 Fixed Topology Neural Networks

Neural networks [27] are a popular architecture for learning AI. They can be used to drive the entire controller, or for specialized tasks. There are many ways to structure neural networks, but in the examples described below there are three popular types: the single-layer perceptron (SLP), multi-layer perceptron (MLP), and recurrent neural networks (RNN). SLPs have inputs connected directly to outputs, and can approximate linear functions. MLPs typically have one additional hidden layer, and can approximate arbitrary functions with precision dependent on their size. Recurrent neural networks have connections running in both directions. They are more complex and computationally expensive to run, but can produce more complex behaviour than a similarly-sized MLP, and can also maintain memory of previous states. For any architecture, the inputs to the network are sensor data and the outputs are decisions, e.g. effector decision or goal selection. Training the network is a matter of teaching the relationship between sensor information and appropriate decisions. The complexity of the relationship must be supported by the size and structure of the network. However bigger networks are more difficult to train. For the AI controllers described here, genetic algorithms are used to train the networks instead of the more typical backpropagation. Backpropagation is unsuitable because it is difficult to obtain offline training data. Instead of training the network to behave according to some training set, we want it to discover useful ways to control an agent with a minimum of guidance. We present a few examples here to get a sense of how neural networks have been used for AI in SEs.

Westra designed a controller for Quake III (a FPS) and used neural networks to make decisions about weapon and item selection [28]. The bot had to decide whether to pick up a weaker item nearby, or a stronger item farther away. The authors compared a neural network with the game's original AI decision module, which used fuzzy logic. The neural network was an MLP, with one input for each item, plus indicators for whether dropped, travel time to items, and avoid time (reflecting time to respawn for a taken item). This

gave a total of fifteen inputs. Thirty hidden units were employed. The bot was rewarded for the number of opponent kills in a match. A small population of bots were unleashed upon one another, playing to 500 kills. The evolved weapon selection turned out to be equal to the default AI on weapon selection, and it had better performance on item selection. So this approach was able to match the hand-designed performance, and also added a new capability.

Neural networks were used to generate a complete controller for Xpilot, a 2D multiplayer space combat game [29]. The controller used a single layer network, with 22 inputs and 3 outputs. The inputs were internal variables like heading, velocity, reload time, and sensors for obstacles and enemy ship. The outputs were the ship's controls. Using another bot with fixed AI as an opponent for training, the authors experimented with different fitness functions. Rewarding for survival time produced bots that evaded the enemy without attacking. Adding a large reward for killing the enemy led to suicidal bots that shot their opponent without regard for their own survival. A smaller reward for killing the enemy led to a more balanced behaviour. Different populations evolved various behaviours: circling and shooting, swooping around enemy and shooting from behind, or just circling, thrusting occasionally, and shooting constantly. The controller architecture clearly prevents the adoption of any kind of long-term strategy, but this reactive approach was appropriate for the environment. However it is not clear whether any of the evolved controllers were competitive with the bot.

Togelius et al evolved a neural network controller for a platform game [30]. The platform game is a clone of Super Mario Brothers with randomly generated levels, used recently for a series of AI competitions. The controller compared MLP and RNN architectures. The inputs to the networks are discrete values representing the contents of a grid surrounding the agent. Fitness is rewarded for the distance the agent travels before dying. As the bot progresses, the levels increase in difficulty. The authors found performance was good for smaller input grids, but the networks were unable to cope with larger input grids. This was mitigated with a growing neural network architecture called HyperGP (we discuss such networks in detail in Section 2.2.2). Notably, to date the best performers in the competition did not use learning [31].

Thompson et al developed controllers for a tank game using co-evolutionary search and a subsumption architecture [32]. MLPs were created for different behaviours: attacking, avoiding walls, dodging incoming attacks, and moving to waypoints. Each had two hidden layers, with 3 neurons at each layer, and was trained with a unique fitness function. For example, for attacking, fitness was a weighted aggregate of the agent's final health and speed to win. Alternatively, if the agent lost, it was rewarded for a longer game and how much damage it inflicted on its enemy. The base layers were trained first, then frozen as the training moved on to the next layer. The authors found that the incremental approach was able to produce a competent controller, whereas training a single MLP for all tasks simultaneously did not produce successful results.

2.2 Evolving Controller Architectures

As we have seen, designing a controller by hand can yield successful searches, but this limits its flexibility. Also the designer must determine through trial and error how to structure the controller, e.g. choose the number of layers and neurons in a MLP. A different approach grows the controller as part of the evolutionary search. As the entire system evolves, the controller increases in complexity. This gradual increase in complexity helps to manage the search by taking only small steps. Here we look at two different ways to do this. One is genetic programming, a method of evolving computer programs. The second is a collection of techniques for evolving the nodes, structure, and connection weights of neural networks simultaneously. One commonality to both these approaches is that researchers have needed to constrain the search in order to get useful results. Usually this is in the form of subdividing overall tasks for the controller, and evolving them separately. This modularization turns out to be a key enabling approach for complexifying controllers.

2.2.1 Genetic Programming

A Genetic Program (GP) is a program evolved with a genetic algorithm [33]. It is usually represented as a tree, with functions as nodes, and variables and constants as terminals. The advantage of GPs is that their complexity is limited only by the primitives used to construct them. Their disadvantage is that the size of the search space defined by a collection of nodes and terminals is much larger than that of fixed topology controllers. The practice of evolving effective programs using genetic programming is currently not completely straightforward. Nevertheless there are many examples achieving human-competitive results [33]. Here we compare some attempts of using GP for game controllers.

One of the first applications of genetic programming to a controller was for a game of tag [34]. The game simulated two robots, with identical speed and sensors, making 25 discrete moves. The pursuer chases the evader; if caught, they switch roles. Controllers for both behaviours were evolved separately when creating offspring. The fitness for each run was the time step count where the individual was not "it", averaged over four games with different opponents. So fitness in each generation is relative to that particular population. Reynolds found that weak performers were quickly dominated and were removed from the population as evolution progressed. Populations converged on good-enough-strategies, leading to a loss in diversity. This version of Tag is actually a solved problem – there is a provably optimal behaviour for both roles. The evolutionary methods used here were not able to achieve this optimum, but they produced behaviour that was very close to it. Experiments where the controllers for pursuit and evade were not segregated were found to take longer to converge to effective solutions.

Another simple example is the generation of a controller for the game of Asteroids [35]. In this variant of Asteroids, the player has a limited but regenerating energy store that can be used to erect an invulnerability shield and fire its weapon. The GP is built using sensor functions, functions that control the ship, and standard relational operators (AND, XOR, NOT). The sensor functions obtain information from the game engine: the ship's field of view ahead and its occupancy, a proximity alarm, and the ship's own state including its

current movement. Control functions cause turning, thrusting, shields, and firing. As a control, the author wrote sub-programs and a high-level controller that could successfully play the game without being damaged indefinitely. GP was unable to match this performance using a monolithic controller. However, GP using the author's hand-coded sub-programs was able to produce a good controller quickly. But he had limited success in encouraging this specialization with separately evolved subprograms. Notably he was unable to find an approach to generate a controller equal to his hand-designed one.

Hierarchical Task Networks (HTNs) are natural targets for GP, since they are already organized as networks. Keaveney and O'Riordan used GP to generate HTNs in a strategy game [36]. The game is similar to the boardgame Risk. Opposing armies attack each other to claim territory. The bigger army always wins, and armies from multiple neighbouring territories can coordinate their attacks to get flanking bonuses. Each player can only see the state of territories that are controlled, occupied, or adjacent to his owned territories. New units are produced proportionally to number of territories controlled. The job of the AI is to generate an order for each tile it controls. A hand-designed controller evaluates these orders through an iterative process that generates plans and allocates units to them. These are re-evaluated and units released if the plan cannot succeed so they can be committed to other plans. Three functions are used to evaluate decisions along the way: minimum needed, order priority, and edge priority. These functions that are evolved with GP, each with its own subset of nodes and terminals. This is an example of a specialized use of learning – the overall controller structure was predefined, and GP was used to make decisions based on the state of the game. Fitness is rewarded based on time in game: wins rewards a quick victory, a draw rewards territories captured, and a loss rewards a longer game. The evolved planners were evaluated using quantitative measures of coordination of their attack and spread behaviour. The authors found the controllers first improved their attack performance, and later learned to spread more effectively. This is another example of co-evolution, and it experienced a problem typical with that approach: cycling. This means strategies evolve that one-up one another, but previous lessons are lost. Here, for example, controllers from the 100^{th} generation dominated some from the 500^{th}. Avoiding cycling by keeping a selection of best controllers from each generation in the competition is very computationally expensive.

Behaviour trees are also natural targets for GP. Lim et al evolved behaviour trees for a strategy game (DEFCON) [37]. They generated some basic trees for specific behaviours, then evolved variations on those trees playing against the hand-coded controller that ships with the game. The evolved controller was able to win a little more than half of them, thus matching the hand-designed AI's prowess.

Ebner and Tiede used GP to evolve a race car controller in an open source racing game called TORCS [38]. Using GP, they evolved two trees: one to decide where to steer, and the other to decide whether to accelerate or brake. The node functions provide information about the car's immediate position in the race and track and its internal state. Each agent was evaluated on five tracks (the authors found evolving on just one track produced controllers that did not generalize to others). Fitness was the average of performance on each track in terms of distance travelled. Experiments were performed varying two things.

First, results were compared where the initial population was random verses results when the initial population was populated with manually constructed seeds. The head-start afforded by the hand-designed tree was found to produce a much better results faster than a random start. The next experiment compared the use of a basic instruction set with an augmented set. The augmented set included additional functions enabling the construction of a classical proportional integral derivative (PID) controller. Here the augmented set made no difference – the GP was unable to discover the PID controller. The authors also noted that the controller may have needed more information about the tracks to achieve better performance.

A variant of GP, more similar to natural evolution than the GP examples above, evolves the genotype of the controller separately from the phenotype. The controller program is built using pre-defined rules following instructions encoded in the genome, similar to how proteins are built from instructions in DNA. This approach makes it easier to exploit things like repetition of structure. This was used to evolve controllers for a simplified version of Ms. Pac-Man [39]. The GA here was a grammar string (genotype) that dictates the creation of a tree program that controls behaviour. Its primitives are relatively high level, such as location of nearest resource or threat. Two different methods of growing the trees from the grammars were compared. The game score was used for fitness. The evolved controllers performed better than the team's hand-coded ones and were competitive with the then state of the art for the Ms. Pac-Man competition.

2.2.2 Neural Networks with Variable Topology

When using a neural network for any task, deciding on its topology is always difficult. Too small a network will fail to manage the necessary complexity of a problem. A general rule of thumb is to create a hidden layer with twice the number of nodes as inputs [40]. But adding more hidden layers makes the networks difficult to train [41]. To manage this complexity, some researchers have invented methods of complexification: allowing evolution to adjust weights in a neural network as well as add nodes. We discuss two popular approaches here, illustrated with applications.

Symbiotic, Adaptive Neuro-Evolution (SANE) is an algorithm for complexifying neural networks [42]. SANE evolves a large population of neurons, selecting some for each trial to participate in the hidden layer of a MLP. A modification called Enforced Sub-Populations (ESP) groups neurons into sub populations that aren't allowed to mix during evolution [43]. This makes connections more consistent, allowing SANE to grow recurrent neural networks (a neuron's relative position in a RNN is more important than for MLPs).

Gomez and Miikkulainen used the SANE technique to evolving a neural network controller for pursuing prey [43]. Here the agent was a predator with a limited sensor range. Its prey was driven by a hand-designed, fixed controller with some random behaviour elements. The increasing complexity of SANE was managed by incremental learning. The authors designed a sequence of goals increasing in complexity. Once the predator had evolved to succeed at each task, it was trained on the next more difficult one. For example: for the first training session, the prey was allowed to take a few moves and then stopped inside the

sensor range. The prey was gradually allowed to move further from the predator, until it began the trial outside the sensor range. Eventually it was allowed to keep moving after its head start. This incremental evolution method produced a SANE controller that could succeed at each task, including the final one. The authors were unable to produce a SANE controller without the incremental steps.

NeuroEvolution of Augmenting Topologies (NEAT) is another method of growing neural networks [44]. NEAT networks are born as small linear perceptrons. The networks are encoded from a genomic representation, and evolved using genetic algorithms. As they evolve, possible mutations include the addition of new nodes and connections. A gene history tracking system makes crossover possible without computationally expensive analysis of the compatibility of network sections. It also protects successful networks with a mechanism called speciation. In speciation, individuals in a population are quantitatively compared in terms of their topology. Similar networks are grouped into species. Individuals compete within species, but not across them. This means that new innovations in the overall population are given time to discover better strategies, without having to immediately outperform veterans. Speciation also helps to preserve diversity through a mechanism called fitness sharing. Individuals of a species must share their fitness with others in the group, thus preventing a very successful species from dominating a population. This brief description explains the main features of NEAT, but it is a complex algorithm with more factors than there is space to explain here.

One of the first applications of NEAT in behavioural AI was a controller in a competitive robot battle [45]. Two robots start at opposite corners of a playing field. The robots can turn and move forward; each movement costs energy. Each robot starts with the same amount of energy, which can be increased by obtaining food items on the field. The contest is decided when the robots collide: the one with the highest energy wins. Each generation, individuals in the population duel multiple times. A point is awarded for each win, and at the end of the generation, the robots are ranked. Fitness is thus relative in each generation. This task has enough complexity to demand non-trivial strategies. For example, a robot may decide to try to obtain more food; but doing so costs energy. Alternatively it may choose to stay still and wait for the opponent to burn energy on its approach. The authors found that the complexity of the best performing robots' strategies improved over time. They also were superior to any controllers evolved with fixed topologies (i.e. connection weights only). This included a run using the best evolved controller's topology. The fixed-topology robot did well, but was not as effective well as the NEAT controller on which it was based. This suggests that there is an advantage in slowly building complexity as opposed to optimizing a full controller.

In the inaugural Super Mario competition [30], discussed earlier in Section 2.1.3, fixed topology controllers were compared neural networks developed using a method called HyperGP [46]. This algorithm is based on NEAT, but replaces its elaboration mechanism with Genetic Programming. HyperGP has been found to converge more rapidly than NEAT. The fixed-topology networks performed slightly better than the HyperGP ones for smaller input spaces. But as the authors increased the size of the input sensor grid, the fixed networks were unable to discover controllers with any effectiveness. The HyperGP networks, however,

was able to find solutions for the larger input grids. This again suggests that incrementally growing networks is useful as a way to cope with more complex inputs, although it may have no advantage in small problems where the designer can specify a complete solution.

This finding was echoed in a comparison of different evolutionary methods for designing neural networks [47]. The authors found that architectures that complexify tend to do better than fixed topologies, unless the best topology is already known. When the best topology is unknown, NEAT generally outperforms SANE/ESP. Overall these methods work well, especially for difficult problems like robot control where standard methods such as backpropagation are difficult to apply.

3 Fitness Functions in Evolutionary Search

Up to this point we have focused on the architecture of the controller. Now we will focus on how to guide the search, from the perspective of its objective, i.e. the fitness function.

The fitness function shapes the path and outcome of the search. It is a signal that provides information to the search process. A poor fitness function may be too complex to provide feedback early in the search, or too simple to stimulate advanced behaviour later on. It can, intentionally or not, contain implicit information about the nature of the solution [48]. This information may limit or enable elements of the final performance. Whether this is desirable or not depends on the goal of the search. To achieve our goal of simplifying design, we may indeed wish to specify some elements of how an agent will behave. However the point of this approach is to relieve the designer from needing to specify details of the implementation of a behaviour. And since our second goal is to produce highly complex controllers, we assume that they will be impossible to fully design by hand. So we assume that the goal of the search algorithm is to find good solutions while minimizing information provided by the designer. This makes the choice and use of the fitness function critical.

First we review two types of unusual fitness measures: imitating human behaviour, and novelty. Second we review methods of guiding the search process, first summarizing organizational methods, then reviewing results using multiobjective search.

3.1 Unusual Fitness Measures

3.1.1 Imitating Humans

Sometimes the goal is not to produce the best possible performance, but instead to produce performance similar to a human's. Alternatively it may be difficult to evolve behaviours as effective as a human's. In this case recordings of human performance can provide a template for the controller to generalize from, or bootstrap to a minimum level of performance.

A simple approach is to record player actions in a situation and associate their response with the situation [49]. A more complex approach is to use recordings of player data in an FPS to train the controller how to behave in different contexts, e.g. for moving, fighting, and aiming [50]. These controllers may not generalize to situations different from those recorded.

For game AI, the best opponent may not be the most competent one; instead more human-like behaviour may be more interesting. Soni et al set out to create human-like bot in Unreal Tournament 2004 that were more enjoyable opponents than the standard bots [51]. Their approach was to modify the default bot controller in UT2004, which is a fuzzy finite state machine (FFSM). The authors modified the FFSM transition logic; they replaced the fuzzy transition rules with neural networks trained using human player recordings. The player's actions in the recordings were classified based on the game state. The neural networks were trained using this information in order to make similar choices as the humans. The authors compared implementations that made both deterministic and randomly modified

state transitions. They were evaluated by humans, who played against the bots for 15 minutes and filled out questionnaires. The evaluators found the evolved bots to be more challenging and interesting than the stock ones.

Thurau et al used player recordings to learn both tactical and strategic behaviours in Quake III [52], [53], [54]. The basic structure of the controller is a self-organizing map (SOM) associating a connected graph to locations in a level. At each node, a neural network is trained on situation/action pairs from player recordings. Initially the network's outputs directed the agent's movement and aim (pitch , movement, and firing). This was later extended to strategic behaviours, e.g. deciding where to go in addition to reactive decisions. This was accomplished by training each node on the SOM on where players tended to go next. Further research built on the same concept, using Bayesian models to predict the future behaviour of players conditioned on the game state and their location [55]. The model is conditioned on the player's prior state, goals, and utility of possible actions, and is trained by recordings of human players. The Bayesian network produces action primitives: new direction, jumping, and firing commands. The idea was to infer a human's objective, but then seek it out in a generalized way. Their network was able to qualitatively mirror the general behaviour of human players.

3.1.2 Novelty

Using novelty of behaviour to guide search is a recent idea for overcoming the bootstrap problem in evolutionary learning. If there is no simple path in the controller space from a random starting point to a useful solution, the search algorithm guided by goal fitness will fail. We may have more success if we can instead reward the discovery of incremental steps towards a solution whether or not they produce better fitness [56]. This can be achieved by rewarding controllers that behave unlike any previous ones. Importantly, the measure is of behavioural novelty and not controller structural novelty. For example, different neural networks can actually behave identically. So we cannot simply reward novel structure – the output of the controller must also be different according to some quantifiable measure. Given such a measure, novelty is calculated measuring sparseness in behaviour space, i.e. by the average distance from one individual's behaviour to its k-nearest neighbours' behaviour. The genetic algorithm search is otherwise the same as for other fitness measures.

Lehman and Stanley introduced the concept of novelty based search using a NEAT controller for a robot search task [57]. The robot could sense the direction to the goal, as well as nearby obstacles in the maze, i.e. walls. The NEAT controller used this information to produce movement commands to its two drive wheels. The authors compared controllers rewarded with traditional fitness, the final distance to goal, and novelty. Novelty was defined as the final position of the robot after a fixed period of simulation time. A position sufficiently different from previous runs received a high reward. Experiments were performed with two mazes; both had walls arranged so that in order to reach the target, the robot had to take long detours around intervening walls. The novelty-based algorithm was able to find a controller that reached the goal in less time than the traditionally trained one, and with a simpler controller. For the more difficult of the two mazes, the fitness-trained controller was completely unable to reach the goal.

The method has been tested on other tasks, such as foraging and robot bipedal walking [56]. A challenge in using novelty is defining the measure of behaviour. For foraging, instead of the standard score awarded for total food gathered, a time history of food gathering rates was used, thus rewarding different trajectories in food-gathering-space. For the bipedal robot learning to walk, the standard fitness is distance travelled. The authors chose to measure a time history of the distance travelled sampled every second over the run of the experiment. Both these methods were able to find better controllers, faster, than their fitness-based counterparts.

While novelty has been shown to overcome some of the problems with bootstrapping, it has been criticized for ignoring useful information from traditional fitness functions [58]. However, as we shall see in Section 3.2.3, combining these methods in multiobjective search provides the best of both approaches.

3.2 Guiding Evolution

As we have seen, oftentimes the main problem in evolving controllers is finding good solutions in the enormous search space of all possible programs. The first approach is to constrain the controller. In Section 2 we reviewed increasingly flexible controller architectures. The simplest ones have a small number of free parameters easily tuned by genetic algorithm search. Fixed topology neural networks and genotype controllers are more open but still constrained by their chosen size, and were also straightforward to tune. A more open controller architecture, such as genetic programs or evolved neural networks, is closer to delivering our stated goal of automatic design. But this flexibility comes at a cost: the search algorithm is unable to find solutions starting from a random start. Here we review methods of managing the search: first managing complex controllers with modularity, then guiding search with diversity, and finally with multi-objective evolution to combine methods.

3.2.1 Organizing Principles

Variable topology controllers usually require some form of constraint to provide guidance to the search. Here we review those we have already seen: incremental evolution, subsumption, and hierarchical organization.

Incremental evolution starts with a simple task. When the controller achieves the initial task, a slightly more difficult one is introduced. This is an intuitive approach, used often in training both humans and animals, and we have already seen an example of its successful use above [43]. One drawback of this approach is that the training plan and tasks must be curated by the designer. A second drawback is that the controller being trained must have some mechanism for producing the additional complexity required without losing its basic capabilities. This is more of a concern in something like a fixed neural network. As we have seen, incremental evolution is actually very complementary to complexifying architectures.

Brooks's subsumption architecture is popular modularizing architecture for robotics [15]. We have already discussed an example of evolving layers sequentially using this approach

[32]. In subsumption, behaviours are divided into modules and layered according to priority. Goal conditions determine the highest priority layer that is active. Subsumption has been used in commercial robots, for example in the Roomba vacuum cleaner; it is a proven method.

Hierarchical organization of controllers is a third approach. Here, the designer designates subcategories of behaviour, and trains separate controllers for each subcategory. An additional component chooses which sub-behaviour to activate at any time. This could also be evolved, as we have discussed above [36]. This approach is more flexible than subsumption, but puts more responsibility in the hands of the designer to decide how to architect the controller's behaviour.

3.2.2 Diversity and Co-Evolution

We have seen two other factors used to guide evolutionary search by putting pressure on its populations to move out of local minima: co-evolution and population diversity.

Competitive methods pit co-evolving populations against each other in hopes of spurring an evolutionary arms race. Competing evolving controllers can be faster than evolving just one side. A problem with this approach is cycling: in rock paper scissors fashion, strategies may be discovered and discarded, then rediscovered. This can prevent evolution from progressing. One solution is to test new individuals against a random assortment of members from previous generations [59]. However, as we discussed above, this can be computationally expensive [36].

Diversity is an approach to dealing with premature convergence in evolutionary search. Premature convergence is an example of search finding a local minimum; a partially successful subpopulation quickly dominates the population and prevents the discovery of better solutions. Some architectures have built-in diversity preserving mechanisms, e.g. fitness sharing in NEAT [44]. With fitness sharing, similar individuals must share their fitness. Fitness sharing prevents individuals from converging to a local minimum; if too many of them behave the same way, their reward is diluted. Siang et al performed a detailed empirical analysis of the effect of diversity in co-evolutionary learning [60] but were unable to find clear answers as to its importance in the context of their study. This question is revisited in the context of multiobjective evolution below.

3.2.3 Multiobjective evolution

The ideal search would be able to find solutions with feedback for just the final behaviour we seek. But as we have seen, this only works for a small search space. Complexifying controllers can offer the needed flexibility, but they create a search space is too big to find good solution without guidance. There are four ways people have tried to solve this problem [61]: incremental evolution, behavioural decomposition, fitness shaping (rewarding sub-objectives explicitly in an aggregate fitness function), and environmental complexification (gradually make the task harder through the environment, e.g. predator chasing prey that gets an increasingly bigger head start [43]). All of these have had limited success, and

furthermore they all rely on the explicit guidance of the designer. Multiobjective search removes some of these limitations. Multiobjective search can make use of the definition of intermediate stages, but removes the requirement to order them or design a sequential training plan. Additionally, the intermediate stages can be overspecified; if they turn out to be unhelpful, the algorithm isn't required to solve them. Multiobjective search makes it easy to use additional methods such as novelty or diversity preservation without changing the controller algorithm. The multiobjective search itself manages the tradeoffs between each subgoal.

Multiobjective evolution requires a new method of comparing individuals. The problem is that individual a might be better than individual b at objective x, but worse than b at objective y. To get around this we define domination. Individual a dominates b if:

1. a is not worse than b on any objective, and
2. a is better than b at at least one objective.

The set of non-dominated individuals is called the Pareto front[2] of the search space. Deb et al introduced an efficient algorithm for sorting individuals according to these rules called NGSA-II [62]. This has become the standard algorithm used in multiobjective evolutionary search.

An additional benefit to multiobjective search is that it allows for the comparison of the value of different objectives, by evolving controllers with different mixes. Van Hoorn used multiobjective search to compare methods for developing driving game controllers that learn from human performance [63]. They combined two approaches to training recurrent neural networks. The first was direct, where the networks were trained on the relationship between the vehicle's sensor inputs and the human's actions. The second was indirect, where a network was evolved with fitness measures that reward similarity with human performance. Using multiobjective evolution enabled the team to compare the effectiveness of different combinations of these factors. The three objective functions were: the distance travelled without leaving the track, similarity to the human's steering choice at the same point on the track, and the similarity to the human's acceleration and braking choice at the same point on the track. Using just a distance metric and ignoring human performance, the evolved bots were able to race on the tracks on which they were trained, though with poor generalization to other tracks. Ignoring distance and using just similarity to human performance, the bots were unable to develop competent behaviour. Using all three objectives, the bots tended towards two ends of the Pareto front – either success at distance or similarity with human behaviour. No evolved controller was capable of satisfying all three objectives.

Agapitos et al also used multi-objective evolution do develop bots in a racing game, exploiting the diversity of results on the Pareto front to obtain diverse bot behaviours [3]. The environment was a two-dimensional remote-controlled car racing game. Each car had noisy sensors measuring its speed, angle and distance to the nearest track wall and opponent. The controller was evolved using genetic programming. It generated discrete state control

[2]Named after economist Vifredo Pareto.

instructions to steer and accelerate the vehicle. Each evolved controller raced against a fixed AI car. The authors created ten diverse and sometimes competing objectives, e.g. absolute distance travelled, progress relative to the opponent, and maximizing or minimizing collisions. The hypothesis was that different combinations of objectives could be chosen to select to generate a range of diverse behaviours. This did occur, but not always predictably. For example, optimizing for absolute progress and minimizing wall collisions led to cars that travelled at very high speed and crashed often. Adding objectives for minimizing the rate of acceleration changes produced more conservative and human-like behaviour. This illustrates the effectiveness of adding objectives to tune behaviour without needing to specify how it is achieved.

Mouret and Doncieux used multiobjective search to compare methods for generating diversity [58]. In particular, they were interested in teasing out factors contributing to Stanley's novelty search approach. Stanley's novelty search combined both the NEAT architecture, which has diversity-preserving mechanisms built-in, and a novelty fitness measure that tracks previous solutions in an archive. Mouret used a simpler complexifying neural network to remove the effects of NEAT from the results. Previous work had shown that some of success of Stanley's novelty method stemmed from his NEAT neural network evolution algorithm, but diversity measures could be applied to other types of complexifying architectures [61]. The main goal here was to compare Stanley's behavioural novelty with behavioural diversity. The difference between the two is that novelty compares the behaviour of individuals with those in an archive of all distinct previous behaviours, and diversity compares the behaviour of each individual only with the others in its generation. In addition they hypothesized that using multiobjective search to combine novelty and/or diversity with traditional fitness could lead to better solutions. To these ends, Stanley's deceptive maze experiment [57] was recreated using six different combinations of factors:

1. the traditional fitness measure, distance to goal;
2. novelty;
3. novelty without diversity – compare only with the archive, not the current population;
4. fitness and novelty;
5. fitness and diversity;
6. fitness, novelty, and diversity.

Overall they found that the single fitness measure failed, and that both novelty and diversity produced good results. Search with novelty using only the archive took significantly longer to converge. The difference between the two was efficiency: population diversity took more generations to arrive at a result than novelty, but novelty is much more computationally expensive since it must maintain the archive of previous behaviours and compare all newcomers with the archive. Adding fitness produced fine-tuned results more efficiently than waiting for novelty to discover them. The authors note that more complex problem spaces may provide more insight into the effects of these different factors.

Mouret and Doncieux used multiobjective evolution as an alternative to incremental learning in a sequential light activation task [61]. The objective was to demonstrate a method where subgoals are specified but a training sequence does not need to be designed. In the experiment, a set of lights must be activated in one of two possible sequences by a robot. The robot is controlled by an evolving neural network, which senses the direction of each light and produces drive commands to each of its two drive wheels. Fitness functions reward the activation of each light in a multiobjective search. A final fitness function rewarded the speed with which the final light was activated. The authors also used behavioural diversity and novelty to improve the results. The authors found that multiobjective search was able to succeed at the task relatively quickly, where using a single fitness (the time to turn on the final light) was unable to produce a working controller.

Van Hoorn et al used multiobjective search with a hierarchical controller to evolve a bot for Unreal Tournament 2004 [9]. The environment was a modified version of Unreal Tournament 2004, in which all floors were flat, no items were available, and only one weapon was used. Three behaviours were evolved: shooting, exploration, and path-following, each using separate multiobjective fitness functions. For example, the shooting controller sensed the enemy location and output move and shoot instructions. It was rewarded for causing damage to its opponents and shooting accurately. Each controller was evolved and evaluated independently before all were combined into the complete controller. For example, the shooting controller found two distinct types of behaviour. The first walked in circles until an enemy was seen, then shot it. The second initially turned without moving until an enemy was detected, at which point the bot moved directly towards the enemy while shooting. The researchers chose the second behaviour for their tournament bot. A fourth RNN was evolved to decide which controller to activate given the world state. The complete hierarchical controller was found to be competitive with a hand-designed bot. The team also attempted to evolve a monolithic controller using multiobjective evolution. This was unable to produce competitive behaviour. However this bot had a much smaller network, and less detailed objective functions. The authors were hopeful that more training time could also produce better results.

4 Conclusions

We have progressed through a succession of controller architectures and evolutionary search methods for devising controllers for synthetic environments. To apply evolutionary search to CGFs, we must consider the application area from the client perspective, and the investigation of these methods from a technical one.

For application, we need to identify practical, near-term goals to demonstrate the potential of this approach. Our primary goal is the automation of development of AI to reduce the need for pucksters in exercises and experiments. We propose to focus initially on delivering the benefits of robust AI to the CF for their exercises and experiments. In support of this, it would be helpful to investigate precisely how a puckster's time is spent. Scenarios are developed ahead of time by planners, and a script is produced for the pucksters to follow. Pucksters must follow the script instructions as well as make some number of decisions as the simulation progresses. What decisions? How much flexibility do they have in choosing behaviours? How much must they interpret the high-level direction in the script? What fraction of their time is spent on low-level, reactive behaviour control in the simulation? Answers to these questions will help us focus on where evolutionary (or even other) methods will be most helpful, or lead us down a different path, if such methods cannot be of any use at the present time.

Once the puckster's task has been identified, we must investigate the shortcomings of the current AI in those tools. This was done from a high level perspective in recent work [8], but we still do not have an understanding of why the client is not using the AI available in CGFs. We should attempt to answer these questions in discussions with our clients and also by observing them in simulated exercises. If we can find the limits of traditional AI, we will have a target for improvement using evolutionary (or other) methods.

We also need a development plan from a technical perspective. Recall our two stated objectives: to create controllers with less effort than traditional methods require, and to create controllers capable of more complexity than is manageable with traditional methods. For the first, a simple goal would be to reproduce the performance of a given controller using evolutionary learning. Starting with very simple problems, e.g. collision avoidance, we can make progress towards controllers and problems of increasing sophistication. The results from these attempts should give an indication of whether acceptable performance will be achievable in the near term. In order to pursue the second goal of increasing sophistication, it will be helpful to investigate issues of complexity in the controllers themselves. The controllers reported in this document are all of limited complexity. For example, none of the evolving neural networks are deeper than four layers, and typically had only dozens of neurons. Recent methods such as deep learning may be able to overcome this limitation [41]. Another factor that may be of use is finding a way to quantify the complexity of controllers. Even an imperfect measurement of this will facilitate comparison of evolved controllers with traditional ones and each other.

The architecture for our controller should be informed by what we have learned here. For both objectives, an evolving structure with search guided by multiobjective evolution is the

most promising choice. However some experimentation will be needed to find the right mix of flexibility and structure. We have seen how more structure trades off efficiency at the expense of flexibility. What is the right balance between structured and purely evolved controllers? Are there general principles for constraining the design that can be effective? For example, some of the complexity of these problems comes from the interpretation of the world. Neural networks are adept at generalizing and compressing this information. This approach was used in the Mario competition discussed previously to reduce the input space of a controller [64], something the contest designers noted as being a problem with their approach. Yet instantly evaluating the complete scene is something a human player can and must do to play the game effectively. An effective structure may be one that is adept at abstracting the input space, then generalizing a response, but constrained in a way that encourages the abstraction of sensor inputs.

A combination of some of the methods presented above may produce a novel approach for the automatic generation of behaviour for entities in synthetic environments. Success in this area will reduce costs for training, concept development, and experimentation, and enable new capabilities for autonomous systems both in and potentially outside of synthetic environments. Even if the AI proposed here does not decrease in development costs, it will still be a success if it provides significantly more capable AI than that which is currently available.

References

[1] Abbass, H., Bender, A., Gaidow, S., and Whitbread, P. (2011), Computational Red Teaming: Past, Present and Future, *Computational Intelligence Magazine, IEEE*, 6(1), 30–42.

[2] Taylor, A., Abdellaoui, N., and Parkinson, G. (2009), Artificial Intelligence in Computer Generated Forces: Comparative Analysis, Huntsville, Alabama: The Society for Modeling and Simulation International.

[3] Agapitos, A., Togelius, J., Lucas, S. M., Schmidhuber, J., and Konstantinidis, A. (2008), Generating diverse opponents with multiobjective evolution, In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pp. 135–142, IEEE.

[4] Laird, J. E., Jones, R. M., Jones, O. M., and Nielsen, P. E. (1994), Coordinated Behavior of Computer Generated Forces in TacAir-Soar, *In Proceedings of the fourth conference on computer generated forces and behavioral representation*, pp. 325—332.

[5] Councill, I. G., Haynes, S. R., and Ritter, F. E. (2003), Explaining Soar: Analysis of Existing Tools and User Information Requirements.

[6] Randall, J. (2007), Using Software Agents to Control the Behaviour of Simulated Entities, *DRDC Atlantic TM 2007-342*.

[7] Levesque, J., Cazzolato, F., and Martonosi, J. (2009), CAMX: Civilian Activity Modelling for exercises and experimentation, (Technical Memorandum DRDC CORA TM 2009-065) Defence R&D Canada – CORA.

[8] Parkinson, G. (2009), AI in CGFs comparative analysis: Summary report, (Contract Report DRDC OTTAWA CR 2009-201) Defence R&D Canada – Ottawa.

[9] van Hoorn, N., Togelius, J., and Schmidhuber, J. (2009), Hierarchical controller learning in a First-Person Shooter, In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009*, pp. 294–301, IEEE.

[10] Mitchell, M. (1998), An introduction to genetic algorithms, The MIT press.

[11] Priesterjahn, S., Kramer, O., Weimer, A., and Goebels, A. (2006), Evolution of Human-Competitive Agents in Modern Computer Games, In *IEEE Congress on Evolutionary Computation, 2006. CEC 2006*, pp. 777–784, IEEE.

[12] Lucas, S. M. (2007), Ms Pac-Man competition, *ACM SIGEVOlution*, 2(4), 37–38.

[13] Gallagher, M. and Ryan, A. (2003), Learning to play Pac-Man: an evolutionary, rule-based approach, In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03*, Vol. 4, pp. 2462– 2469 Vol.4, IEEE.

[14] Thawonmas, R. and Ashida, T. (2010), Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3, In *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 235–240, IEEE.

[15] Brooks, R. (1986), A robust layered control system for a mobile robot, *Robotics and Automation, IEEE Journal of*, 2(1), 14–23.

[16] Laramee, F. D. (2002), Genetic algorithms: Evolving the perfect troll, In *AI game programming wisdom*, pp. 629—639, Hingham, MA: Charles River Media.

[17] Cole, N., Louis, S. J., and Miles, C. (2004), Using a genetic algorithm to tune first-person shooter bots, In *Congress on Evolutionary Computation, 2004. CEC2004*, Vol. 1, pp. 139– 145 Vol.1, IEEE.

[18] Overholtzer, C. A. and Levy, S. D. (2005), Evolving AI opponents in a first-person-shooter video game, In *Proceedings of the 20th national conference on Artificial intelligence - Volume 4*, pp. 1620–1621, Pittsburgh, Pennsylvania: AAAI Press.

[19] Overholtzer, C. A. and Levy, S. D. (2005), Adding Smart Opponents to a First-Person Shooter Video Game through Evolutionary Design, *Proceedings of AIIDE 05: Artificial Intelligence and Digital Entertainment*.

[20] Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S., and Sollitto, C. (2001), Gamebots: A 3d virtual world test-bed for multi-agent research, In *Proceedings of the second international workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pp. 47—52, Montreal, Canada.

[21] Esparcia-Alcázar, A. I., Martínez-García, A., Mora, A., Merelo, J. J., and García-Sánchez, P. (2010), Controlling bots in a First Person Shooter game using genetic algorithms, In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE.

[22] Mora, A. M., Moreno, M. A., Merelo, J. J., Castillo, P. A., Arenas, M. G., and Laredo, J. L. (2010), Evolving the cooperative behaviour in Unreal^{TM} bots, In *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 241–248, IEEE.

[23] Cuadrado, D. and Saez, Y. (2009), Chuck Norris rocks!, In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009*, pp. 69–74, IEEE.

[24] Schwab, B. (2004), AI game engine programming, Charles River Media.

[25] Russell, S. J. and Norvig, P. (2003), Artificial Intelligence: A Modern Approach, 2 ed, Pearson Education.

[26] McPartland, M. and Gallagher, M. (2011), Reinforcement Learning in First Person Shooter Games, *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(1), 43–56.

[27] Bishop, C. M. (1995), Neural networks for pattern recognition, Oxford university press.

[28] Westra, J. and Dignum, F. (2009), Evolutionary neural networks for Non-Player Characters in Quake III, In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009*, pp. 302–309, IEEE.

[29] Parker, G. B., Parker, M., and Johnson, S. D. (2005), Evolving autonomous agent control in the Xpilot environment, In *The 2005 IEEE Congress on Evolutionary Computation, 2005*, Vol. 3, pp. 2416– 2421 Vol. 3, IEEE.

[30] Togelius, J., Karakovskiy, S., Koutnik, J., and Schmidhuber, J. (2009), Super mario evolution, In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009*, pp. 156–161, IEEE.

[31] Togelius, J., Karakovskiy, S., and Baumgarten, R. (2010), The 2009 Mario AI Competition, In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE.

[32] Thompson, T. and Levine, J. (2008), Scaling-up behaviours in EvoTanks: Applying subsumption principles to artificial neural networks, In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pp. 159–166, IEEE.

[33] Poli, R., Langdon, W. B., and McPhee, N. F. (2008), A field guide to genetic programming, Lulu Enterprises UK Ltd.

[34] Reynolds, C. (1994), Competition, coevolution and the game of tag, In *Artificial Life IV, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, p. 59–69.

[35] Anderson, E. F. (2002), Off-Line Evolution of Behaviour for Autonomous Agents in Real-Time Computer Games, In Guervós, J. J. M., Adamidis, P., Beyer, H., Schwefel, H., and Fernández-Villacañas, J., (Eds.), *Parallel Problem Solving from Nature — PPSN VII*, Vol. 2439, pp. 689–699, Berlin, Heidelberg: Springer Berlin Heidelberg.

[36] Keaveney, D. and O'Riordan, C. (2011), Evolving Coordination for Real-Time Strategy Games, *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2), 155–167.

[37] Lim, C., Baumgarten, R., and Colton, S. (2010), Evolving Behaviour Trees for the Commercial Game DEFCON, In *Applications of Evolutionary Computation*, pp. 100–110.

[38] Ebner, M. and Tiede, T. (2009), Evolving driving controllers using Genetic Programming, In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009*, pp. 279–286, IEEE.

[39] Galván-López, E., Fagan, D., Murphy, E., Swafford, J. M., Agapitos, A., O'Neill, M., and Brabazon, A. (2010), Comparing the performance of the evolvable πGrammatical Evolution genotype-phenotype map to Grammatical Evolution in the dynamic Ms. Pac-Man environment, In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE.

[40] Champandard, A. J. (2002), The dark art of neural networks, In *AI Game Programming Wisdom*, pp. 640–651, Hingham, MA: Charles River Media.

[41] Bengio, Y. (2009), Learning Deep Architectures for AI, *Foundations and Trends® in Machine Learning*, 2(1), 1–127.

[42] Moriarty, D. E. and Mikkulainen, R. (1996), Efficient Reinforcement Learning through Symbiotic Evolution, *Machine Learning*, 22(1), 11–32.

[43] Gomez, F. and Miikkulainen, R. (1997), Incremental Evolution of Complex General Behavior, *Adaptive Behavior*, 5(3-4), 317 –342.

[44] Stanley, K. O. and Miikkulainen, R. (2002), Evolving Neural Networks through Augmenting Topologies, *Evolutionary Computation*, 10(2), 99–127.

[45] Stanley, K. O. and Miikkulainen, R. (2004), Competitive coevolution through evolutionary complexification, *J. Artif. Int. Res.*, 21(1), 63–100.

[46] Buk, Z., Koutník, J., and Šnorek, M. (2009), NEAT in HyperNEAT Substituted with Genetic Programming, In Kolehmainen, M., Toivanen, P., and Beliczynski, B., (Eds.), *Adaptive and Natural Computing Algorithms*, Vol. 5495, pp. 243–252, Berlin, Heidelberg: Springer Berlin Heidelberg.

[47] Floreano, D., Dürr, P., and Mattiussi, C. (2008), Neuroevolution: from architectures to learning, *Evolutionary Intelligence*, 1(1), 47–62.

[48] Nelson, A. L., Barlow, G. J., and Doitsidis, L. (2009), Fitness functions in evolutionary robotics: A survey and analysis, *Robotics and Autonomous Systems*, 57(4), 345–370.

[49] Alexander, T. (2002), GoCap: Game observation capture, In *AI Game Programming Wisdom*, p. 579–585, Hingham, MA: Charles River Media.

[50] Zanetti, S. and Rhalibi, A. E. (2004), Machine learning techniques for FPS in Q3, In *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, ACE '04, p. 239–244, Singapore: ACM. ACM ID: 1067374.

[51] Soni, B. and Hingston, P. (2008), Bots trained to play like a human are more fun, In *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*, pp. 363–369, IEEE.

[52] Thurau, C., Bauckhage, C., and Sagerer, G. (2003), Combining Self Organizing Maps and Multilayer Perceptrons to Learn Bot-Behavior for a Commercial Game, In *Proc. GAME-ON*.

[53] Thurau, C. and Bauckhage, C. (2004), Learning human-like movement behavior for computer games, In *From animals to animats 8: proceedings of the seventh [ie eighth] International Conference on Simulation of Adaptive Behavior*, Vol. 8, p. 315, The MIT Press.

[54] Thurau, C., Bauckhage, C., and Sagerer, G. (2004), Synthesizing Movements for Computer Game Characters, In Rasmussen, C. E., Bülthoff, H. H., Schölkopf, B., and Giese, M. A., (Eds.), *Pattern Recognition*, Vol. 3175, pp. 179–186, Berlin, Heidelberg: Springer Berlin Heidelberg.

[55] Gorman, B., Thurau, C., Bauckhage, C., and Humphrys, M. (2006), Bayesian Imitation of Human Behavior in Interactive Computer Games, In *18th International Conference on Pattern Recognition, 2006. ICPR 2006*, Vol. 1, pp. 1244–1247, IEEE.

[56] Lehman, J. and Stanley, K. O. (2010), Efficiently evolving programs through the search for novelty, In *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, p. 837, Portland, Oregon, USA.

[57] Lehman, J. and Stanley, K. O. (2008), Exploiting open-endedness to solve problems through the search for novelty, *Artificial Life*, 11, 329.

[58] Mouret, J. (2011), Novelty-Based Multiobjectivization, In Doncieux, S., Bredèche, N., and Mouret, J., (Eds.), *New Horizons in Evolutionary Robotics*, Vol. 341, pp. 139–154, Berlin, Heidelberg: Springer Berlin Heidelberg.

[59] Nolfi, S. and Floreano, D. (2011), Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution?, *Artificial Life*, 4(4), 311–335.

[60] Chong, S. Y., Tino, P., and Yao, X. (2009), Relationship Between Generalization and Diversity in Coevolutionary Learning, *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3), 214–232.

[61] Mouret, J. B. and Doncieux, S. (2009), Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity, In *IEEE Congress on Evolutionary Computation, 2009. CEC '09*, pp. 1161–1168, IEEE.

[62] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002), A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Transactions on Evolutionary Computation*, 6(2), 182–197.

[63] van Hoorn, N., Togelius, J., Wierstra, D., and Schmidhuber, J. (2009), Robust player imitation using multiobjective evolution, In *IEEE Congress on Evolutionary Computation, 2009. CEC '09*, pp. 652–659, IEEE.

[64] Handa, H. (2011), Dimensionality reduction of scene and enemy information in Mario, In *2011 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1515–1520, IEEE.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

   Defence R&D Canada – Ottawa
   3701 Carling Avenue, Ottawa ON K1A 0Z4, Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

   UNCLASSIFIED
   (NON-CONTROLLED GOODS)
   DMC A
   REVIEW: GCEC June 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

   Survey of evolutionary learning for generating agent controllers in synthetic environments

4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.)

   Taylor, A.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
December 2011	42	64

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

   Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

   Defence R&D Canada – Ottawa
   3701 Carling Avenue, Ottawa ON K1A 0Z4, Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
13oc	

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Ottawa TM 2011-212	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

   (X) Unlimited distribution
   ( ) Defence departments and defence contractors; further distribution only as approved
   ( ) Defence departments and Canadian defence contractors; further distribution only as approved
   ( ) Government departments and agencies; further distribution only as approved
   ( ) Defence departments; further distribution only as approved
   ( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

Using evolutionary methods is a promising approach to overcoming limitations in developing traditional artificial intelligence (AI) controllers in synthetic environments. Traditional AI is difficult and expensive to create and maintain. As a result exercises and training simulations often require the participation of human players who exist solely to provide believable and realistic behaviour for allied, enemy, and neutral forces. Here we review recent results in evolutionary learning for achieving two goals: reducing the difficulty of creating AI, and creating highly capable and robust AI. Results are organized first according to their architectures; these range from hand-designed controllers with evolved parameters to neural networks that grow in complexity. Second we compare methods of guiding the evolutionary search process, necessary because of the large size of the controller search space. These methods, such as modularization and multiobjective evolution, help to reduce the designer's workload. We explain the need to identify practical applications for evolutionary methods by examining shortcomings in current client use of AI, and outline a research plan for developing evolutionary methods for application to these client requirements. Successful application of these methods will generate better AI, reduce cost, and reduce the human workload in executing distributed simulation exercises.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

genetic algorithm; evolutionary learning; artificial intelligence; computer generated forces

Defence R&D Canada

Canada's leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

DEFENCE DÉFENSE

www.drdc-rddc.gc.ca