Evolving Behaviour Trees for the Commercial Game DEFCON

Chong-U Lim, Robin Baumgarten and Simon Colton

Computational Creativity Group
Department of Computing, Imperial College, London
www.doc.ic.ac.uk/ccg

Abstract. Behaviour trees provide the possibility of improving on existing Artificial Intelligence techniques in games by being simple to implement, scalable, able to handle the complexity of games, and modular to improve reusability. This ultimately improves the development process for designing automated game players. We cover here the use of behaviour trees to design and develop an AI-controlled player for the commercial real-time strategy game DEFCON. In particular, we evolved behaviour trees to develop a competitive player which was able to outperform the game's original AI-bot more than 50% of the time. We aim to highlight the potential for evolving behaviour trees as a practical approach to developing AI-bots in games.

1 Introduction

The ability of Artificial Intelligence methods in games to deliver an engaging experience has become an important aspect of game development in the industry, and as such, numerous techniques have been developed in order to deliver realistic game AI. However, as Jeff Orkin remarked, that if the audience of the Game Developers Conference were to be polled on the most common A.I techniques applied to games, one of the top answers would be Finite State Machines (FSMs) [11]. Behaviour trees have been proposed as an improvement over FSMs for designing game AI. Their advantages over traditional AI approaches are being simple to design and implement, scalability when games get larger and more complex, and modularity to aid reusability and portability. Behaviour trees have recently been adopted for controlling AI behaviours in commercial games such as first-person-shooter Halo2 [8] and life-simulation game Spore [7].

We investigate here the feasibility of applying evolutionary techniques to develop competitive AI-bots to play commercial video games. The utility of this is two-fold, i.e. to enable intelligent agents to compete against human players in 1-player modes of games, and to act as avatars for players when they are not able to play themselves (e.g. as temporary substitutes in multiplayer games). The application of genetic programming has seen positive results in the fields such as robotic games [10] and board games like Chess [6], as well as racing track generation [13]. It has also been used to evolve human-competitive artificial players for Quake3 [12], and real-time strategy games [5]. By investigating the feasibility of applying an evolutionary approach with behaviours trees to develop a competitive automated AI player for a commercial game, we hope to further

exemplify it as a viable means of AI development that may be adopted by the video games industry. We demonstrate this by evolving a competitive automated player for the game DEFCON, which is able to beat the hand-designed AI-bot written by the programmers at Introversion Software Ltd. more than 50% of the time over a large number of games. We provide relevant background reading in section 2, outlining DEFCON and behaviour trees. Next, we describe how evolution is applied to behaviour trees in section 3, and provide a description of the different fitness functions we employed in section 4. The experiments conducted and results obtained are in section 5, and we conclude and propose areas where future work may be applicable in section 6.

2 Background

2.1 DEFCON

DEFCON[1] is a commercial multiplayer real-time strategy game that allows players to take the roles of the military commanders of one of six possible world territories. Players are given a set of units and structures at their disposal, and have to manage these resources and inflict the greater amount of damage against opposing players. The game is split into 5 discrete time intervals (DEFCON5 to DEFCON1), and each dictate the movements and manipulation of units that are allowed. There are 7 available **territories** in each game, with each controlled by up to 1 party. A **party** represents the player, either human or AI-controlled, each allocated a fixed quantity of units that it may place and make use of throughout the course of the game. Playing the game involves strategic planning and decision making in coordinating all these units and winning by attaining the highest score, calculated via various scoring modes. We used the default scoring mode: 2 points awarded for every million of the opponent's population killed and a point penalty for every million people belonging to the player lost.

Several implementations of automated players exist for DEFCON, and we refer to these as **AI-bots**. For example, the default bot that comes with DEFCON is a deterministic, finite-state-machine driven bot. It consists of a set of 5 states and transits from one state to the next in sequence. Upon reaching the final state, it remains in it until the end of the game. In 2007, an AI-bot was developed by Baumgarten [2] using a combination of case-based reasoning, decision tree algorithms and hierarchical planning. For the case-based reasoning system, high-level strategy plans for matches were automatically created by querying a case base of recorded matches and building a plan as a decision tree, which classified each case's plan dictating the placement of fleets and units, and the sequence of movements and attacks. The starting territories of each player were used as a similarity measure for retrieving a case. The results, plans, and structure information were retained as a case in the case base at the end of a match,

[1] The official DEFCON website is here: http://www.introversion.co.uk/defcon

to enable an iterative learning process. Furthermore, the low-level behaviour of units, such as precisely timed bomber attacks and ship manoeuvres within fleets were added to improve the tactical strength of the AI-bot. As a result, in certain configurations, these low level modifications were able to influence the game outcome significantly, resulting in a victory for Baumgarten's AI-bot over the Introversion implementation in roughly 7 out of 10 matches on average.

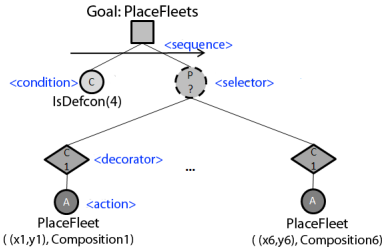


Fig. 1. Behaviour Tree: PlaceFleets

Both AI-bots covered above were implemented using the source-code of DEFCON, which gave it access to game state information that was required for various calculations. The implementation of the AI-bot described here made use of the an application programming interface (API) [2] that allows the control of a player in DEFCON by an external AI-bot, which can retrieve and invoke method calls on DEFCON, providing a way for developers to build AI-bots without having to work with the source-code of DEFCON directly. We avoided dictating low-level strategic decisions in our implementation but still successfully managed to evolve the behaviour trees to produce a competitive AI-bot with little human intervention. In a DEFCON competition competing API-designed AI-bots held at the Computational Intelligence and Games conference in Milan in August 2009, our AI-bot emerged victorious (although it was a rather hollow victory, as we were the only entrant to the inaugural competition!)

2.2 Behaviour Trees

A traditional approach to developing AI-controlled players for games has been to use Finite State Machines (FSMs). The problem with FSMs is that as the AI-bot grows in complexity, the number of states and transitions between them grows exponentially with the size of the game, making it difficult to manage. Even though Hierarchical Finite State Machines (HSFMs) overcome this, reusability is often a problem. Behaviour trees provide a simple, scalable and modular solution to embody complex AI behaviours. Each tree is goal-oriented, i.e. associated with a distinct, high-level goal which it attempts to achieve. These trees can be linked together with one another, allowing the implementation of complex behaviours by first defining smaller, sub-behaviours. Behaviour trees are constructed from 2 types of constructs. Firstly, primitive constructs form the leaves of the tree, and define low level actions which describe the overall behaviour. They are classified into 2 types, **actions**, which execute methods on the game, and **conditions**, which query the state of the game. Secondly, composite constructs can be used to group such primitive constructs to perform a higher-level function. the 3 main types of composites are **sequences**, **selectors** and **decorators**.

[2] API and documentation available at: http://www.doc.ic.ac.uk/~rb1006/projects:api

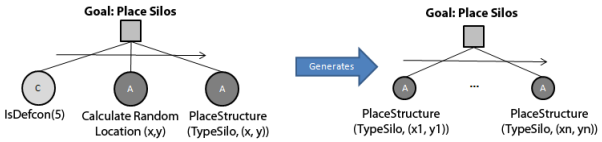


Fig. 2. Using a hand-crafted tree to generate a random behaviour tree

As an example, consider Figure 1 which shows a behaviour tree, with its nodes labelled to identify the constructs used. At the root, the tree has a **high-level goal** of placing a fleet. The **sequence** node dictates that, in order to achieve this goal, it has to first achieve the **sub-goal** of checking that the game is in DEFCON4 using a **condition**, followed by another composite sub-goal to place the fleet within the game. Thus, all child nodes must succeed for a **sequence** to succeed. The sub-goal is dictated by a **selector**, which will succeed as soon as one of its child nodes succeeds. Each child task begins with a **decorator**, which in this case acts as a counter to ensure that its child **action** node, used to place a fleet of a specified *composition* at location *(x,y)*, is only executed once. We read the tree as one that achieves the goal of placing a fleet by first checking that it is the correct point in the game to do so. Then, it selects one (x, y, composition) combination at random and executes it in the game. If that combination fails, it will try the next set combination until it has exhausted all of its options.

3 Evolving Behaviour Trees

We used behaviour trees to hand-craft an initial AI-bot to demonstrate that we could encode the basic abilities that a player would be able to perform. The AI-bot was given sufficient functionality to execute the basic actions to play DEFCON. However, its decision of when and whether to apply the actions were performed randomly. Ultimately, we planned to evolve the behaviour trees of this random AI-bot by playing games against the default Introversion AI-bot and afterwards, extract the best performing behaviour trees in different areas before combining them to produce a competitive AI-bot overall.

3.1 Randomly Generating Behaviour Trees

To produce the original set of behaviour trees, we adopted Bryson's Behaviour Oriented Design approach [3, 4]. We define a high-level goal for the AI-bot before subsequently breaking it down into smaller sub-tasks that would form the basis of the required behaviour trees. This iterative process identifies building blocks of which to define more complex behaviours upon. Figure 2 shows how the hand-crafted tree on the left was used to produce a new tree on the right for thepurpose of placing silo units randomly. The left tree checks whether it is the appropriate DEFCON level, selects a coordinate at random, and places the silo at that randomly chosen location. The resultant tree capable of placing silos at those locations is shown on the right. We vary the AI-bot's behaviour (i.e. the number and positions of silos placed) by choosing which of its branches to attach or remove. We later made use of evolution to make these choices. We continued

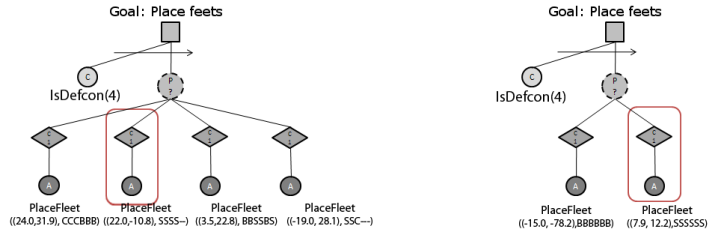


Fig. 3. Two behaviour trees with branches selected for recombination

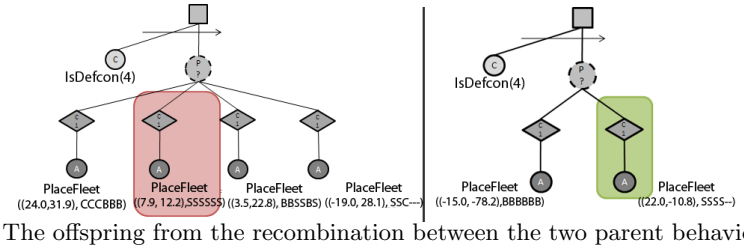


Fig. 4. The offspring from the recombination between the two parent behaviour trees.

with this approach to produce other behaviour trees that performed random decisions with regards to other aspects of playing the game.

3.2 Applying Genetic Operators

Trees are structures which genetic operators are naturally applicable to [9], with crossovers on branches and mutation on nodes. Figure 3 shows two behaviour trees which have the same goal of placing fleet units. Using the left behaviour tree as a reference, what occurs is a `sequence` that first checks if the current DEFCON level is 4, and then proceeds to the priority selector if it is so. Looking at the left most `action` node, it will attempt to place a fleet composed of 3 carriers and 3 battleships at longitude 24.0 and latitude 31.9. The parent counter `decorator` ensures that the action is only executed once.

Crossovers are applied to randomly selected portions of the trees. Figure 4 shows the resulting offspring from recombination.Instead of placing a second fleet at (22.0, -10.8) with a composition of 4 submarines, it now places a second fleet at (7.90, 12.2) with a composition of 6 submarines. Random mutation can be used to increase genetic diversity. In Figure 5, the green portion shows how incremental mutation might occur to the left behaviour tree of figure 4, resulting in a different topology. Instead of placing 4 fleet units, the AI-bot now places 5 fleet units. The location and fleet compositions used for the new branch (highlighted green) were generated randomly during mutation. The red portion shows how a point mutation might occur. Since behaviour trees are not strongly-typed for recombination, inferior offspring trees may result (i.e. placing units which the AI-bot doesn't possess, or in illegal positions.) These trees would presumably be naturally selected against as the system evolves. In section 6, we mention ways to extend this approach to produce richer and more descriptive behaviour trees.

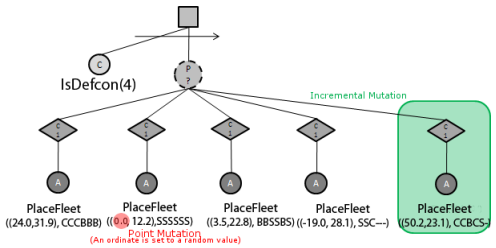


Fig. 5. Two types of mutations occurring in one of the behaviour trees

4 Fitness Functions

As mentioned previously, we evolved behaviour trees for individual behaviours, and combined the best performing trees into an overall AI-bot control mechanism. For the 4 behaviours, we used these fitness functions to drive the search:

• **Defence Potential of Silo Positions.** Silos are ground installations which play a significant role in both attack and defence. In both cases, a silo's effectiveness is affected not only by its absolute position, but also its position relative to the other silos. We chose to focus on the defensive aspect of the silos by measuring their **defence potential** – the total number of air units that were successfully destroyed for a given game.

• **Uncovered Enemy Units by Radars.** While radars do not have a direct influence on a player's attack or defensive abilities, they provide an indirect means of support by uncovering enemy units. This allows the AI-bot to then react appropriately to the situation, i.e. by sending fighters to enemy sea units or by shooting down missiles earlier. The coverage of the radars is determined by their positions, and we used the number of enemy sea units uncovered before the start of DEFCON1 as a fitness measure for evolving radar placement positions.

• **Fleet Movement & Composition.** The fitness of the fleet movement and composition in a game was calculated by evaluating the number of enemy buildings uncovered, the number of enemy buildings destroyed and the number of enemy units destroyed by the AI-bot. For each of these, a score was calculated and the average of the 3 scores taken as an overall fitness for the behaviour tree.

• **Timing of Attacks.** We used the difference between the final end-game scores as an indicator of how well the AI-bot performed for timing of attacks. A larger difference indicated a convincing win whereas a smaller difference would mean a narrower victory. We fixed the bounds for the maximum and minimum difference to +250 and -250 respectively (with these values found empirically through some initial experimentation). The fitness was calculated using the function:

$$fitness = \frac{difference_{score} - (-250)}{250 - (-250)} = \frac{(score(AIbot) - score(Enemy)) + 250}{500}$$

5 Experiments and Results

5.1 Experimental Setup

We chose the following game options for our experiments. Firstly, the starting territories for our AI-bot and the enemy were fixed as Africa and South America respectively. Secondly, the default game scoring was used. Four main experiments were performed, each evolving a set of AI-bots with the aim of improving the population AI-bot's performance as per the respective fitness functions described above. Each population contained 100 individuals, and each experiment was evolved between 80 to 250 generations. We employed a fitness proportionate selection method to choose pairs for recombination and set the mutation rate to 5%. These parameters were chosen after performing several initial experiments. Naturally, there is a vast array of other parameter settings we could have experimented with, but given the time constraints imposed by the number of experiments and running time of each game, we had to decide upon the values empirically. Via four evolutionary sessions, we evolved behaviour trees for:

1. Placement of silos to maximise defence against enemy air units
2. Placement of radar stations to maximise the number of enemy units detected throughout the course of the game
3. The placement, composition and movement of the player's fleet units to maximise their overall attack capabilities
4. The timings of the attacks for 4 types of attach, namely, submarine attacks using mid-range missiles, carrier bomber attacks using short range missiles, air base bomber attacks using short range missiles and silo attacks using long range missiles.

5.2 Distribution

Ultimately, the fitness functions rely on playing a game to completion. Unfortunately, with 4 experiments, each running for about 100 generations with 100 individuals in a population would require 40,000 game runs. With each game taking approximately 90 seconds to complete, a total time of 3.6 million seconds ($\sim$41 days) of continuous processing would be required for the project. To bring the time-frame down to $\sim$2 days per experiment, we distributed the running of DEFCON games over 20 computers, connected together via a local area network.

5.3 Results

For silo placements, Figure 6(a) shows that the mean fitness of the population increased over the generations. The mean number of missiles destroyed increases from around 70 to almost 100, and similarly, the mean number of bombers destroyed increases from about 18 to about 34. However, the mean number of fighters destroyed remained at around 18 across the generations, which we believe is due to the silo placement locations evolving further away and out of enemy

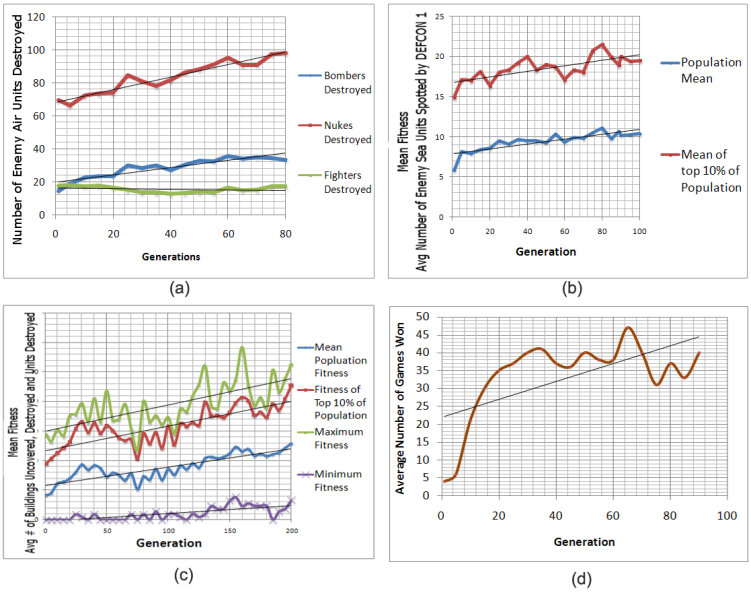
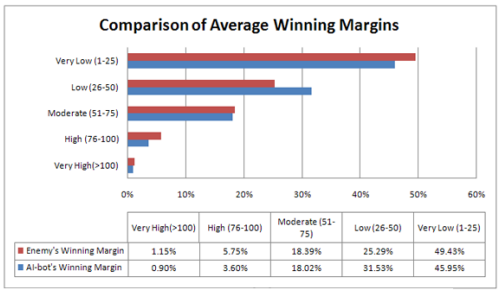


Fig. 6. Mean fitnesses over generations for each behaviour. (a) silo placement (b) radar placement (c) fleet coordination (d) attack timings

fighter scout range over the generations. For radar placements, we observe a similar increase in mean fitness. Figure 6(b) shows the mean number of detected enemy sea units increasing from around 6 to 10, with the mean fitness of the top 10% of the population even reaching 21.5 (∼90% of detectable enemies when excluding submarines) at generation 80. For fleet placement, composition and movement, Figure 6(c) shows an increase in the AI-bots' mean fitness as the AI-bot evolved over the generations, with the mean fitness increasing from about 8 to 26. Similarly, when evolving the attack timings for the AI-bot, Figure 6(d) shows an increase in the average number of games won. Initially having on average 4 wins in a population of 100 AI-bots, the number reached 47 at Generation 65. The average number of wins appears to be plateauing, which might indicate a need to continue the evolution over more generations.

We constructed an AI-bot with a controller that used the best trees evolved for each of the four behaviours. It was set to play 200 games against the default Introversion AI-bot. The difference between the scores obtained by both the AI-bot and the opponent was used as an indicator of how well the AI-bot had performed, which we term as the margin of victory. Prior to performing the evolution, the AI-bot which consisted of behaviour trees which performed random movements and choices (Section 3.1) managed to win around 3% of the time out of the 200 games. We ran the evolved AI-bot which beat the default Introversion AI-bot 55% of the time in 200 games. Figure 7 shows the distribution of the margins of victory in these matches. We note that in a large number of games, our AI-bot only lost to the opponent by a very low margin, indicating



	Very High(>100)	High (76-100)	Moderate (51-75)	Low (26-50)	Very Low (1-25)
Enemy's Winning Margin	1.15%	5.75%	18.39%	25.29%	49.43%
AI-bot's Winning Margin	0.90%	3.60%	18.02%	31.53%	45.95%

Fig. 7. Summary of the Performance of the Evolved AI-bot

that the the number of wins by our AI-bot could have been larger. The opponent managed to beat our AI-bot by a moderate to a very high margin fairly often, indicating more convincing victories. Using a binomial test at the 4% significance level with a 50% winning probability, we were unable to reject the null hypothesis that both AI-bots were equal in abilities.

6 Conclusions and Future Work

By evolving behaviour trees for individual behaviours and combining the trees into an overall AI-bot, we have been able to produce a competitive player that was capable of beating the original, hand-coded DEFCON AI-bot more than 50% of the time. This hints at the possibility that such an approach is indeed feasible in the development of automated players for commercial games. Speculating on the effect of further experimentation, we refer to the graphs in Figure 6. Although we have seen improvements after approximately 100 generations, we notice that the mean fitness seems to have reached a plateau, which might indicate that performing the evolution for a greater number of generations may not show significant improvements in mean fitness. This raises the question of whether evolutionary techniques need to be supplemented with other techniques in automating AI-bot design, and if so, which techniques should be investigated.

Evolving against a single opponent could have caused over-fitting. An improvement would be to perform experiments against other AI-bots or human players. Also, in the event that no training AI-bot was present, it raises the question of whether co-evolution [12] could have been applied. DEFCON is a non-deterministic game, especially when involving human players. We did not consider all possible permutations of starting locations for both the player and the opponent. Africa and South America are within close proximity, so other starting locations would increase the distance between players and might require different strategies. We also picked 4 tasks to concentrate on, but there are other game-play aspects such as the coordination of air units that could have been investigated. Our implementation only considered the transition between two stages, from being defensive to launching an attack, which had to occur in that order. It would be interesting to see the application of the evolution of sub-trees using `lookup` decorators to allow the AI-bot to exhibit complex behaviours and adaptive game play styles to match opponents, resulting in more

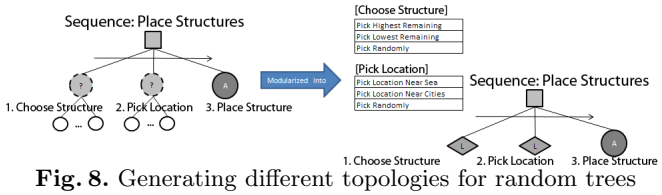


Fig. 8. Generating different topologies for random trees

descriptive behaviour trees. Figure 8 shows how the use of `decorators` can be used to generate behaviour trees which may differ in topology.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments.

References

1. C. Bauckhage and C. Thurau. Exploiting the fascination: Video games in machine learning research and education. In *Proceedings of the 2nd International Workshop in Computer Game Design and Technology*, 2004.
2. R. Baumgarten, S. Colton, and M. Morris. Combining AI Methods for Learning Bots in a Real-Time Strategy Game. *Int. J. of Computer Games Tech.*, 2009.
3. J. Bryson. Action selection and individuation in agent based modelling. In *Proceedings of the Argonne National Laboratories Agent Conference*, 2003.
4. J. Bryson. The behavior-oriented design of modular agent intelligence. In *Proceedings of the Agent Technologies, Infrastructures, Tools and Applications for E-Services Workshop*, LNCS 2592, Springer, 2003.
5. J. Hagelbäck and S. Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-agent Systems, volume 2*, 2008.
6. A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In *Proceedings of the 8th European Conference on Genetic Programming*, LNCS 3447, Springer, 2005.
7. C. Hecker, L. McHugh, M. Argenton, and M. Dyckhoff. Three Approaches to Halo-style Behavior Tree AI. Games Developer Conference, Audio Talk, 2007.
8. D. Isla. Managing complexity in the Halo 2 AI system. In *Proceedings of the Game Developers Conference*, 2005.
9. W. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic programming and evolvable machines*, 1(1/2):95–119, 2000.
10. S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup. *Genetic Programming: Proceedings of the 3rd Annual Conference*, 1998.
11. J. Orkin. Three states and a plan: the AI of FEAR. In *Proceedings of the Game Developers Conference*, 2006.
12. S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels. Evolution of human-competitive agents in modern computer games. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.
13. J. Togelius, R. De Nardi, and S. Lucas. Towards automatic personalised content creation for racing games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.