A Strongly Typed GP-based Video Game Player

Baozhu Jia
Ernst-Moritz-Arndt Universität Greifswald
Institut für Mathematik und Informatik
Walther-Rathenau-Strae 47, 17487 Greifswald, Germany
Email: baozhuj@uni-greifswald.de

Marc Ebner
Ernst-Moritz-Arndt Universität Greifswald
Institut für Mathematik und Informatik
Walther-Rathenau-Strae 47, 17487 Greifswald, Germany
Email: marc.ebner@uni-greifswald.de

***Abstract*—This paper attempts to evolve a general video game player, i.e. an agent which is able to learn to play many different video games with little domain knowledge. Our project uses strongly typed genetic programming as a learning algorithm. Three simple hand-crafted features are chosen to represent the game state. Each feature is a vector which consists of the position and orientation of each game object that is visible on the screen. These feature vectors are handed to the learning algorithm which will output the action the game player will take next. Game knowledge and feature vectors are acquired by processing screen grabs from the game. Three different video games are used to test the algorithm. Experiments show that our algorithm is able to find solutions to play all these three games efficiently.**

I. Introduction

As a challenging problem and simulation environment, video games have been used for many years in artificial intelligence. They have some properties which make them an ideal tool for artificial intelligence research. A lot of progress has been made in creating artificial game playing agents which are able to play certain specific games such as chess [1], Go [2], backgammon [3] and even video games [4]. Most game agents are based on game knowledge and are usually limited to a specific game. Recently, more and more researchers are interested in developing a general artificial game player which is capable of playing many different video games without game knowledge [5], [6].

In this paper, we present a game player which is based on genetic programming [7]–[9]. It uses a population of programs or individuals. Each individual is able to compute the next action of the game player based on current information about the game obtained from screen grabs. The population of computer programs is evolved using strongly typed genetic programming [10]. Whenever a new video game is encountered, this game player will have to play it a certain number of times. Using feedback from this game play, the population of programs adapts to the current game. Once a reasonably good solution is found, it can be stored in an archive once this game is encountered again at a later time. Three hand-crafted game features are extracted from the screen grabs and conveyed to the learning algorithm as input.

We test our general game player on three different games: Space Invaders, Frogger and Missile Command. For our experiments we use the Py-vgdl game engine, which is based on the video game description language (VGDL) [11]. The games are similar to old Atari 2600 games which have been played by many players all over the world. The complexity of these games lies in between classic board games and modern 3D games.

Section II briefly summarises previous works on general game players. Section III describes how we compute feature vectors from screen grabs. Section IV presents three different representations of the game state. Section V demonstrates how the game player is evolved using strongly typed genetic programming. The conclusions are given in the final section.

II. Related work

General game playing dates back to the AAAI general game playing competition in 2005 [12]. This competition focuses on turn-taking board games. The game rules are not known to the players in advance. Each game is described in a logical game description language. The players will get the game state information from a game manager once the game begins. The game manager is used to track the game state and sends game information to the players, e.g. how opponents move. Successful players mainly use Monte Carlo Tree Search [13]–[15].

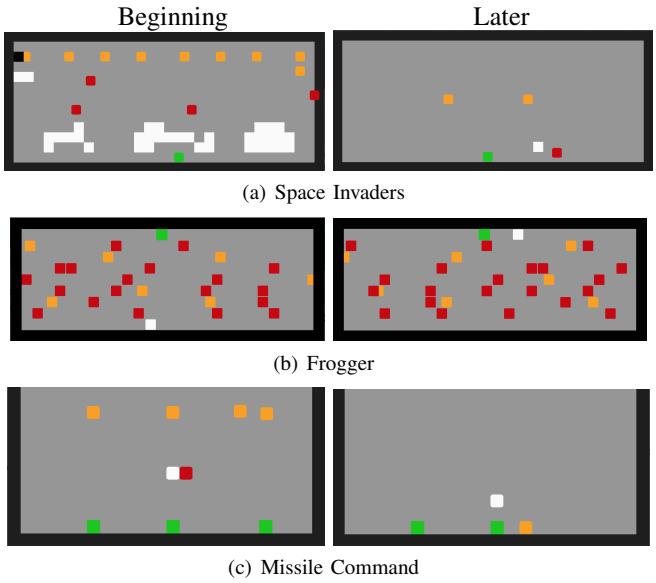
The Atari 2600 games group developed an Arcade Learning Environment [16] to explore how computers learn to play games through experience. Naddaf [17] presented two model-free algorithms: reinforcement learning (Sarsa(λ)) and Monte Carlo tree search in his master's thesis. Game features were generated from game screen grabs as well as from console RAM. Hausknecht et al. [18] have presented a HyperNEAT-based Atari general game player. HyperNEAT is said to be able to exploit geometric regularities present in the game screen in order to evolve highly effective game playing policies. The game screen is represented as a grid which will be conveyed to the learning algorithm as input. The avatar is detected using information gain which is obtained when moving the game controller and watching the objects move on screen. The same approach to detect the avatar is also used in this paper. In the paper [19], they improved upon previous work by adding an output layer to the architecture of the learning algorithm and were using three new game features.

Mnih et al. [5], [6] combined deep learning with reinforcement learning to learn the game policies from raw image pixels. Their method was able to create game playing agents which are comparable to human players. Guo [20] presented another approach which consisted of deep learning and Monte Carlo tree search methods. This approach retained the advantage of learning policies from raw image pixels and improved the performance of the previous paper. These two

works represent the state-of-the-art technology in the area of learning policies by observing screen pixels without using hand-crafted features.

The general video game AI competition has been held in 2014 [21]. The GVG-AI Competition explores the problem of creating controllers for general video game playing. The game state information is gained from an emulator rather than from screen grabs. Perez et al. [22] presented a knowledge-based Fast Evolutionary Monte Carlo tree search method (Fast-Evo-MCTS), where the reward function adapts to the knowledge change and distance change when no game score is gained. Game state information is obtained from the emulator. Hence, the environment is fully observable. Their results also show that the three Monte Carlo tree search methods are superior the previous methods. Here, we will also compare our approach with these three MCTS approaches.

Another GP-based game player is presented by Jia et al. [23]. Three decision trees are evolved to determine the agent's behaviours. Each tree is comprised of arithmetic functions and terminals. This paper attempts to explore more complex functions, such as logistic functions, to evolve the game playing policies.



(a) Space Invaders

(b) Frogger

(c) Missile Command

Fig. 1: For each game, two snapshots are shown. One from the beginning of the game and one well into the game.

III. Image processing and self detection

For our method, it is important to know which object on the screen corresponds to the game player. This information is obtained by playing the game and observing how the screen changes as the controls are manipulated. Apart from finding out which game object corresponds to the game player's self, one needs to find out what the game is about, i.e. what goal needs to be achieved. All the game state information is obtained from screen grabs. Figure 1 illustrates the games which we have used for our experiments. We assume that each object class has one unique colour which can be used to identify this object. This can eventually be generalised to more complex visual information such as texture.

The method used to determine the position of the avatar and of other game objects on the screen is described by Jia et al. [23]. Since each object can be identified by a unique color, we simply transform the RGB screen grab to a gray scale image. Each value in the transformed image corresponds to a unique hue. The gray image is downsampled to 1/16th of its original size to reduce the computational complexity. Next, a 9×9 neighbourhood non-local maximum suppression algorithm is used on the down-sampled image to locate individual objects of the game. The whole process is illustrated in Figure 2.

A. Avatar Identification

The avatar is the entity that can be controlled by the actions of game player. Its movement depends on the actions of game player. To locate the avatar on the screen, we need to find out which object is affected most by the actions of the game player. This can be done by computing the information gain after a sequence of moves, i.e 100 frames. This method was first used by Hausknecht [18].

In order to compute information gain for each object, we need to get every object's velocity list. As discussed above, we do visual processing for each frame, extract and save feature points in an arraylist. Each feature point is considered as one object. After one action is executed by game player, objects in the screen may move a little distance. In order to calculate every object's velocity, we need to use two arraylists, where one is used to save the position of feature points in the previous frame, the other is used to save their current position. We then match the elements between these two lists, which should have the same object class and nearest distance in a local neighbourhood. If such a match is found, we assume the two elements which are from two successive frames correspond to the same object. The displacement between these two points is the velocity of this object. The position of this object in the old arraylist needs to be updated using its current position every time. The velocity is a two dimensional vector (v_x,v_y). For the sake of computational simplicity, we map them to one dimension integer value v and save it in a velocity list. The mapping approach is shown in Equation 1. When v_x changes, the first part of this equation varies between -3 and 3 (7 numbers), so there is a "7" in the second part. mod(x,y) is a modulo function. The final velocity is mapped to an integer in the range of [-24, 24].

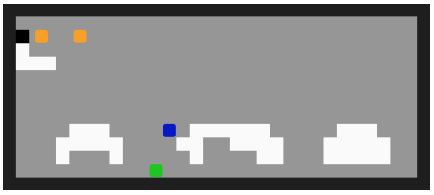
$$\begin{aligned} v = & \, mod(max(min(v_x, 19), -19), 5) \\ & + mod(max(min(v_y, 19), -19), 5) * 7 \end{aligned} \tag{1}$$

In this paper, a normalised information gain is used, which is given by:

$$I(o) = (H(V_o) \sum_{a \in A} (P_a H(V_o|a))) / H(V_o). \tag{2}$$

$H(V_o)$ is the entropy of the velocity and $H(V_o|a)$ is the selective entropy, where the action a is fixed. P_a is the possibility that action a appears.

The object o with the highest normalised information gain, i.e. the object which always moves consistently in the same



(a) RGB image

(b) Gray image

(c) Downsampled image

(d) Feature points

(e) Objects

Fig. 2: Image Processing for Space Invaders. (a) is the original input RGB image. (b) shows the gray image from (a). (c) is obtained by downsampling image (b). (d) shows the feature points after using non-local maximum suppression. In figure (e), shows the feature points on objects.

direction, whenever a certain control is pressed by the player, is considered to be the avatar of the player.

The identification of avatar only needs to be invoked once during the learning process. Once we have identified the avatar, its colour value will be stored in a text file.

IV. Game State Features

We assume that each game has a score count and that the goal of each game is to achieve a score that is as high as possible. It is conceivably that for some games a low score is more desirable. However, this case can be treated in a similar way. Our game playing agent takes the state parameters of the game as input and outputs the controls activated by the player. An obvious question is how many game parameters should be made available to our game agents. A sparse representation that nevertheless provides sufficient information to play the game seems to be desirable.

Since we have used genetic programming, we need to define the representation of our individuals, i.e. we need to decide which set of elementary functions and which set of terminal symbols to use. One possibility would be to provide the game agent with the complete game information, i.e. each individual pixel on the screen. Working with some form of preprocessed input makes probably more sense. As described above, we extract moving objects where each one is classified by its colour. Humans probably focus at any given time on a small number of objects. We only supply the game agent with the information on the location of the one nearest object for each class. In other words, the game agent is only able to see the nearest goal position, the nearest shot fired or the enemy which is nearest to the game agent. As we will see later on, this information is sufficient to play the games used for our experiments. Only five different object classes are considered. In the case that a game only has less than five different object classes which can be detected, then the return value of the corresponding terminal symbol is set to 10000.

We experiment with three different representations for the terminal symbols. Each representation provides information about the object which is nearest to the avatar for each class. However, the way this information is made available differs for each representation. Representation A (shown in Table I) contains only the x and y coordinates of the nearest object for each class relative to the avatar and also the Euclidean distance between the object and the player's avatar. Representation B (shown in Table II) contains the x and y coordinates of the nearest object for each class relative to avatar, the Euclidean distance between the object and the player's avatar, and also the angle of the vector pointing from the avatar to the nearest object. Table III contains only the Euclidean distance between the object and the player's avatar and the angle of the vector pointing from the avatar to the nearest object. X and y coordinates are not made available in Representation C. This representation may be especially useful for some games, where the agent needs to rotate and move forward/backward to shoot and avoid enemies. How object information is made available through the terminal symbols for the game Space Invaders is illustrated in Figure 3.

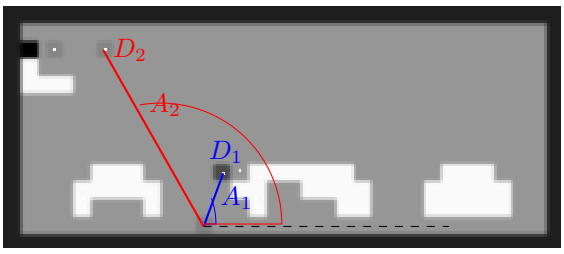


Fig. 3: Object 1 is a bomb. Object 2 is an alien. Angle(A_i) and Euclidean distance(D_i) are demonstrated on this map.

TABLE I: Representation A: The coordinate and euclidean distance of nearest object for each class relative to avatar.

Terminal Symbol	Description
X_i	x coordinate of the nearest object for class i relative to avatar
Y_i	y coordinate of the nearest object for class i relative to avatar
D_i	Euclidean distance between the avatar and the nearest object for class i

TABLE II: Representation B: Angle of vector pointing from the avatar to the nearest object is also included.

Terminal Symbol	Description
X_i	x coordinate of the nearest object for class i relative to avatar
Y_i	y coordinate of the nearest object for class i relative to avatar
D_i	Euclidean distance between the avatar and the nearest object of class i
A_i	Angle of vector pointing from the avatar to the nearest object of class i

V. Evolving a game playing agent using strongly typed genetic programming

Genetic programming starts from a high-level statement of the problem and does not need problem knowledge. The population of programs is progressively evolved towards the optimum solution with high fitness. In this paper, the game knowledge is unknown and only game snapshot and score points can be acquired from the game engine. So GP is used to search for the playing strategies for the games.

Table IV shows the set of terminals used for all our experiments. This basic set of terminals is extended by the terminals from Table I, II, III for Representations A, B, and C. Actions are selected by the terminals `Left`, `Right`, `Up`, `Down`, `Null` and `Button`. The first four terminals determine the moving direction of the avatar.

The set of elementary functions is shown in Table V. We use only three different functions. Two functions (`greaterThan`, `lessThan`) are used to compare game state

TABLE III: Representation C: Euclidean distance between nearest object for each class and the avatar and the angle of vector pointing from the avatar to the nearest object.

Terminal Symbol	Description
D_i	Euclidean distance between the avatar and the nearest object of class i
A_i	Angle of vector pointing from the avatar to the nearest object of class i

TABLE IV: Terminal symbols.

Terminal Symbol	Description
`ERC`	ephemeral random constant from the range [-400, 400]
`Left`	action: move left
`Right`	action: move right
`Down`	action: move down
`Up`	action: move up
`Button`	action: button press
`Null`	no action

TABLE VI: Parameters for Genetic Programming.

Parameter	Value
population size	200
crossover probability	0.4
reproduction probability	0.4
mutation probability	0.1
ERC mutation probability	0.1
mutate.tries	2
builder	HalfBuilder
tournament size	3
maximum depth	10

information with the ERC value. ERC values are changed using normal distributed random numbers. The terminals from game state features are assigned to the type integer. The ERC terminals are assigned the type float. The nodes in GP tree are allowed to connect as parent and child if their corresponding type objects are type compatible. The `if` function can be used to select which command will be executed in next step. The parameters that we have used for our experiments are shown in Table VI. We have used tournament selection to select individuals.

A. Experiments and Evaluation

For our experiments, we have used the game engine developed by Tom Schaul. Sean Luke's ECJ package [24] has been used to implement the strongly typed genetic programming learning algorithm. The game engine and the learning algorithm are two separate programs. They communicate with each other through TCP/IP. First, the game emulator is started. If the player's self has not yet been identified, it will be identified using the method for self localisation as described above. Once the self has been identified, each individual will be evaluated in turn. Whenever a new individual has been evaluated, the emulator is informed that a new game needs to be started in order to evaluate the next individual. The game state information calculated from the emulator is conveyed to the learning algorithm via GP terminal symbols. Based on this information the decision tree will compute the next action. Once this action has been chosen, it will be sent to the emulator. The game emulator then executes this action. This process continues until the game ends. As soon as the game reaches the end, the resulting score from the emulator will be sent to the learning algorithm. The average score obtained for these runs is used as the fitness of the individual.

Our general game player is evaluated on three different games: Space Invaders, Frogger and Missile Command. Screen grabs from these games are shown in Figure 1. Space Invaders is a classic arcade game. The goal is to protect a planet using a small gun shooting at an alien invasion coming in from above. Frogger is a classic game where a small frog needs to cross a road. The player essentially needs to move from the bottom of the screen to a goal position located at the top of the screen. While doing that, the player needs to watch out for cars as he is crossing the road. In Missile Command, the player needs to use smart bombs to destroy incoming ballistic missiles. The smart bomb's moving direction is consistent with avatar's moving direction. A smart bomb will destroy all incoming missiles within a certain radius.

TABLE V: Set of elementary functions.

Function	Description
`bool greaterThan(integer arg1,float arg2)`	if `arg1` is greater than `arg2` return true, otherwise return false
`bool lessThan(integer arg1, float arg2)`	if `arg1` is less than `arg2` return true, otherwise return false
`void if(bool arg1,void arg2,void arg3)`	if arg1 is true, command arg2 will be executed, otherwise arg3 will be executed

It is clear that the game engine only recreates the original game idea but does not use the graphics from the original games. Different objects can easily be identified through their colour. Instead of using only the object's colour to identify it, we could use texture and other more complex features for more realistic graphics.

Figure 4 shows the fitness of the best individual for each generation averaged over 10 runs. For Space Invaders and Missile Command it will take usually 100 generations to find an optimal playing strategy. There are 200 individuals which need to be evaluated in each generation. For each individual, we run the game 3 times with a different random seed and game environment. This sums up to 60000 game plays for these two games in one run. For Frogger, the generation is set to 150. The number of individuals is 200. Each individual is tested twice different random seeds and game environment. Hence, we also have 60000 game plays for Frogger in each run. Since each game play will take 10 seconds on average, it will take nearly 2 days for a single run. Due to the large number of game plays required, we only use 10 runs for each experiment.

Table VII shows the average best fitness obtained over 100 generations for Space Invaders and Missile Command and 150 generations for Frogger. The standard deviation is also shown. We use a Mann-Whitney U-test to compare these three representations. The uncertainty that representation B performs better than C is in the range [20% 30%]. The uncertainty that representation B performs better than A is in the range [10% 20%]. That means that there is no significance difference among these three representations for these three games.

We also compare our algorithm with three Monte Carlo Tree Search algorithms: Vanila MCTS, Fast-Evo MCTS and KB-Evo MCTS [22]. MCTS approximates the value of actions by random sampling and simulation iteratively. These values are used to adjust the decision policy. The MCTS agent accesses the full game state information from the game engine directly, and it can also advance the game state to a maximum depth. The research of Guo et al. [20] also reveals that the performance of Monte Carlo Tree Search is superior to other methods. Fast-Evo MCTS embedded the roll-outs within evolution, which can adapt to the number of game features. KB-Evo MCTS is a MCTS method that biases the roll-outs by using both knowledge base and evolution. The comparison results are also shown in table VII. Using the Mann-Whitney U-test, we find that our algorithm has slightly worse performance when playing Space Invaders, but it has a similar performance when compared to the three Monte Carlo Tree Search approaches when playing Frogger and Missile Command. This is quite remarkable since our algorithm uses screen grabs to compute game features whereas the other algorithms obtain the game state features directly from the game engine.

VI. Conclusion

Video games provide a challenging environment for research in artificial intelligence. In this paper, we have presented a strongly-typed genetic programming game player. Only screen grabs are used to compute input features. The score that the player obtained when playing the game was used to compute the fitness of each individual. The score was extracted from the game engine. Three simple genetic programming representations were evaluated. The results show that there are no significant difference among them when testing on the three given games. Our evolved game player performed similarly to Monte Carlo Tree Search algorithms. Even though we only used three games for our experiments, this provides a small step forward in creating a general genetic programming video game player using only screen grabs as input.

References

[1] M.Campbell, A. H. Jr., and F.-H. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[2] S. Gelly and D.Silver, "Monte Carlo Tree Search and Rapid Action Value Estimation in Computer Go," *Artificial Intelligence*, vol. 175, pp. 1856–1875, July 2011.

[3] G. Tesauro, "Td-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Computation*, vol. 6, pp. 215–219, March 1994.

[4] A.M.Alhejali and S. M. Lucas, "Evolving diverse Ms. Pac-Man playing agents using genetic programming," in *Workshop on Computer Intelligence*, 2010, pp. 1–6.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M.Riedmiller, "Playing with Deep Reinforcement Learning," in *Neural Information Processing Systems Workshop*, 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidieland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, and D. Kumaran, "Human-level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, pp. 529–533, 2015.

[7] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[8] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs* . The MIT Press, 1994.

[9] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming - An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. dpunkt-Verlag and Morgan Kaufmann, 1998.

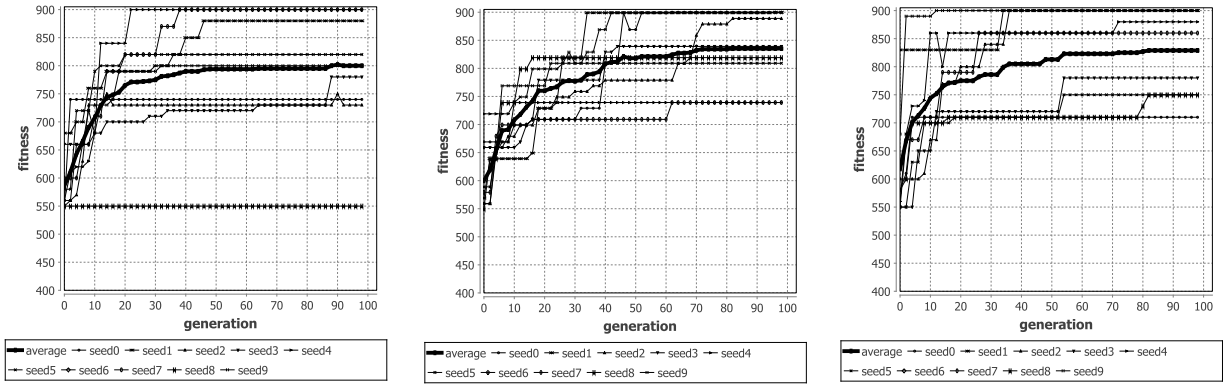
[10] D. Montana, "Strongly Typed Genetic Programming," *Evolutionary Computation*, vol. 3, pp. 199–230, 1995.

[11] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, " General Video Game Playing," in *Dagstuhl Follow-Ups*, vol. 6, 2013, pp. 77–83.

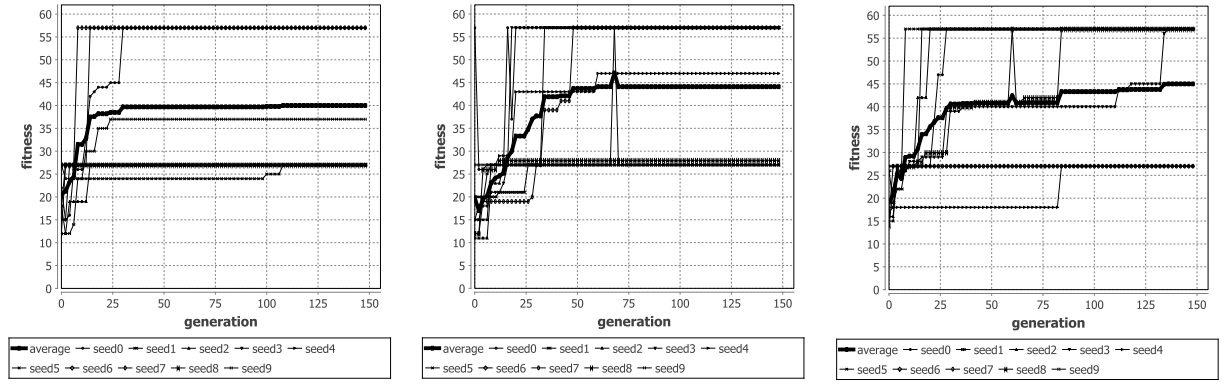
[12] M. Geneserech and N. Love, "General Game Playing: Overview of the AAAI Competition," *AI Magazine*, vol. 26, pp. 62–72, 2005.

[13] J. Mehat and T. Cazenave, "Monte-Carlo Tree Search for General Game Playing," LIASD, Dept. Informatique, Université Paris 8, Tech. Rep., 2008.

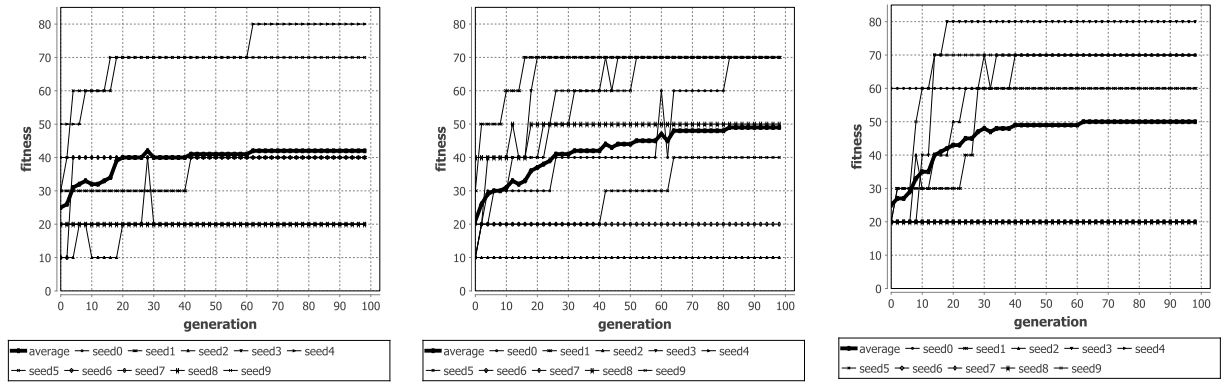
[14] ——, "Combining UCT and Nested Monte-Carlo Search for Single-Player General Game Playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2(4), pp. 225–228, 2010.



(a) Fitness of Space Invaders implementing Representation A (b) Fitness of Space Invaders implementing Representation B (c) Fitness of Space Invaders implementing Representation C



(d) Fitness of Frogger implementing Representation A (e) Fitness of Frogger implementing Representation B (f) Fitness of Frogger implementing Representation C



(g) Fitness of Missile Command implementing Representation A (h) Fitness of Missile Command implementing Representation B (i) Fitness of Missile Command implementing Representation C

Fig. 4: Fitness of the best individual over 10 different runs for each game. The average fitness for each game is shown in the bold line.

TABLE VII: Average scores over 10 runs obtained from the games: Space Invaders, Frogger, Missile Command. The maximum possible scores, Space Invaders:900, Frogger:57 and Missile Command:80.

Games	Game state from screen grabs			Game state from game engine		
	GP+Repr.A	GP+ Repr.B	GP+Repr.C	Vanila MCTS	Fast-Evo MCTS	KB Fast-Evo MCTS
Space Invaders	802 ±108.4	836±62.21	824±71.37	**900 ±0**	**900 ±0**	858±132.82
Frogger	40± 14.94	**47.1±13.99**	45±15.49	24.2±4.7563	25±3.16	37±10.59
Missile Command	44± 22.91	50±25.82	50±26.24	39±8.756	47.27±11.9	**59±3.17**

[15] H. Finnsson and Y. Bjornsson, "Simulation-Based Approach to General Game Playing," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008, pp. 259–264.

[16] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research*, pp. 253–279, 2012.

[17] Y. Naddaf, "Game-Independent AI Agents for Playing Atari 2600 Console Games," Master's thesis, University of Alberta, 2010.

[18] M. Hausknecht, P. Khandelwal, R.Miikkulainen, and P. Stone, "HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player," in *Genetic and Evolutionary Computation Conference(GECCO)*, 2012.

[19] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A Neuroevolution Approach to General Atrai Game Playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, pp. 355–366, 2013.

[20] X. Guo, S. Singh, H. Lee, R. Lewis, and X. Wang, "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning," in *Advances in Neural Information Processing Systems 27*, 2014, pp. 3338–3346.

[21] D.Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas, "GVG-AI Competition," http://www.gvgai.net/index.php.

[22] D. Perez, S. Samothrakis, and S. Lucas, "Knowledge-based Fast Evolutionary MCTS for General Video Game Playing," in *Proceedings of IEEE Conference on Computational Intelligence and Games*, 2014, pp. 68–75.

[23] C. S. B. Jia, M. Ebner, "A GP-based Video Game Player," in *Genetic and Evolutionary Computation Conference(GECCO)*, 2015.

[24] Luke, *The ECJ Owner's Manual*, 22nd ed., 2014.