Towards More Intelligent Adaptive Video Game Agents: A Computational Intelligence Perspective

Simon M. Lucas
University of Essex
Wivenhoe Park
Colchester CO4 3SQ
00 44 1206 872048
sml@essex.ac.uk

Philipp Rohlfshagen
University of Essex
Wivenhoe Park
Colchester CO4 3SQ
00 44 1206 874444
prohlf@essex.ac.uk

Diego Perez
University of Essex
Wivenhoe Park
Colchester CO4 3SQ
00 44 1206 874444
dperez@essex.ac.uk

ABSTRACT
This paper provides a computational intelligence perspective on the design of intelligent video game agents. The paper explains why this is an interesting area to research, and outlines the most promising approaches to date, including evolution, temporal difference learning and Monte Carlo Tree Search. Strengths and weaknesses of each approach are identified, and some research directions are outlined that may soon lead to significantly improved video game agents with lower development costs.

Categories and Subject Descriptors
A.1.2 [Artificial Intelligence]: *Applications and expert systems – games*. G.3 [Probability and Statistics]: *Probabilistic algorithms, including Monte Carlo.* I.2.6 [Learning]: *Connectionism and neural nets, Parameter learning.*

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
Games, Artificial Intelligence, Computational Intelligence, Monte Carlo Tree Search, Evolutionary Algorithms, Temporal Difference Learning.

1. INTRODUCTION
This paper describes a promising approach towards building intelligent adaptive video game agents. The aim is to design an architecture that can be used to provide a variety of intelligent capabilities across a range of games, with a minimum of human design input required for achieving acceptable performance on each individual game. The computational intelligence (CI) approach involves a minimum of game-specific programming. Instead, the main idea behind CI methods is that the intelligence emerges from the statistics of many simple low-level interactions, whether these be activations in a neural network, hypothetical actions explored in Monte Carlo Tree Search, or parameters adjusted to optimize a reward signal while performing Temporal Difference Learning.

CF'12, May 15–17, 2012, Cagliari, Italy.

To give a human-oriented perspective on the type of intelligence we are aiming for, imagine the task of learning to play a video game to a reasonable standard without any prior knowledge of the game and without explicitly knowing the rules. This is the task typically faced by human players of video games, exemplified by the classic arcade games of the 1980's. With this long-term aim in mind, we also have a significantly simpler version of the problem where the agent has access to the complete game state and the forward model, and hence is able to construct and search game trees using the forward model. This is something that a human player does not have access to, but can be used to significantly simplify the problem of generating intelligent adaptive game agents. Part of being an expert human player may involve constructing an approximate forward model, but this in itself is a major challenge.

There are a number of good reasons for investigating more adaptive game agent AI, including the following:

- Self-learning or adaptive agents are one of the long-term grand challenges of AI, and games provide an excellent test bed on which to evaluate such agents. In addition to providing challenges of wide-ranging complexity, games also enable humans to interact with AI agents in many of these scenarios.
- Incorporating intelligent agents into games could provide players with a more immersive experience, with the spine-tingling feeling of competing against intelligent beings, whether they display human-like intelligence or some strange alien type of intelligence; the commercial opportunities are immense.
- Provision of reasonable-performance intelligent agents with no (or minimal) programming effort is useful in the design, evaluation and testing of procedurally generated game content, such as game-levels, weapons and vehicles.

Regarding the latter point, a recent review of procedural content generation for games can be found in [1]. The idea of using the ability of agents to learn to play a game was explored by Togelius and Schmidhuber [19]. More recently, Tozour[1] has been evolving scripted agents to aid in the design process of a robot tower defence game. Clearly, there is much potential for using automatically designed agents in this way, and as the agents become smarter so the potential for exploitation will increase.

[1] http://aigamedev.com/open/interview/evolution-in-cityconquest/

Given that this is a useful and interesting endeavor, the question arises of the best way to create such adaptive agents. Although no one has yet done this, many of the enabling technologies are becoming increasingly mature. This paper presents a perspective on which are the essential and desirable techniques, and how they can be used in conjunction with each other.

2. TECHNIQUES

This section describes the main computational intelligence techniques which have an important role to play in the construction of adaptive game agents, together with discussion of their strengths and weaknesses.

2.1 Evolutionary Algorithms

Evolutionary algorithms (EA) are one of the most popular approaches for adapting an agent to perform well on a problem, and they are one of the easiest to deploy. Unfortunately, it is also very easy to get poor or mediocre results with an EA, and a great deal depends on the choice of representation, and other details. The process is as follows:

1. Design a representation.
2. Design a fitness function.
3. Choose an evolutionary algorithm.
4. Run the algorithm and save a selection of the best or most interesting evolved agents.

Although this can indeed be very simple, there is ample opportunity for expertise and innovation in the above steps, especially in steps 1 and 2. For step 3, Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [8] is a good default choice if the adaptive elements of the agent can be coded as a vector of real numbers, which is the case when evolving game agents based on neural networks and many other agent architectures where some real-valued parameters control aspects of an agent's behavior.

Note that the above process focuses on using an EA rather than on designing one, which is why there is no mention of the variation operators (e.g. mutation and crossover). When using an algorithm such as CMA-ES, those details are the responsibility of the algorithm.

There is a good deal of skill in designing an appropriate representation, and one should be aware of the limitations of EAs and what can realistically be achieved within a given number of fitness evaluations. One of the simplest problems for an EA is the standard one-max problem: the aim is to evolve a bit string of length B consisting entirely of ones, where the fitness is given as the number of ones in the string. EAs solve this problem for a bit string of length B in an expected $B\ log\_2(B)$ number of fitness evaluations, learning B bits of information in the process. When co-evolving agents, however, it is common to evaluate a population of N players by playing a full round-robin league of (approximately) N-squared games. If single parent selection is used, the identity of the winning player can be coded in log_2(N) bits, which places an upper bound on the information gained from that number of games. In practice it is hard to get close to this upper bound [11] even for simple games. Furthermore, evolution is sensitive to the representation used. For example, [10] found evolution to perform relatively well when evolving multi-layer perceptrons, but extremely poorly when evolving interpolated table functions, due to epistasis in the representation.

General discussion of using evolutionary algorithms in conjunction with games can be found in [13],[12].

2.1.1 *Evolution versus Coevolution*

For single-player games evolution can be used directly to evolve agents, using the game score as a fitness function. By single-player games we include games that involve any number of non-adaptive or generally "not very smart" opponents, such as the enemies in Super Mario or the ghosts in the original version of Ms Pac-Man. In the latter case the ghosts do chase the player while exhibiting some non-determinism (so learning a fixed route for example is ineffective), but some reasonably effective strategies can be learned without even using game-tree search.

However, for evenly balanced two-player games, including classic board games, coevolution offers a more interesting approach than straight evolution if the aim is to generate strong players without using an existing strong player to compete against. Coevolution has the potential to create strong players where none previously existed, whereas using evolution to evolve strong players would require an existing strong player to play against. If evolution is used to evolve agents against a weak player then the evolutionary process will just do enough to beat the weak player convincingly and then have no incentive to progress any further.

Coevolution solves this problem by using a relative fitness function: fitness is estimated by the playing performance of each player against other players in the current population, and perhaps also against players in a dynamically created "hall of fame" archive created from the best players found during an evolutionary run. In principle coevolution could produce a long-running arms race culminating in players that eventually solve the game at hand. However, there are several reasons why this rarely happens in practice, including:

- Limitations imposed by the chosen representation. For example, if a value function is being evolved as a weighted combination of some simple game features, then there is a limit to how smart this could ever be.
- Intransitivities in the population of game players. This leads to problems in measuring the fitness of an individual agent. An agent may appear very strong with respect to the current population, but actually be the weakest member of the current population when in competition with a different set of players. This problem can only occur when the players of a game exhibit intransitivities. The extent to which intransitivities exist depends not only on the nature of the game, but also on the nature of the players.
- Insufficient number of games to effectively train parameters. This arises when the parameter space is large e.g. an N-Tuple network for playing Othello may have thousands or tens of thousands of weights. To learn good values of these weights could require millions of fitness evaluations, and an even larger number of games played.
- Noisy or inaccurate fitness evaluations caused by games with random elements, or even in games of perfect information where games may be played from many different states in order to gain a more accurate picture of the relative merits of each player.

- The search space induced by the chosen representation may be difficult to search, containing many local optima or neutral plateaus.

Note that only the problem of intransitivities between the agents is unique to co-evolution; the other problems pose an equal threat to non-co-evolutionary EAs.

Interestingly, it is possible to design experiments to test which of these problems is most serious for a given combination of game and player architecture. For example, if intransitivities are suspected as being the main problem in co-evolving a game agent, then the same experimental setup can be used with the exception of replacing the relative (co-evolutionary) fitness measure with one based on playing against a controlled strength agent: a strong agent that is artificially weakened using forced random moves to always match the level of the evolving agents, such that the evolving agents win 50% of games on average. This removes any problems caused by intransitivities; if evolution still fails, then one of the other problems listed above could be to blame.

2.2 Reinforcement Learning (RL)

Although technically EAs could be placed within a broad RL umbrella in the sense that they aim to improve over time with respect to some reward function, in practice there are clear differences in how each approach is normally applied. Classic EAs operate at the population level, and measure "bottom-line" fitness i.e. how well an agent performs on a complete task or set of tasks. Conversely, classic RL algorithms such as Temporal Difference Learning (TDL) operate at the level of an individual: an individual modifies its behavior during its lifetime to improve its expected reward, which may be given at the end of each task or during a task. Temporal difference learning works outstandingly well on small toy problems where the game states can be exactly enumerated in a table. In such cases learning then corresponds to estimating the value of each table entry. For most games of interest the state space is either discrete and large, or continuous, and this direct tabular representation cannot be applied. In such cases some form of function approximation must be used, and this can be fraught with difficulty. Choosing the correct form of function approximator is of critical importance and can mean the difference between success and failure.

TDL is also sensitive to parameter choices such as the learning rate. Recent approaches such as Least Squares TD (LSTD) work in batch mode and choose a locally optimal step size for each parameter updated, in the sense of minimizing the mean square error (MSE). Interestingly, recent results [Thomas Runarsson, personal communication] indicate that the common practice of using TDL to minimize the MSE may be far from optimal for game playing. When playing a game, what matters is the action selected at each stage. The actual state value or state-action value estimates are not what really matters: they only matter as a means to select the correct action. For this reason there has been interest in applying preference learning to this problem instead. In preference learning, the aim is to learn the correct decisions directly rather than estimate the expected rewards for each action.

2.3 Monte Carlo Tree Search (MCTS)

Computer chess players have played at super-human levels for over a decade, and during that time Go has been one of the main challenges for reaching or surpassing expert human performance. For many years progress on Computer Go had been rather slow, and reaching expert levels of human play seemed many decades away. MCTS changed all that, causing a radical improvement in performance. The best MCTS-based players are now on a par with the best human players for the smaller 9 x 9 version of the game, and are making good progress on the full size 19 x 19 game. This has naturally sparked a great deal of interest in researching other games that MCTS might be good for, and already it has achieved dominance on connection games such as Hex and Y [1]. For a comprehensive survey see [3]. MCTS is also the leading approach to general game playing.

MCTS builds a game tree selectively by performing random simulations (also known as roll-outs) from a game state to predict the value of being in that state. This is depicted in Figure 1 (from [2]). The tree is grown selectively. A node in the tree is selected for expansion using a tree policy to navigate down the tree (shown as the bold line on the left tree). The Upper Confidence Bounds for Trees (UCT) formula [9] is often used to guide child selection while navigating the tree. UCT aims to optimally balance the opposing needs of exploration versus exploitation, though it is usually used in conjunction with some heuristics to achieve better performance.

A random simulation (also known as roll-out or play-out) is then made from the selected leaf node of the tree. The roll-out normally continues to the end of the game, at which point the exact value is known. This value is then propagated up the tree, and a new leaf node is added where the roll-out was made, as shown on the right tree in the figure. The roll-out may be made by choosing uniform random moves, or may be biased towards more favorable moves.

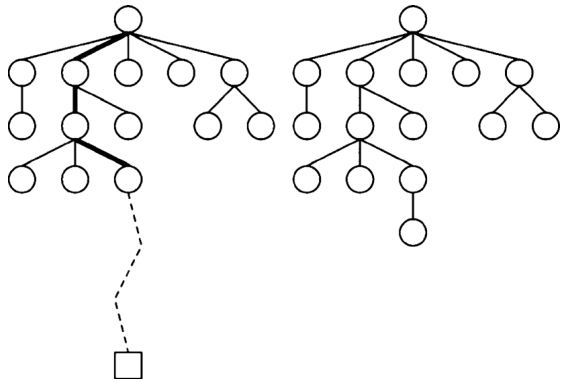

Figure 1: Illustration of how MCTS operates (from [2]).

MCTS has many attractive properties, such as being an anytime algorithm where playing performance typically increases (in some cases logarithmically e.g. [5]) with the number of roll-outs that can be performed given the available time. Perhaps more importantly, MCTS can be used in the absence of any good heuristic evaluation function. The fact that MCTS works at all is at least a little surprising: good players do not play randomly, so why should random simulations which sample only a tiny fraction of the search space provide any clue as to which move is good? Nonetheless, there is clearly important information in the roll-out statistics; at least enough to outperform non MCTS approaches. Furthermore, the tree grows with each roll-out, and increasingly represents more meaningful information. On tasks as difficult as playing Go, it is not that MCTS plays anywhere close to optimal, it is more the case that the problem is hard for any type of agent, and MCTS performs well compared to the competition (though not as well as expert humans yet on the full-size game).

Of more interest to the current paper is whether MCTS can be used to endow video game agents with more intelligence. Many

researchers and games industry insiders have questioned the value of this, imagining that an intelligent opponent would be boring to play against because it would simply thrash the human player every time. This is not so, however, for at least three reasons. One is that the intelligent opponent might have a different objective other than winning; for example, it might be aiming to maximize the human player's fun, much as a parent aims to do when playing a game with their child. Secondly, the game can be re-balanced in other ways: it might be fun to play against super-intelligent opponents that are limited in their physical strength, mobility, firepower, health or armor. Thirdly, the aim may be to create an intelligent agent that is not competing against the player, but acts as a partner or an assistant. The illusion of intelligence in this type of agent is crucial, and can be more important than the intelligence of the enemies. For instance, in first person shooter games, a companion may be a partner of the player during the whole game, while typical enemies appear on screen for only 5 seconds on average.[2]

There are some challenges to be overcome in using MCTS to boost the intelligence of video game agents, including making enough roll-outs in the severely limited time available to compute each action, and coping with the long roll-out depth needed to make progress in the game. Depending on the type of game, the game state for a video game may be significantly more complex than for a classic board game, involving many continuous variables describing the position and velocity of each agent. To ameliorate this it may be possible to use a simplified model of the game, or to represent the game state efficiently. For example, when using MCTS to control a Pac-Man agent, the game state can be represented compactly using bit-sets to model the state of each pill (which can change from *available* to *eaten*), and then maintaining a separate data structure of pill positions, which for a given map never changes. In this way it is possible to perform hundreds of roll-outs for each game tick. Further improvements can be made by keeping part of the tree from one game tick to the next (the branch that corresponds to the selected action may be kept). In this way actions may be selected on the basis of tens of thousands of roll-outs even though only a few hundred are made per game tick.

When applying MCTS to video games a significant problem is to decide the value that should be fed back at the end of a roll-out. For games such as Go this is not a problem: each roll-out ends in a terminal state of the game at which time the value is known exactly: either 1 or 0 (either a win or a loss for the current player). The value at each tree node then approximates the probability of winning from that node. In the case of video games, the situation is less clear. Due to the nature of the game, most roll-outs will not end in a terminal state, and some heuristic value must be constructed to estimate the value of a state. Finding a good heuristic is a significant problem, and for this EAs or TDL can be used; so far TDL has been used, but EAs would seem to offer an interesting alternative. Interestingly, there are three distinct ways in which heuristics can be applied within MCTS. They can be used to inform the tree-policy, and/or to bias the roll-outs, or to (as already mentioned) provide a heuristic value at the end of a roll-out (for the frequent cases where the true value is not obvious). Even in arcade games such as Pac-Man, where the score is updated every time a pill, edible ghost, or fruit is eaten, heuristics play an important part in evaluating game states, since many states with identical scores will have very different true values for the Pac-Man agent.

Without a good heuristic MCTS may fail if applied naively, largely because the vast majority of roll-outs do not do anything of interest. If at each game tick a video game character performs an action selected uniformly at random from the set of available actions, then most roll-outs will not do anything interesting at all, but just dither and not move much. There are a number of ways of overcoming this problem, including choosing a higher-level action space (i.e. a space of macro-actions where each high-level action then has to be translated to a sequence of lower-level actions). Another simpler way is to bias the roll-outs to increase the likelihood that the previous action is repeated.

3. Evaluation and Competitions

One of the most important driving forces behind progress in this area has been regular and rigorous evaluation. For many games there are regular competitions. These provide an ideal means by which to test any number of approaches, and to tune each approach to see which works best in practice. While rigorous evaluation has been the feature of many research communities, this has been embraced with particular enthusiasm in games.

Evaluation in pattern recognition and machine learning normally involves measuring performance on some pattern classification or prediction problem, where the correct answers are already known. In contrast to this, intelligent game-playing agents need to work out for themselves what actions to take in novel situations where no supervised training data exists. Furthermore, game playing algorithms usually compete under strict time limits, so an appropriate balance must be found between optimality and timeliness. Games naturally promote techniques which work well in practice. The remainder of this section discusses two game competitions that are of particular relevance to this paper. The first example, general game playing, is another area where MCTS has proven to be very successful, and the general aspects of this have some relevance to developing general purpose video game agents. The second example, the physical travelling salesman problem is a simple video game being run as an open competition where naïve MCTS only achieves limited success, but more sophisticated MCTS approaches are already showing great promise.

3.1 General Game Playing (GGP)

In focusing on a single game there is a danger that the results will be of limited interest to the goal of developing a general purpose AI agent. This danger may sometimes be overstated, since the AI community has learnt a great deal over the years with results from specific games often having a more general impact. However, the fact remains that achieving high performance on a particular game can involve an enormous amount of game-specific hand tuning. Hence GGP [7] was developed as a way to make games a true challenge for machine learning. GGP games operate in two phases. In the first phase the game rules (specified in a type of first-order logic) are given to each player in order that it can analyse the rules, potentially do some learning about the game, set up any data structures etc. In the second phase play commences and continues until the end of the game. MCTS now seems to be the dominant algorithm in GGP, with the AAAI 2007 and 2008 competitions being won by CadiaPlayer [6], an MCTS-based player, and the 2009 competition being won by Ary [14], another

[2] Mikael Hedberg, AI Game Dev Conference 2010, discussing the AI of Battlefield: Bad Company 2, for details see: http://aigamedev.com/open/coverage/paris10-report/#session10.

agent with a significant MCTS component. GGP as it stands offers a fascinating challenge, but its use of a logic-based game description language naturally tailors it toward certain types of game (essentially mind-games), and means it is not appropriate for video or physics-based games. Developing a type of GGP system for this type of game is an interesting possibility, and it remains to be seen which type of algorithm would perform best on this type of problem.

3.2 Physical Travelling Salesman Problem

This competition [15],[16] combines aspects of the classic Travelling Salesmen Problem (TSP) with aspects of vehicle driving (physics) – hence the Physical Travelling Salesman Problem (PTSP). The aim is simply to visit all cities in the minimum time, but the salesman is now driving a physical object and has momentum and steering to take care of: in most cases the optimal TSP city order is very different to the optimal PTSP city order. Figure 2 shows a sample map from the current IEEE World Congress on Computational Intelligence Competition [15].

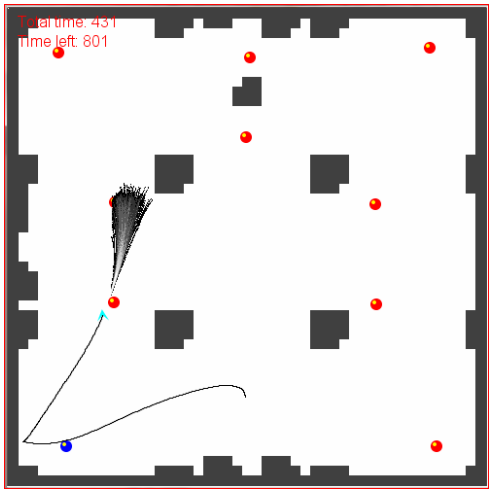

Figure 2: A sample map from a currently running Physical Travelling Salesman Problem (PTSP) Competition.

Applying MCTS to a simple 2D navigation game such as the PTSP provides an ideal way to study its weaknesses. Since the roll-outs can be overlaid on top of the map, it is easy to see how far MCTS is exploring ahead, and whether it is able to incorporate long-term planning considerations into its solutions, or whether it is acting in a greedy manner. If MCTS is applied in its most basic form to this problem, then it tends to do the latter, as also shown in Figure 2. In the standard PTSP configuration, actions are very low level, and specify a force vector to be applied for the next time instant. A good solution for the map in Figure 2 would involve more than 1,000 such actions. Interestingly, if MCTS is used with a higher-level action space [Whitehouse and Powley, personal communication] then this problem is alleviated, and good performance can be obtained. Such an approach is currently leading the rankings on the Human versus Bot version of the PTSP (http://ptsp-game.net/). This is of particular interest here, since it is a type of innovation (i.e. using macro-actions rather than actions from the original more fine-grained set) that could potentially be created through evolutionary adaptation, but not through temporal difference learning. Another way to achieve long-term planning in the PTSP is to explicitly solve the problem in two steps, where one step optimizes the order of cities to visit and then second step works tries to find the best action sequence to drive that route.

Given that the PTSP is a one-player game (at least in its current form) it would also be interesting to investigate the use of MCTS algorithms that have already been shown to work well on one-player games, such as nested MCTS and Monte-Carlo Beam Search [4].

4. Proposed Approach

Based on the above discussion, the architecture shown in Figure 3 is proposed for a general purpose intelligent game agent generation system. The system involves a population of MCTS game agents which evolves over time. Evolution offers a very flexible way to do this and can easily incorporate major architectural changes. Changes could include the nature of the function approximators used in the agent, such as multi-layer perceptrons or interpolated table functions. Reinforcement learning algorithms such as TDL are unable to do this: they are restricted to adapting a fixed-size parameter vector. Each agent is controlled by a number of parameters, including things such as roll-out depth, the value of the UCT exploration constant, plus many other variables controlling the behavior of the MCTS algorithm. These can be adapted using evolution.

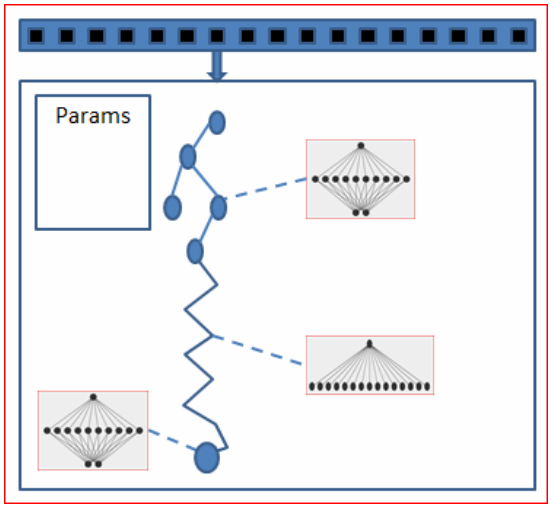

Figure 3: Proposed adaptive MCTS agent architecture. The system evolves a population of MCTS players: major structural changes occur at the evolutionary level. Each player also has many parameters that are adapted during game play. Function approximators help control tree policy, roll-out policy, and heuristic values given to terminal nodes of roll-out which are not terminal game states.

For MCTS to be effective, heuristic value functions (expressed in some form of function approximator, which in the simple case could be a weighted sum of features) are often very important. For video game agents these can be applied in three ways: in the tree policy to help select which node to expand, in the roll-out policy to bias the roll-out to more interesting states, and in the case that a roll-out does not reach a terminal node of the game, to place a heuristic value on that state.

How best to update these is an interesting problem. Silver et al [18] incorporate TDL within MCTS to good effect. Robles et al [17] also found TDL could improve performance by updating a heuristic function both for the tree-policy and for guiding the roll-outs. However, given what was mentioned earlier about the value

of preference learning and focusing on actions made rather than mean-square error, alternative approaches are also of great interest. One such approach would be to use an evolutionary algorithm to update the heuristic functions during the execution of the MCTS algorithm. For example, roll-out bias heuristics could be evaluated on the quality of end state that they tend to reach. In this way each roll-out informs the fitness function, and extremely rapid evolution may be possible. Other types of adaptation that TDL is ill-suited to deal with and are best tackled using evolutionary approaches include adapting the temporal resolution of actions (e.g. repeating each movement action *N* times).

5. Conclusions

This paper discussed the motivation behind developing more intelligent and adaptive video game agents, and described the main research areas needed for this, namely evolution, reinforcement learning and Monte Carlo Tree Search. Some of the strengths and weaknesses of each approach were identified, and placed in the context of some recent game-based competitions. A game agent architecture was proposed, incorporating elements of evolutionary design, temporal difference learning, and Monte Carlo Tree Search. The complete architecture is still a work in progress, but many of the components have been rigorously and independently shown to work in many different games and other domains. The next step is to integrate these into an effective system, able to control agents in a variety of video games with a minimum of programming effort. Although the big-budget game studios have been reluctant to use many statistical AI methods such as evolutionary algorithms and neural networks, there is a burgeoning market for mobile and casual games, and this offers an ideal testing ground for releasing these agents into the wild.

6. ACKNOWLEDGMENTS

Thanks go to members of the Game Intelligence Group at the University of Essex for extensive discussions of many of the ideas in this paper. This work was funded by EPSRC Grant EP/H048588/1: UCT for Games and Beyond.

7. REFERENCES

[1] Arneson, B., Hayward, R.B. and Henderson, P., 2010, Monte Carlo Tree Search in Hex, *IEEE Transactions on Computational Intelligence and AI in Games*, vol.2, no. 4, pp.251-258.

[2] H. Baier and P. D. Drake, 2010, The power of forgetting: Improving the last good reply policy in Monte Carlo Go, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 303–309.

[3] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., 2012, A Survey of Monte Carlo Tree Search Methods, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp.1–43.

[4] Cazenave, T., 2012, Monte Carlo Beam Search, *IEEE Transactions on Computational Intelligence and AI in Games*, vol.4, no. 1, pp.68-72.

[5] Enzenberger, M., Müller, M., Arneson, B., Segal, R., 2010, Fuego—An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search, *IEEE Transactions on Computational Intelligence and AI in Games*, vol.2, no.4, pp.259-270.

[6] Y. Bjornsson and H. Finnsson, 2009, Cadiaplayer: A simulation-based general game player, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 4 –15.

[7] M. R. Genesereth, N. Love, and B. Pell, 2005, General game playing: Overview of the AAAI competition, *AI Magazine*, no. 2, pp. 62 - 72.

[8] N. Hansen, S. Mueller, and P. Koumoutsakos, 2003, Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES), *Evolutionary Computation*, vol. 11, pp. 1-18.

[9] L. Kocsis and C. Szepesvári, 2006, Bandit based Monte-Carlo planning, in *Proceedings of European Conference on Machine Learning*, Berlin, Germany, pp. 282–293.

[10] Lucas, S.M., 2010, Estimating Learning Rates in Evolution and TDL: Results on a Simple Grid-World Problem, *IEEE Conference on Computational Intelligence and Games*, pp. 372-379.

[11] Lucas, S.M., 2008, Investigating Learning Rates for Evolution and Temporal Difference Learning, *IEEE Symposium on Computational Intelligence and Games*.

[12] Lucas, S.M., 2008, Computational Intelligence and Games: Challenges and Opportunities, *International Journal of Automation and Computing*, vol. 5, pages: 45 – 57.

[13] Lucas, S.M. and Kendall, G., 2006, Evolutionary Computation and Games, *IEEE Computational Intelligence Magazine*, vol. 1, pages: 10 – 18.

[14] Méhat, J. and Cazenave, T., 2010, Combining UCT and Nested Monte-Carlo Search for Single-Player General Game Playing, *IEEE Transactions on Computational Intelligence and AI in Games* vol. 2, pp. 271-277.

[15] Perez, D., Rohlfshagen, R. and Lucas, S.M., 2012, The Physical Travelling Salesman Problem: WCCI 2012 Competition, *IEEE Congress on Evolutionary Computation*, to appear.

[16] Perez, D., Rohlfshagen, R. and Lucas, S.M., 2012, Monte-Carlo Tree Search for the Physical Travelling Salesman Problem, *Proceedings of EvoGames*, to appear.

[17] Robles, D, Rohlfshagen, P and Lucas, S.M., 2011, Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search, *IEEE Conference on Computational Intelligence and Games*, pp. 305 – 312.

[18] Silver, D, Sutton, R.S. and Müller, M., 2008, Sample-based learning and search with permanent and transient memories, in *Proceedings of 25th Annual International Conference on Machine Learning*, Helsinki, Finland, pp. 968–975.

[19] Togelius, J. and Schmidhuber, J., 2008, An Experiment in Automatic Game Design. *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 111-118.

[20] Togelius, J., Yannakakis, G.N., Stanley, K.O. and Browne, C., 2011, Search-Based Procedural Content Generation: A Taxonomy and Survey, *IEEE Transactions on Computational Intelligence and AI in Games*, vol.3, no.3, pp.172-186.