A GP-based Video Game Player

Baozhu Jia, Marc Ebner, Christian Schack

(baozhuj,marc.ebner,christian.schack)@uni-greifswald.de

Ernst Moritz Arndt Universität Greifswald, Institut für Mathematik und Informatik,

Walther-Rathenau-Straße 47, 17487 Greifswald, Germany

ABSTRACT

A general video game player is an an agent that can learn to play different video games with no specific domain knowledge. We are working towards developing a GP-based general video game player. Our system currently extracts game state features from screen grabs. This information is then passed on to the game player. Fitness is computed from data obtained directly from the internals of the game simulator. For this paper, we compare three different types of game state features. These features differ in how they describe the position to the nearest object surrounding the player. We have tested our genetic programming game player system on three games: Space Invaders, Frogger and Missile Command. Our results show that a playing strategy for each game can be found efficiently for all three representations.

Keywords

Genetic Programming, General Video Game Player, Game State Features

1. INTRODUCTION

The idea of general video game playing is to create a video game player that is able to learn to play different video games without domain-specific knowledge. When encountering a new game, the game player first needs to learn where the avatar is located, what is the goal of the game and what is the current game state. Most game state information can be obtained by image processing frame grabs of the game. Using this information, the game player is able to search for a good playing strategy by playing the game several times.

Playing video games is a challenging problem even for human game players. Each video game has a certain goal. Most video games give feedback to the player via a score which describes to what extent the player has achieved the this goal. Usually, we have one or more game agents that can be manipulated by the game player. For each game, a sequence of actions has to be found that for any given game state maximises the score that will eventually be achieved. Hence, video games are excellent testbed for research in artificial intelligence or machine learning approaches.

Game state is extracted from frame grabs and conveyed to the learning algorithm. For this paper, we have used genetic programming [11, 12, 2] to evolve game playing strategies. The strategies are tested on three different video games: Space Invaders, Frogger and Missile Command. These games were implemented using the Python video game description language game engine developed by Tom Schaul [13]. The games are similar to old Atari 2600 games.

Three trees are used to determine the action of the player. Each tree computes a scalar value based on the current game state. The output of the first tree determines joystick movement in the horizontal direction, the second joystick movement in the vertical direction and the third tree determines whether a button is pressed or not. We have used ECJ [14] to evolve our individuals.

Section 2 summarises previous work on general video game playing. Section 3 introduces the image processing methods and algorithm for self detection which was implemented in this work. Section 4 describes the game state feature representations and also demonstrates how the game player is evolved through genetic programming. The result are discussed in section 5. The conclusion is given in the final section.

2. PREVIOUS WORK

Artificial intelligence research has achieved great progress in developing players for certain specific games, such as chess [15], Go [6] or Pac Man [1]. These game players are dedicated to playing their respective game but cannot be used to play another game.

The relatively new research area of general game playing focuses on developing general game players that are capable of playing arbitrary games. A competition on general game playing is annually held since the AAAI 2005 conference [7]. This competition, however, focuses on turn-taking board games. For general game playing, it is difficult to design an algorithm that can cope with a variety of games. Most successful participants of the general game playing competition use Monte Carlo Tree Search as their controlling algorithms [16, 17, 5].

The general video game AI Competition has been held since 2014 [4]. This competition aims to create a controller which can learn to play a variety of video games, without previously knowing which games are to be played. In this competition, there is no need to analyse screen grabs, be-

GECCO '15, July 11 - 15, 2015, Madrid, Spain

 ISBN 978-1-4503-3472-3/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739480.2754735

cause all game state information is accessible via encapsulated objects. Perez et al [21] describe a knowledge-based fast evolutionary Monte Carlo tree search approach for General Video Game Playing. A new score function is provided in this paper based on score change, knowledge change and distance change.

Naddaf [20] presented two approaches to play Atari 2600 console games without having any game-specific prior knowledge. The agent must analyse the screen grab to identify the game avatar as well as game state information. He used two methods to learn the game strategy: reinforcement learning based methods and search based methods. The reinforcement learning based methods use feature vectors extracted from the game screen to learn to play a given game. The search-based methods use the emulator to simulate the consequence of actions into the future, aiming to play as well as possible by only exploring a very small fraction of the state-space. Hausknecht et al. [9, 10] presented a HyperNEAT-based general game player. In this framework, a high level of game state representation can be extracted from game screen. It is said that such a method is capable of exploiting geometric regularities which is very important in game playing.

Mnih et al. [18, 19] first try to learn the vision features and control policies at the same time. A deep learning model, called DQN which had great achievement in playing some Atari games, has been presented. The model combines convolutional neural network and Q-Learning, which takes raw pixels as input and outputs a value function estimating future rewards.

Guo et al. [8] described another method which combines deep learning (DL) and MCTS planing. They collect training data from a UCT agent, where UCT is one of MCTS methods. Using these data, three convolutional neural network (CNN) are trained. They attempt to retain the Deep Learning advantage of not needing hand-crafted features and the online real-time play ability of the model-free Reinforcement Learning(RL) agents. They evaluated their model on seven Atari games and found that their model outperforms previous DQN models.

3. IMAGE PROCESSING AND SELF DETECTION

Before we are able to play a game, we need to acquire some knowledge about the game, i.e. Who am I? Where am I? What is the current game state and what is the game about? We only use screen grabs to find answers to these questions.

3.1 Visual Processing

Each game has several classes of objects. These objects may belong to the game agent, a goal position, enemies or food and so on. We need to locate these objects and determine the relationship between them. The objects can be identified using distinguishing features. The objects of the games used in this paper can be identified easily by their colour. Each object has a unique colour. Hence, we first compute a colour histogram. To do this, the image, originally an RGB image, is converted to a gray scale image by mapping each hue to a particular gray scale. This allows us to detect objects rapidly. A histogram is computed from the gray scale image (shown in Figure 1 for the game Frogger). From the histogram it is clear that there are six object classes. Each peak in this histogram corresponds to one object class.

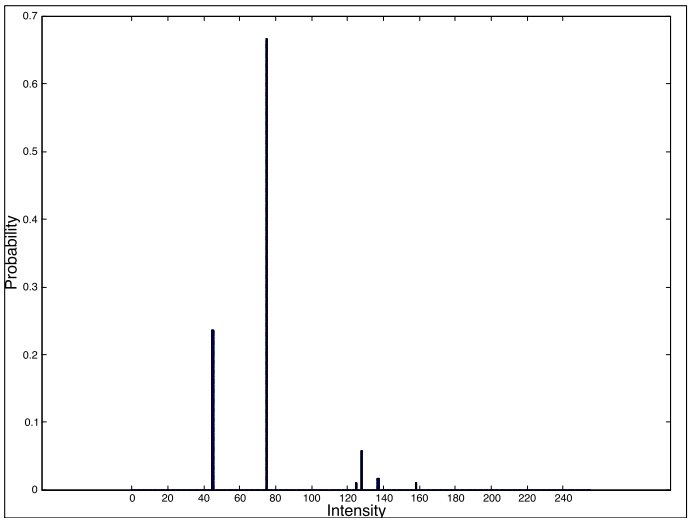
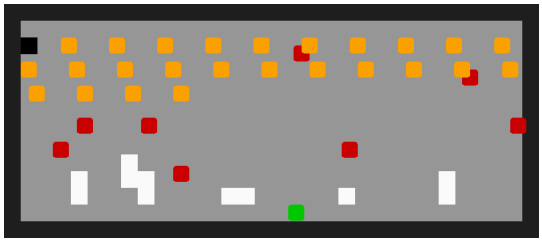


Figure 1: Histogram for Frogger.

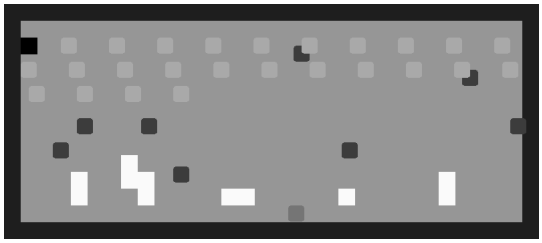
If each object is basically a blob of a unique colour, then its position can be determined easily by applying blur and then extracting local maxima. This would also work for games with more complex graphics by applying a convolution with a filter having the same graphics as the object to be extracted. To reduce the computation complexity, we first down sample the gray image using a blur filter to 1/16th of its original size. Next we apply a non-local maximum suppression filter on the down-sampled image. Each object in the game is then represented by one point. The position of these points are saved in a **feature list**. The list of features is used to detect the game agent and to acquire game state information. This will be explained in detail in the following sections. Figure 2 illustrates the image processing pipeline.

3.2 Avatar Identification

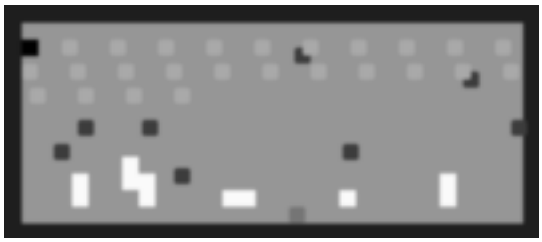
Avatar identification is another key task that has to be solved before the game can be played. We modify Hausknecht et al.'s method [9] to identify the avatar. The avatar is the entity, whose movement is affected most by actions of the game player. The movement of an object is represented by its velocity in the x direction v_x and its velocity in the y direction v_y. Two array lists are used to calculate the velocities of the objects. One array list List_c is used to record the position and the colour of feature points from the current frame. The second list List_p is used to record the position and the colour of feature points from the previous frame. For each element $a \in \text{List}_p$, we search List_c for objects which are located in a small neighbourhood around a to find the nearest object b which belongs to the same class as a. This object is assumed to be the same object as a. The difference between these two positions of objects b and a is the velocity (v_x, v_y) of object a. The velocity vector is mapped to a one-dimensional velocity with the range [-24, 24], as shown in Equation 1, where mod(x, y) is the modulo operation.



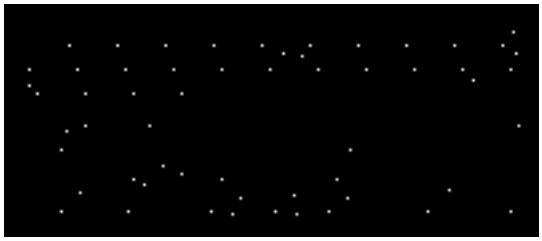
(a) RGB image



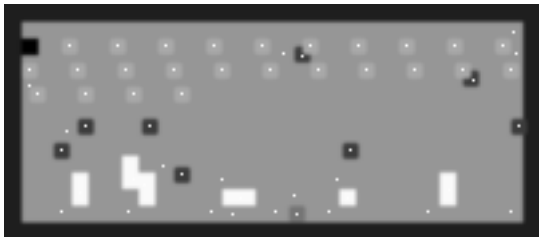
(b) Gray image



(c) Downsampled image



(d) Feature points



(e) Objects

Figure 2: Image processing for Space Invaders. (a) input RGB image. (b) gray scale image. (c) downsampled image. (d) feature points after using non-local maximum suppression. (e) feature points overlaid on RGB image.

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 avatar and the nearest object for class i
A_i	Angle between vector pointing from self to the nearest object and the horizontal axis.

Table 1: Terminal symbols

Because we want to get an integer velocity here.

$$v = mod(max(min(v_x, 19), -19), 5) + mod(max(min(v_y, 19), -19), 5) * 7 \quad (1)$$

Naturally, there may be situations where the nearest object is actually some other object of the same class. In these cases, the velocity computation fails.

Hausknecht [9] proposed a method to identify the avatar using information gain which is the difference between one object's velocity entropy and the weighted sum of its selective entropy. Entropy is calculated using the formulation: $H(V) = \sum_{i=1}^{n} p(v_i) * ln(p(v_i))$. It uses the entire velocity history. Selective entropy is the entropy calculated by only implementing the velocities corresponding to one fixed action. The avatar's motion is affected by the actions of the game player. Hence, it should have the largest information gain. But it fails in some games in our work, because one object's information gain may be even larger than avatar's entropy. Then this object will be mistaken as avatar. A normalised information gain is used in our work, which is calculated using formulation:

$$I_r = I/H(o) \quad (2)$$

I is the information gain and H(o) is the entropy. There is an assumption that the movement of avatar is fixed when the same actions are taken. If the games do not obey this rule, this algorithm fails.

3.3 Game State Features

A game player needs to know what the current game state is before making a decision. A game player will be able to make a better decision the more information is known about the game. It would be ideal if our game agent had access to the position of all objects. If we would provide our game agent with the position of all objects, it would probably take a long time for the game agent to learn what objects are most useful in making the next step. We use a quite simplistic approach. We only provide information about the nearest object of each class to the game agent. The nearest object may be most dangerous to avatar if it is an enemy. It can also be the food the avatar is searching for.

The nearest object is the object which has the smallest Euclidean distance to the avatar. This feature is easy to compute and quite useful.

In this work, we present three representations for the game state to find solutions to each of these three games: Space Invaders, Frogger and Missile Command. They differ in the

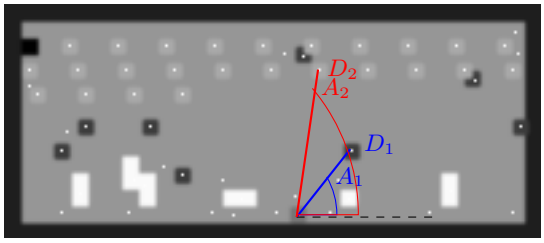


Figure 3: Computation of the angle and the Euclidean distance to the nearest object.

Function	Description
Add(arg1, arg2)	return arg1+arg2
Subtract(arg1,arg2)	return arg1-arg2
Divide(arg1,arg2)	return arg1/arg2 if arg2 != 0, otherwise return 100000
Multiply(arg1, arg2)	return arg1*arg2
Negate(arg1)	return -arg1
Sqrt(arg1)	return $\sqrt{arg1}$
Square(arg1)	return $arg1 * arg1$

Table 2: Set of elementary functions.

way how to represent the nearest object. The terminal symbols are shown in Table 1, where each terminal symbol is used for each $i \in \{1, ..., n\}$ where n is the number of different object classes. Representation A uses only terminal symbols X_i, Y_i, D_i. Representation B uses terminal symbols X_i, Y_i, D_i, A_i. Representation C uses only terminal symbols D_i, A_i. How the angle and the Euclidean distance to the nearest object is computed is shown in Figure 3 for a sample scene. The first representation is used to explore the importance of Cartesian coordinates. The third representation is used to explore the importance of Polar coordinates.

4. EVOLVING A GAME PLAYER USING GENETIC PROGRAMMING

We have used Sean Luke's ECJ System [14] to evolve our game agents. A modified version of Tom Schaul's py-vdgl game engine [13] was used to play the games. Both programs, the ECJ system which constructs the genetic programming individuals which need to be evaluated and the game engine are maintained as separate programs. They communicate with each other through the TCP/IP protocol. In each game step, ECJ system receives the game state feature vector from game engine, and then sends the calculated action back to it. The image processing program is carried out directly in the game engine.

The elementary functions that we use are shown in Table 2. This set consists of basic arithmetic functions. The terminal symbol which we have used for our representation have already been shown in Table 1.

Each individual consists of three trees. Figure 5 illustrates one example for the structure of each tree. The value of the first tree T_1 determines avatar's movement in the horizontal direction (left, right or no action). The value of the second tree T_2 determines the avatar's movement in vertical direction (up, down or no action). The value of the third

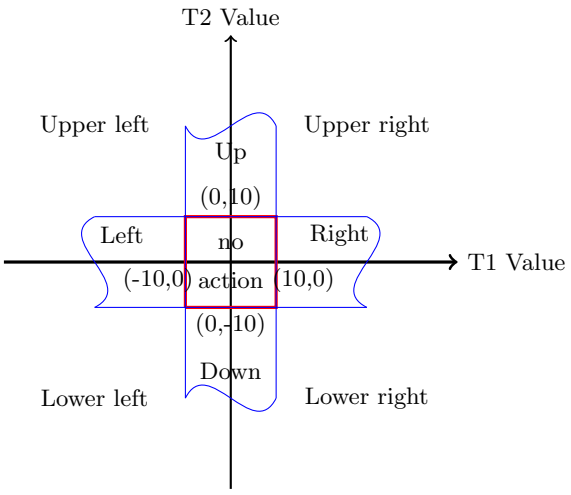


Figure 4: The moving direction of avatar depending on the value of Tree 1 and Tree 2.

Value of Tree 3	Actions
[0 , ∞)	Press_button
(-∞, 0)	No_action

Table 3: Whether the button will be pressed down depending on the value of Tree 3.

tree T_3 determines whether the button should be pressed or not. Figure 4 illustrates what motion will be chosen for the avatar depending on the values of the trees T_1 and T_2. Table 3 shows when the button will be pressed down depending on the value of T_3. One of a total of 18 different actions will be chosen depending on the value of these three trees.

5. EXPERIMENTS AND RESULTS

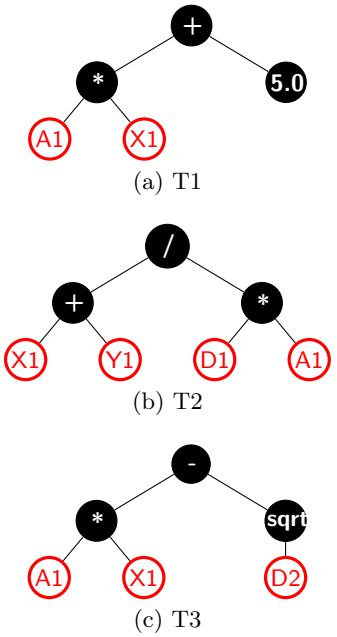
The parameters which we have used for our experiments are shown in Table 4. We have used a population of 200 individuals. Tournament selection is used to select individuals with a tournament size of 3. Evolution is carried out for 100 generations. As genetic operators we use crossover, mutation and reproduction. The probabilities with which these operators are applied are also shown in Table 4. Ephemeral random constants are mutated by adding a random value with a Gaussian distribution. The mutation operator tries to generate a subtree maximally twice before giving up if the mutated tree is not within the depth limit of 10. ECJ's halfBuilder is used to create individuals of the first generation.

In order to reduce noise, we evaluate each individual on several different game levels of a game. The levels differ in avatar's initial position and random seed which affects the behaviour of game sprites. The sum the game scores is used as the fitness of the individual.

Whenever an individual is evaluated, a new game is started. For each game step, the feature vector (with the terminal symbols as described above), is handed to the individual. The values for the three trees of this individual are calculated and then passed back to the game engine.

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

Table 4: Parameters for genetic programming.



(a) T1

(b) T2

(c) T3

Figure 5: One example of the tree structure.

We use three different games to evaluate the representations: Space Invaders, Frogger and Missile Command. Space invaders is a classic arcade game. The goal is to protect the earth from the aliens coming in from above. The player has to shoot at the aliens with missiles. The player also needs to avoid bombs from aliens. Frogger is a game where a frog has to cross several roads to reach its home. The frog needs to move from the bottom of the screen to the goal position which is located at the top of screen. While finishing this task, the frog needs to be careful with cars on the roads. The agent will get some bonus when it moves up one step or reaches its home. Missile Command is a game where the player's cities are attacked by ballistic bombs. The player should intercept these bombs using missiles. The missile's moving direction is consistent with avatar's moving direction. These three games as implemented in Py-vdgl are illustrated in Figure 6.

For each representation, we carry out 10 runs with different random seeds. It usually takes 5 seconds to play a single game. In order to evaluate one individual, we need to play this game three times, as described above. It takes approximately two days to complete a run of 100 generations. Hence, we only carry out 10 runs for each representation.

The fitness of the best individual for every generation in each run and the average fitness over 10 runs are shown in Figure 7. As the figure shows, the fitness improves considerably after 100 generations. In over 90% of the runs for both representation A and B genetic programming is able to find a perfect strategy for all three games. Table 5 shows the average best fitness and standard deviation after 100 generations for each game. The results show that both representation A as well as representation B work well in finding a program which will play the respective game.

We use the Mann-Whitney U-test to compare the three game state representations. Both Repres.A and B perform significantly better than Repres.C ($p < 0.01$).

We also compare our method with three Monte Carlo tree search methods: Vanila MCTS, Fast-Evo MCTS, KB Fast-Evo MCTS [21], as shown in Table 5. These three methods all the game state from the emulator. Vanila MCTS is a normal Monte Carlo tree search method that estimates the average reward by iteratively sampling the actions and building a search tree. The game agent will take the action that is visited most often or provides the highest average reward. Fast-Evo MCTS embedded the roll-outs within evolution, dynamically adapting to the game features. KB Fast-Evo MCTS used both knowledge base and evolution to bias the roll-outs. For more details about these three algorithms, the reader is referred to [21]. Their results show that MCTS method outperforms previous approaches.

For Space Invaders, our genetic programming players perform slightly worse than MCTS and Fast-Evo MCTS. But for Frogger and Missile Command, our genetic programming players show a much better performance than the three MCTS-related methods. The latter three algorithms need game domain information. The game state information are obtained from the game simulator rather than from the screen grabs as presented in this paper. They also need to advance the game state [3] before making a decision. But for our method, once the same game is encountered again, the evolved playing strategy can be used directly.

In this paper, we only test our algorithm on three games because of the limitation of evolving time. In the future plan, we will add more games into our work.

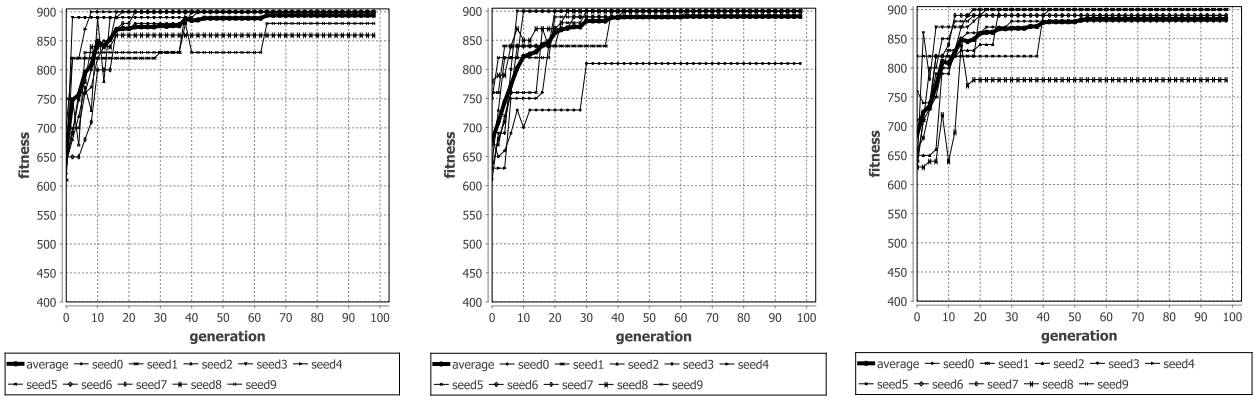
6. CONCLUSION

In this paper, we present three simple representations for the game state. The results show that both Repres.A and B work better than Repres.C for these three games. But we lack evidence that the same results among these three representations will be obtained for other games. Genetic programming is used as the learning algorithm in our work. GP players achieved better performance than MCTS methods in finding the playing strategy for these three video games. The genetic programming method represents a first step towards a general video game player.

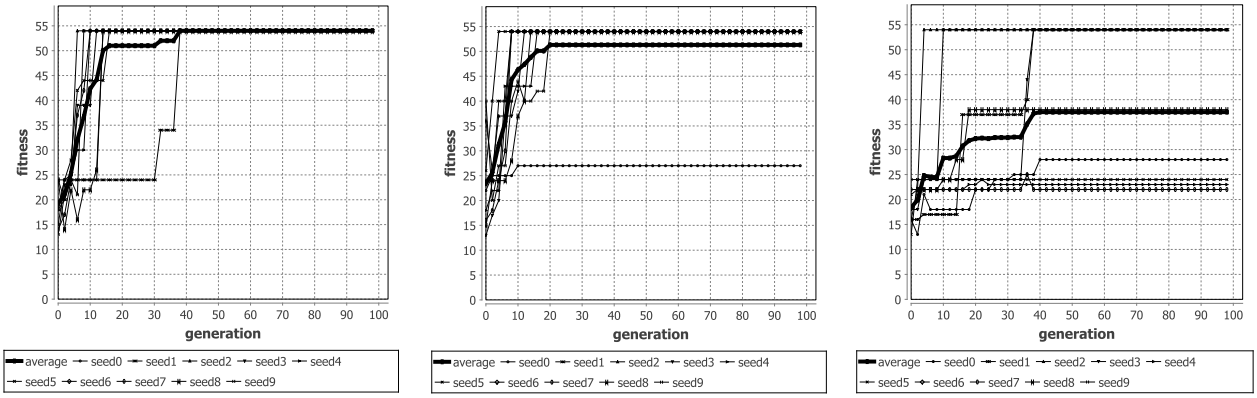
7. REFERENCES

[1] A.M.Alhejali and S. M. Lucas. Evolving diverse ms. pac-man playing agents using genetic programming. In *Workshop on Computer Intelligence*, pages 1–6, 2010.

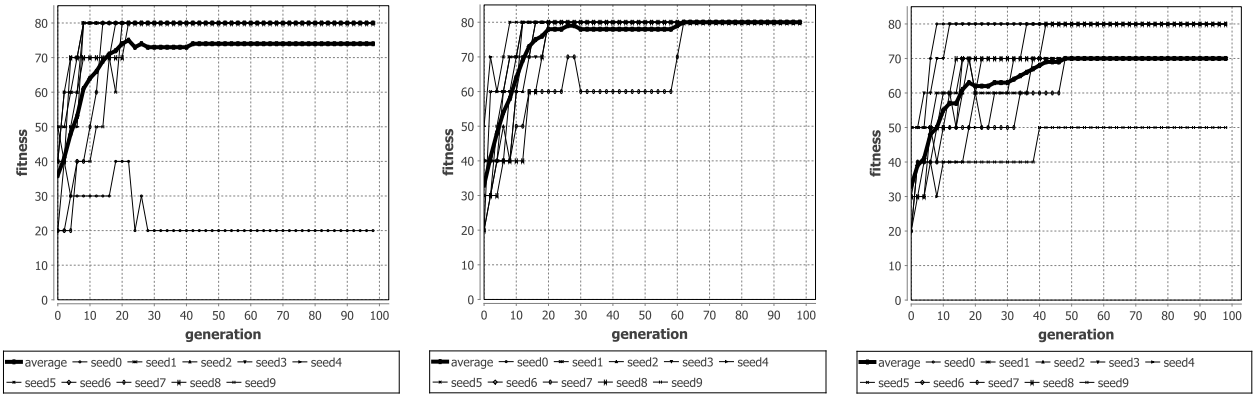
[2] W Banzhaf, P Nordin, RE Keller, and FD Francone. *Genetic Programming - An Introduction: On the*



(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

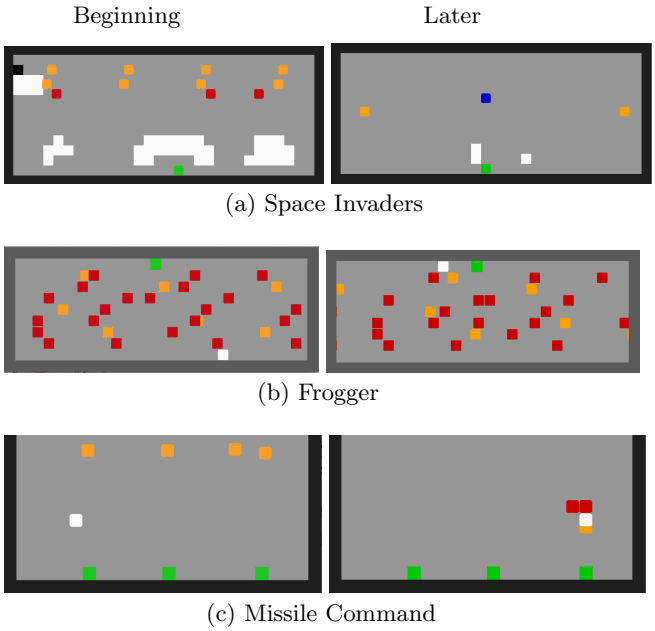


(g) Fitness of Missile Command implementing Representation A (h) Fitness of Missile Command implementing Representation B (i) Fitness of Missile Command implementing Representation C

Figure 7: Fitness of the best individual over 10 different runs for each game. The average fitness for each game is shown in the bold line.

	Average Score					
Game	GP+Repr.A	GP+Repr.B	GP+Repr.C	Vanila MCTS	Fast-Evo MCTS	KB Fast-Evo MCTS
Space Invaders	894 ±13.50	891±28.46	884±7.18	900 ±0	900 ±0	858±132.82
Frogger	54 ± 0	51.3±8.53	38.4±14.23	24.2±4.7563	25±3.16	37±10.59
Missile Command	77 ± 9.50	80 ±0	70 ±11.574	39±8.756	47.27±11.9	59±3.17

Table 5: Average scores over 10 runs obtained from the games: Space Invaders, Frogger, Missile Command. The maximum possible scores, Space Invaders:900, Frogger:54 and Missile Command:80.



(a) Space Invaders

(b) Frogger

(c) Missile Command

Figure 6: For each py-vgdl game, two snapshots are shown. One from the beginning of the game and one well into the game.

Automatic Evolution of Computer Programs and Its Applications. dpunkt-Verlag and Morgan Kaufmann, 1998.

[3] E. Browne, C.B. andPowley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):1–43, 2012.

[4] D.Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas. Gvg-ai competition. http://www.gvgai.net/index.php.

[5] H. Finnsson and Y. Bjornsson. Simulation-based approach to general game playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264, 2008.

[6] S. Gelly and D.Silver. Monte carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175:1856–1875, July 2011.

[7] M. Geneserech and N. Love. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.

[8] 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.

[9] M. Hausknecht, P. Khandelwal, R.Miikkulainen, and P. Stone. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of Genetic and Evolutionary Computation Conference*, 2012.

[10] 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*, 6:355–366, 2013.

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

[12] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994.

[13] J. Levine, C. Bates Congdon, M. Ebner, G. Kendall, S. Lucas, R. Miikkulainen, T Schaul, and T. Thompson. General video game playing. In *Dagstuhl Follow-Ups*, volume 6, pages 77–83, 2013.

[14] Luke. *The ECJ Owner's Manual*, 22 edition, 2014.

[15] M.Campbell, A.Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134:57–83, 2002.

[16] J. Mehat and T. Cazenave. Monte-carlo tree search for general game playing. Technical report, LIASD, Dept. Informatique, Université Paris 8, 2008.

[17] J. Mehat and T. Cazenave. Combining uct and nested monte-carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):225–228, 2010.

[18] 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.

[19] 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*, 518:529–533, 2015.

[20] Y. Naddaf. Game-independent ai agents for playing atari 2600 console games. Master's thesis, University of Alberta, 2010.

[21] 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.