Constructing Game Agents Through Simulated Evolution

Jacob Schrum, Department of Mathematics and Computer Science,
Southwestern University, Georgetown, TX, USA
Risto Miikkulainen, Department of Computer Science,
University of Texas at Austin, Austin, TX, USA

Synonyms: Evolutionary computation, evolutionary algorithms, evolutionary machine-learning, neuroevolution, evolutionary agent design.

Definition: Construction of game agents through simulated evolution is the use of algorithms that model the biological of process of evolution to develop the behavior and/or morphology of game agents.

1 Introduction

Computer game worlds are often inhabited by numerous artificial agents, which may be helpful, neutral, or hostile toward the player or players. Common approaches for defining the behavior of such agents include rule-based scripts and finite state machines (Buckland, 2005). However, agent behavior can also be generated automatically using evolutionary computation (EC; Eiben and Smith 2003). EC is a machine-learning technique that can be applied to sequential decision-making problems with large and partially observable state spaces, like video games.

EC can create individual agents or teams, and these agents can be opponents or companions of human players. Agents can also be evolved to play games as a human would, in order to test the efficacy of EC techniques. EC can even create game artifacts besides agents, such as weapons. The reason EC is so flexible is that it requires little domain knowledge compared to traditional approaches. It is also capable of discovering surprising and effective behavior that a human expert would not think to program. If applied intelligently, this approach can even adapt to humans in a manner that keeps providing interesting and novel experiences for players. This article focuses mostly on discovering effective opponent behavior (since that is the focus of most research), although examples of other applications are also given when appropriate.

2 Evolutionary Computation

EC models the process of Darwinian evolution by natural selection (Darwin, 1859) for the purpose of generating solutions to difficult embedded problems. Initially, a random collection of candidate solutions, called the population, is generated and evaluated in a task within some environment. Because of randomness in how the population was generated, there will be variation in the performance of different candidate solutions. At this point a new population is generated from the old population using a mixture of selection, recombination, and mutation.

Selection is based on Darwin's concept of natural selection, by which fitter individuals enjoy higher reproductive success. It involves identifying members of the population that perform best, typically through a fitness function that maps candidate solutions to numeric measures of performance. Sometimes a certain number of top performers are selected directly (a technique known as elitism), but selection is generally a random process that merely favors high-performing individuals, while still allowing some poor-performing, but lucky, individuals to be chosen. This random element is one way of maintaining diversity in the evolving population and is generally

important to the long-term success of evolution.

In order for evolution to progress, some of the slots in the new population must be filled by results of recombination or mutation. Recombination creates a new solution to the problem by combining components of solutions that were selected from the old population. Generally, two solutions from the old population, called parents, are selected and recombined to create a new solution, a child or offspring, via simulated crossover, which models the process of genetic crossover that is a major benefit in biological sexual reproduction. In addition, some of these offspring undergo mutation before joining the new population.

Mutation operations are applied with low probability and generally result in small changes to a candidate solution. It is also possible, and common, for mutation to be applied directly to members of the old population to generate new solutions, which can also fill slots in the new population. Mutation without recombination models asexual reproduction.

The new population of candidate solutions is labelled the next generation of the evolutionary process. The new population now also undergoes evaluation and is subject to selection, recombination, and mutation, which leads to yet another generation, and so on. Because recombination and mutation keep creating new individuals, this process is able to search the space of possible solutions in parallel, and because selection favors high-performing individuals, this search will gradually focus on the best solutions in the search space. As such, the evolutionary process is repeated until some stopping criteria is reached, such as the attainment of a desired level of performance, or the end of a preset number of generations.

A major benefit of this process is that it is general: it can be applied to any domain in which there is a measure of fitness/performance that allows certain solutions to be identified as being better than others.

3 Evolution in Games

Games are typically full of numeric scores and metrics that can easily be used as a means of measuring agent performance. Each possible agent is a candidate solution to the problem of how to behave in the game world. Several different representations for such agents are discussed later, but even given such a representation, there are different ways of evaluating an agent's performance.

Although most game agents are ultimately designed to interact with humans, having humans evaluate all candidate solutions is seldom feasible because it is difficult for humans to maintain focus and evaluate solutions consistently. Completely automated approaches are more commonly used, but sometimes humans can also be incorporated into the process.

3.1 Evolution in Stationary Worlds

A simple approach to evolving agent behavior is to have an evolved agent interact only with a static or stationary world. Such worlds may have no other agents in them or may only have agents with fixed control policies. A world is stationary if it and its agents do not adjust or adapt to what occurs during evaluation. In other words, the probability of experiencing certain outcomes in certain situations remains the same.

An example of an agent evolving in a stationary world without other agents is a racecar controller on a track without other cars. This process can produce skilled racing behavior (Cardamone et al., 2009). To add to this agent the ability to interact with other racecars, a scripted component could be added to the controller that takes over when other cars are near, thus combining scripted and evolved components. Another option is to evolve a racecar controller in an environment filled with scripted opponent cars. A variety of different scripted opponents could be used, either in one trial or across the course of several, in order to make the discovered behavior more robust in the face of different opponents.

Scripted controllers could be rudimentary yet still pose an interesting challenge for an evolved controller to overcome. However, scripted opponents may have weaknesses that evolution can discover and exploit. Such behaviors may result in a high score, even though they may be uninteresting or easily defeatable for human players. Fortunately, the evolutionary approach can be generalized and extended into a

process that discovers good behaviors in an absolute sense. This process is coevolution.

3.2 Coevolution

Coevolution occurs when individuals in a population are evaluated with respect to other evolved individuals. Such individuals can come from the same or different populations and can be evaluated in tasks requiring cooperation or competition. A prominent example of competitive coevolution within a single population is Fogel's (2002) evolved checkers player, Blondie24. Blondie24 was evolved by an evolutionary algorithm that pitted evolved players from a single population against each other. The players that did a better job of defeating other members of the same population had higher fitness and were used to create more offspring for the next generation. The best individual after many generations used the name Blondie24 on an online checkers service and was found to be highly competitive against the human players it faced.

Although checkers is a traditional board game, the same coevolutionary process can be used in video games where bots are needed to fill in for human players. First-person shooter (FPS) games, like the Unreal Tournament and Quake franchises, fit this model because during the deathmatch mode of play (a free-for-all competition between agents trying to kill each other for points), all agents in the game have the same in-game representation and available action set, making it straightforward to evolve such agents with a single homogeneous population.

When the representations and available actions of different classes of agents are different from each other, it makes more sense to evolve separate populations for each type of agent and define their fitnesses in relation to each other. For example, fighting games, like the Street Fighter and Tekken franchises, pit two characters against each other in direct one-on-one competition and generally feature a variety of characters. Therefore, the abilities of the two players may be completely different from each other.

For example, assume that the goal of coevolution is to discover skilled controllers for Ryu and Guile in Street Fighter (at least, each controller will become skilled with respect to its particular opponent). In this scenario, there is a population of Ryu controllers and a population of Guile controllers: each evaluation is a match between a member of each population in which performance depends on the amounts of damage dealt and received by each controller (there are various ways to evaluate performance with respect to these two scores). Any improvement in the performance of individual Ryu controllers will come at the expense of Guile controllers, because the two populations are in direct competition. When set up correctly, this process will result in an evolutionary arms race, encouraging each population to find new ways to overcome the other.

However, there are many potential pitfalls to this process. For example, because each member of each population is different, evaluations of the Ryu population will not be consistent if each Ryu controller faces a different member of the Guile population. There is a risk of a mediocre Ryu controller receiving a high performance rating simply because it was paired with a poor Guile controller. This problem can be somewhat mitigated if every member of each population faces off against several members of the other population, and overall performance depends on performance in all evaluations. However, performance will only be completely consistent if the set of opponents for each population is the same, and picking an appropriate set of opponents is challenging.

Unfortunately, if the set of opponents is chosen poorly, the two populations will not improve in an absolute sense. Instead, they may simply get better with respect to each other in ways that a human player will find bizarre or incompetent. Such improvements may go through alternating cycles because they lead to behavior that beats the current prevalent opponent behavior, but has a weakness against another easily discovered opponent behavior. The trick with coevolution is to discover behavior that incorporates all of the strengths while avoiding all of the weaknesses available within the population's range of possible behaviors.

In some domains, performance that is good in an absolute sense will be achieved automatically. In others, it may be necessary to keep evaluating each population against an archive of defeated opponents to

assure that agents never lose the ability to overcome opponents their ancestors could defeat.

Although coevolution can give rise to behavior that is intelligent in an absolute sense, it is hard to implement correctly. However, agent behavior only needs to be interesting with respect to human players, and there are also ways to evolve agent behavior by including humans in the loop.

3.3 Evolving with Humans in the Loop

As mentioned before, the main challenges to evolving against humans are that they have a limited ability to maintain focus for long periods of time, and that they are not consistent in their evaluations.

A computer can usually run many evaluations between computer opponents very quickly, but all evaluations with a human must occur in real time. After many such evaluations, a human is likely to become fatigued, and be unwilling to expend the necessary effort to evaluate agents properly. Naturally, this tendency also makes evaluations inconsistent. However, fatigue is less likely to occur if it is possible to evaluate many agents at once, or if the population is sufficiently small. Fatigue can also be avoided if a prolonged evaluation process is simply the point of the game.

For example, the commercial Creatures (Grand et al., 1997) series of games is centered around raising artificial creatures called Norns. Superficially, the game looks like a virtual pet style game, but among many other AI techniques applied in the game is support for evolution. The creatures the player raises grow, mature, and seek mates. The Creatures games take place in open-ended worlds in which the fun comes from nurturing and interacting with Norns. However, these lengthy interactions influence when and with whom each Norn mates, and therefore influence the direction evolution takes in the creation of new Norns.

The model used in the Creatures games is interesting and unique, but too slow and time intensive to be useful in most other genres. Inconsistency in human evaluations is also not terribly relevant in the Creatures games because variation and novelty in the results of evolution are part of the fun of the game. Additionally, there is no set goal that the evolved Norns are supposed to achieve, but the game is entertaining precisely because of the variety it produces.

Another manner in which a human may be an inconsistent evaluator is due to a human's tendency to learn and adapt: a human player that changes strategy mid-generation will evaluate members of the same generation differently, which would likely give an advantage to agents evaluated before the human adopted a new strategy.

However, human adaptation is also a potential benefit. Inconsistent evaluations may add noise to the evolutionary process, but in the long run a human or set of humans who evaluate artificial agents will settle on strategies that suit their computer opponents. However, if the humans adapt and improve, then the evolved agents should improve as well. In fact, if this improvement happens in real time, then the resulting experience is more exciting and engaging for the human player.

Therefore, the primary challenge to evolving agents with humans in the loop is in generating new and interesting behaviors quickly enough to keep humans engaged. In general, having one human be responsible for evaluating all individuals puts an unreasonable burden on that individual, so methods that keep humans in the loop need to distribute evaluations in novel ways. These evaluations can either be distributed among several different humans or split between humans and the computer.

Sharing evaluations with the computer means that the computer still evaluates the majority of candidate solutions in the usual way, using a computer-controlled opponent as a stand-in for a human player. This process could in fact be carried out for many generations, only occasionally letting a human face the best evolved agents. If performance against the human is comparable to performance against the computer-controlled stand-in, then evolution is on the right track. Otherwise, data on how the human plays can be collected and used to adjust the behavior of the stand-in. These adjustments can be made using supervised learning techniques, or by evolving the stand-in to emulate human play better. However, such a system is complex, and a great deal of effort is

required to make sure all of the separate components successfully interact.

A conceptually simpler way to distribute evaluations is across many human players. Although using different human players makes inconsistencies in evaluation even more likely, there will at least not be any systematic tendency toward generating behaviors that are inappropriate for human consumption: if any human can exploit an agent's failings, then it will eventually be weeded out of the population. Furthermore, distributing evaluations across many humans is made easier by the Internet: specifically, tools such as Amazon's Mechanical Turk and massively multiplayer online (MMO) games.

In fact, although the MMO model has not yet been used to evolve agent behaviors specifically, EC has succeeded in the MMO video game Galactic Arms Race (Hastings et al., 2009). This space-based action shooter game evolves diverse weapons for users to find and equip on their spaceships. The weapon preferences of all users determine the fitness of weapons. The most popular weapons are more likely to create offspring, i.e., new weapons that players are given when they defeat certain enemies. A similar model could apply for enemy agents in many MMO worlds, with enemies that are more successful in combat with human players being considered more fit, and giving rise to increasingly challenging offspring. Such a process has the potential to give rise to new types of games in which all agents evolve and adapt based on a community of human players.

3.4 Evolving Humanlike Behavior

Pitting evolved agents against human opponents will assure that incompetent behaviors are weeded out of the population. However, simply discovering skilled behavior is not always a problem. Because artificial agents are differently embodied than human-controlled avatars, they may have unfair access to skills that are difficult for humans to develop, which in some cases means that they quickly become too skilled to be good opponents for humans. For example, in Unreal Tournament 2004, artificial agents can be programmed using a system called Pogamut (Gemrot et al., 2009). It is easy for these agents to shoot their weapons with pinpoint accuracy: evolution thus creates skilled agents, albeit in a way that human players find frustrating and inhuman.

However, evolution can still be applied in these situations. Agents can be evolved to maximize performance, but under restrictions similar to those experienced by humans. The ability of such an agent to behave in a humanlike manner was demonstrated in the 2007–2012 BotPrize competition. The purpose of the competition was to develop bots for Unreal Tournament 2004 that human players would mistake for humans at least 50 % of the time. The bot `UT^2` achieved this goal with evolved combat behavior (Schrum et al., 2012). The key idea was to optimize the behavior under humanlike restrictions: the more quickly it was moving and the farther its targets were, the less accurate it was. These restrictions forced the bot to evolve humanlike movement patterns in order to have skilled behavior.

This example demonstrates how the abilities available to an evolved agent have a strong influence on the range of behaviors that are likely to be evolved. These abilities are in turn influenced by the type of controller evolved for the agent. A variety of controllers that can be evolved to produce game agents are discussed next.

4 Evolved Representations

When constructing agents for games via evolution, each candidate solution is a means of representing an agent. Often, this representation needs only account for the behavior of the agent, because its form is often fixed by the constraints of the game. However, diverse agent morphology can also be evolved. Regardless, there are a variety of representations that can be used to suit the needs of any particular game.

4.1 Parameter Tuning

The simplest way to incorporate evolution into traditional agent design is via parameter tuning. If there is an existing controller for an agent whose behavior is influenced by some key parameters, then these parameters can be optimized using evolution (typically

via genetic algorithms or evolution strategies).

For example, a hand-coded controller for an agent in an FPS may have numeric parameters indicating which weapon to favor, depending on the agent's distance from its opponents. Similarly, such an agent may have several distinct control modules, like *attack*, *retreat*, and *explore*, and might decide which one to use based on numeric features such as its own health and its distance from enemies and items. Evolved parameters then specify the exact thresholds for each feature, indicating when one module is used instead of another.

The strength of parameter tuning depends on the strength of the underlying controller. For a bad controller, no amount of parameter tuning may be able to help. Similarly, a very good controller may not be very difficult to tune, resulting in quick but small improvements in performance. In order for evolution to reach its full potential, the evolved representation needs to exist within a search space that is rich enough to contain skilled solutions that a human designer either would not consider, or would have difficulty creating.

4.2 Rule-Based Scripts

Rule-based scripts are a common approach to specifying the behaviors of agents in commercial games. Typically, considerable effort and person-hours go into designing scripts for each agent in the game. Simple agents can have simple scripts, but scripts for opponents must be complicated in order for the game to be challenging and interesting.

Scripts generally consist of a list of rules, and each rule consists of a trigger and a corresponding action or sequence of actions. Triggers and actions may also be parameterized. Evolution can easily rearrange blocks of information and search the parameter spaces of each rule and trigger. Of course, the process can be difficult if there is a large number of basic triggers and actions.

One game genre in which opponents have a large range of possible actions is real-time strategy (RTS) games. Because the computer opponent must control a collection of agents in a large space, the number of actions available is massive. Therefore, it makes more sense to reason about behavior at a higher level. Given a set of high-level actions, or tactics, to choose from, a reinforcement learning technique called dynamic scripting can be used to select the best tactic for each situation, leading to improved behavior. In its basic form, this technique is still limited by the preprogrammed tactics available to the agent. However, dynamic scripting can be combined with evolution that generates new tactics. This process has been successfully applied to Wargus, a clone of the very popular Warcraft II RTS game (Ponsen et al., 2006).

Since commercial game designers are already comfortable using scripts, evolving scripts is a straightforward way to combine existing industry knowledge with cutting-edge AI techniques. However, there are also evolvable representations that are potentially more powerful, but less well known in the game industry.

4.3 Genetic Programming

Genetic programming (GP) is a technique for evolving computer programs, or more accurately subroutines, that are often represented as trees. Each internal node is a function call whose branches are input parameters, and leaves are either constants, or functions with no parameters. These functions with no parameters provide sensor values from the agent to the program.

For any given game, the specific functions that can be used in evolved trees need to be specified by the programmer. The types of functions used depend on how the evolved trees are used to control an agent. Evolved trees could be straightforward function approximators made up of purely mathematical functions using agent sensors to provide numbers. However, trees with arbitrarily complex functions can also be evolved. For example, functions can have side effects that directly lead to agent action or that alter a stored memory structure whose contents can influence future function evaluations.

GP can also be used to evolve behavior trees. Such trees hierarchically decompose behavior into a collection of tasks that are prioritized and then executed only if certain triggers are satisfied. In fact, a behav-

ior tree can be thought of as a hierarchical rule-based script. Behavior trees were initially developed for the commercial release of Halo 2 (Isla, 2005) and have since been evolved in Unreal Tournament 2004 using Pogamut (Kadlec, 2008).

GP can also be used as part of a developmental process: the evolved programs are executed to create some other structure that is actually used to control the agent. Such a process more closely emulates the creation of complex organisms from DNA. With GP, an evolved program tree can be used to create the structure and weights of a neural network (Gruau et al., 1996) or simply be queried to fill in the weights of a predefined network architecture (Togelius et al., 2009). Neural networks have their own set of advantages as agent control mechanisms, which are discussed next.

4.4 Neuroevolution

The human brain is a neural network made up of neurons that connect to each other via synapses and communicate via electrical signals. An artificial neural network is an abstraction of this idea that transmits numerical values in place of electrical signals, and neuroevolution is the process by which artificial neural networks are evolved to solve problems.

There are many neural network models (Haykin, 1999), but the most common is a multi-layer perceptron (MLP), consisting of input neurons, output neurons, and hidden neurons in between. Each neuron is connected to every neuron in the next layer, and a continuous activation function, typically a sigmoid, transforms the numerical signals accumulated in each neuron. MLPs are universal function approximators, assuming the correct number of neurons/layers is available, so they are useful in defining agent behavior. MLPs can be trained by supervised learning if labelled training data is available, but this is seldom the case when defining agent behavior in games.

MLPs typically have their architecture (number of neurons in each layer) fixed before learning, and in such a setting there is a known number of synaptic weights in the network. Discovering the weights for such networks is therefore a special case of parameter tuning. Although intelligent behavior can be learned using MLPs, the large number of parameters can make it difficult to learn particularly large MLPs.

An alternative approach is NeuroEvolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen, 2002), which does not produce MLPs. Rather, NEAT networks can have neurons connected to each other in an arbitrary topology. All networks start evolution with a minimal topology with no hidden neurons. The networks in the population gradually complexify across generations as new neurons and links are added via mutations, which allows for convoluted, but effective topologies. In fact, by beginning the search in a small space with few links, it is often possible to find very effective simple networks with fewer links than an MLP with the same number of inputs and outputs.

A variant of NEAT that allows a team of agents to learn in real-time (rtNEAT; Stanley et al., 2005) was actually the driving force behind a machine-learning game called Neuro-Evolving Robotic Operatives (NERO), in which the player takes on the role of a virtual drill sergeant to train robot soldiers that learn via neuroevolution. NEAT has since then been applied to many other video games.

An extension to NEAT called HyperNEAT (Stanley et al., 2009) can exploit the geometry of a state space to make learning certain behaviors easier. HyperNEAT networks are evolved with NEAT, but with extra activation functions possible in the neurons to capture symmetries and repeated patterns in the domain. Most importantly, each evolved network is used to create another network, which becomes the actual controller of an agent. This is another example of a developmental process (cf. section "Genetic Programming"). A benefit of this process is that it becomes feasible to generate very large, but effective, controller networks from small evolved networks. In fact, HyperNEAT has been effectively applied to simulated RoboCup Soccer Keepaway (Verbancsics and Stanley, 2010) and general game playing of Atari games (Hausknecht et al., 2012) using controller networks whose input layers were linked to 2D grids spanning the entire visual display. Such massive networks are difficult to evolve when each connection

weight must be learned individually.

HyperNEAT is known to produce regular networks with repeating patterns. However, these networks are not inherently modular (though techniques to encourage such modularity exist; Huizinga et al. 2014). Modularity is useful because a challenging problem can be broken down into smaller components that are easier to learn. Breaking up a controller into several distinct sub-controllers is a useful way to achieve multimodal behavior, i.e., behavior that consists of distinct modes subjectively different from each other. Such behavior is necessary in many games, because different strategies often require different actions, such as attacking, retreating, searching, hiding, etc.

Such multimodal behavior can be discovered with neuroevolution through architectures that support multiple distinct output modules. Such modules can exist in the initial population or be added by a mutation operator called module mutation (Schrum and Miikkulainen, 2014). This technique was applied to Ms. Pac-Man, and the evolved networks discovered both expected modes of behavior – such as a mode for fleeing threat ghosts and a mode for chasing edible ghosts – and unexpected modes of behavior, such as one for dodging ghosts after luring them near a power pill, so that when the ghosts became edible they would be easier to eat.

So far, only means of evolving complex controllers have been discussed. However, it is possible to go beyond evolving controllers and evolve the bodies of agents as well.

4.5 Morphology

EC can be used to create many types of structures beside function approximators. The Evolved Virtual Creatures (EVCs; Sims, 1994; Lessin et al., 2014) community has developed ways of evolving interesting creature morphologies, often using graph-based encodings. These encodings allow for arbitrary numbers of limbs and joints arranged in novel ways. Sometimes these morphologies mimic those of real-world organisms, but more unusual morphologies can also emerge; the strange quality of such morphologies would lend itself well to a game filled with aliens, robots, or other bizarre creatures.

Given the body, a means of controlling it is required. Specifically, engaging and disengaging the existing joints and/or artificial muscles will cause parts of the body to move, which can lead to complex behavior if done properly. Sometimes simple repetitive control signals, as from a sine wave, can lead to interesting behavior given the right morphology. Naturally, a human designer could also step in and provide the behavior for an evolved morphology.

However, EVCs can also have their control routines embedded into their morphologies. In particular, specific sensors situated on an EVC can be linked to its muscles and joints. Internally, these connections can be wired in a manner similar to a neural network or electrical circuit, meaning that sensor values may be aggregated and/or serve as inputs to functions, whose outputs are passed on until they eventually determine muscle and joint behavior. Such controllers have been evolved to run, jump, swim, grab objects, chase after a light source, and fight or flee different opponents. These skills could serve as the building blocks for more complex and interesting game agents.

5 Conclusion

Evolutionary computation is a powerful machine-learning technique that has been used to discover skilled and interesting agent behavior in many domains. Video game agents can be evolved to play the game as a human would, to serve as opponents for human players, or can be evolved in a context where interacting with the evolutionary process is the point of the game.

Despite the ability of evolution to discover diverse and interesting agent behaviors, the commercial games industry has not yet harnessed the power of evolution (and other advanced AI techniques). This article provides a useful starting point for understanding what can be done with evolution in games and also points out some areas of untapped potential.

References

Buckland, M. (2005). *Programming Game AI by Example*. Jones and Bartlett Learning.

Cardamone, L., Loiacono, D., and Lanzi, P. L. (2009). Evolving Competitive Car Controllers for Racing Games with Neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, (GECCO'09), 1179–1186. New York: ACM.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life*. London: Murray.

Eiben, A. E., and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Berlin: Springer.

Fogel, D. B. (2002). *Blondie24: Playing at the Edge of AI*. San Francisco: Morgan Kaufmann.

Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., and C., B. (2009). Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. *Agents for Games and Simulations, LNCS 5920*, 1–15.

Grand, S., Cliff, D., and Malhotra, A. (1997). Creatures: Artificial Life Autonomous Software Agents for Home Entertainment. In *Proceedings of the 1st International Conference on Autonomous Agents*, AGENTS'97, 22–29. New York: ACM.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks. In *Proceedings of the 1st Annual Conference on Genetic Programming*, GP'96, 81–89. Cambridge: MIT Press.

Hastings, E. J., Guha, R. K., and Stanley, K. O. (2009). Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263.

Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. (2012). HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO'12, 217–224. New York: ACM.

Haykin, S. (1999). *Neural Networks, A Comprehensive Foundation*. Upper Saddle River: Prentice Hall.

Huizinga, J., Mouret, J.-B., and Clune, J. (2014). Evolving Neural Networks That Are Both Modular and Regular: HyperNeat Plus the Connection Cost Technique. In *Proceedings of the 16th Annual Conference on Genetic and Evolutionary Computation*, GECCO'14, 697–704. New York: ACM.

Isla, D. (2005). Managing Complexity in the Halo 2 AI System. In *Proceedings of the Game Developers Conference*, GDC'05. San Francisco, CA.

Kadlec, R. (2008). *Evolution of Intelligent Agent Behaviour in Computer Games*. Master's thesis, Charles University in Prague, Czech Republic.

Lessin, D., Fussell, D., and Miikkulainen, R. (2014). Adapting Morphology to Multiple Tasks in Evolved Virtual Creatures. In *Proceedings of The 14th International Conference on the Synthesis and Simulation of Living Systems*, ALIFE '14. Cambridge: MIT Press.

Ponsen, M., Muñoz-avila, H., Spronck, P., and Aha, D. W. (2006). Automatically Generating Game Tactics via Evolutionary Learning. *AI Magazine*, 27(3):75–84.

Schrum, J., Karpov, I. V., and Miikkulainen, R. (2012). *Humanlike Combat Behavior via Multiobjective Neuroevolution*, 119–150. Berlin: Springer.

Schrum, J., and Miikkulainen, R. (2014). Evolving Multimodal Behavior With Modular Neural Networks in Ms. Pac-Man. In *Proceedings of the 16th Annual Conference on Genetic and Evolutionary Computation*, GECCO'14, 325–332. New York: ACM.

Sims, K. (1994). Evolving Virtual Creatures. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH'94, 15–22. New York: ACM.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Evolving Neural Network Agents in the NERO Video Game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, CIG'05. Piscataway: IEEE.

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A Hypercube-based Encoding for Evolving Large-scale Neural Networks. *Artificial Life*, 15(2):185–212.

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

Togelius, J., Karakovskiy, S., Koutnik, J., and Schmidhuber, J. (2009). Super Mario Evolution. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, CIG'09, 156–161. Piscataway: IEEE.

Verbancsics, P., and Stanley, K. O. (2010). Transfer Learning Through Indirect Encoding. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO'10, 547–554. New York, NY, USA: ACM.