General Video Game Playing by means of Evolution-based Rolling Horizon Algorithm

Oxana Gorshkova

Dissertation 2020

DEPEND Erasmus Mundus Joint MSc in Advanced Systems Dependability

Department of Computer Science,

Maynooth University,

Co. Kildare, Ireland.

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc in Advanced Systems Dependability

Head of Department: Dr Joseph Timoney

Supervisor: Dr Edgar Galvan

Date: 22 June, 2020

Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of Erasmus Mundus Joint MSc in Advanced Systems Dependability qualification, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Oxana Gorshkova Date: 22/06/2020

Acknowledgements

I want to express my sincere gratitude to my supervisor Dr Edgar Galvan for all his guidance and encouragement. I appreciate the time and advice he has given me throughout the course of this project.

A special acknowledgement goes to the Educational, Audiovisual and Cultural Executive Agency (EACEA) that granted me an opportunity to pursue an Erasmus Mundus Joint Master Degree (EMJMD).

I would also like to acknowledge the DJEI/DES/SFI/HEA Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and support.

Abstract

General Video Game Playing represents a branch of Artificial General Intelligence. It aims to develop an agent that can achieve a high level of play of any given game. The Rolling Horizon Evolutionary Algorithm has recently emerged on stage of General Video Game Playing and showed promising results. However, there are some issues to be addressed to boost the algorithm's performance. This work proposes an improvement of the vanilla version of Rolling Horizon Evolutionary Algorithm. The enhancement represents a combination of two techniques: population seeding from statistical tree and replacement of individuals. This work compares the proposed approach with the baseline algorithm, as well as with two other studied enhancements: Shift Buffer and Statistical Tree. The agents based on the algorithms were implemented in the General Video Game AI Framework and assessed on the test set of 20 single-player games. The scores and number of wins achieved by each agent were analyzed with a statistical significance test. The results show that the proposed enhancement of statistical tree seeding combined with replacement of individuals can significantly improve the performance of vanilla version of Rolling Horizon Evolutionary Algorithm. It appeared, that in some games, the proposed algorithm outperforms other popular enhancements applied to the Rolling Horizon Evolutionary Algorithm. Strong and weak sides of the proposed approach can be further investigated by testing the algorithm on more configurations.

Chapter 1: Introduction

1.1 Topic

The topic of this work belongs to the Artificial Intelligence (AI) domain, creation of intelligent game agents in particular. One of the ways to test intelligence of a program is by making it play a game. For a long time, classical board games have been a center of attention for AI software creators. It led to development of world-famous programs that outperformed best human players in games like checkers, chess and Go. Chess and Go were considered particularly difficult challenges for software to overcome, many people were skeptical that computer would ever be able to beat world's leading champions. Nevertheless, in 1997 a program called DeepBlue[1] defeated the best human player in chess and in 2016 AlphaGo[2] gained victory over the professional dan player in Go. Today, computers can defeat human experts in all popular board games with perfect information.

Although such programs as DeepBlue and AlphaGo excelled in games that require strong logical thinking skills, their intelligence is still questionable. These programs' scope of application is extremely narrow as they utilize game-specific heuristics. Their success can arguably be attributed to the program's ability of learning how to play a game. It can be considered rather a programmer's achievement for an agent to excel in a particular game. The human that created an agent can be viewed as a real expert, not a program itself. General game playing (GGP), on the other hand, puts emphasis on the agent's ability to adapt to any given unknown game during the gameplay. If an agent can learn how to play any game and perform well in it, that can represent true intelligence of the program.

General Video Game Playing (GVGP) is a subset of GGP that focuses on AI application in the realm of video games. They provide a rich variety of challenges for AI in the ever-growing game industry. The goal of the GVGP is to create an intelligent agent that learns how to play any previously unseen game on the fly without knowing the rules. These agents decide what strategy to follow based on the information they get during the play.

1.2 Motivation

The purpose of GVGP is to devise generalized algorithms that suit any game, but they can also be beneficial for development of more specialized programs. These approaches are devised to perform optimally under majority of circumstances. They can serve as a good base for an agent in any game because of their versatility. Picking one algorithm and extending it with heuristics for a particular game can be a good choice for someone who is interested in a narrower field. AI approaches developed in GVGP can be applied to virtually any discipline. They can be adopted by AI specialists to create useful programs that are able to solve real life problems.

1.3 Problem statement

The majority of agents employed in GVGP are based on tree search methods like Monte-Carlo Tree Search algorithm, which has shown to yield results superior to other approaches[3]. Recently, evolutionary algorithms started to gain attention in this area and evolution-based agents proved themselves to be successful competitors to MCTS variants. Rolling Horizon Evolutionary Algorithm (RHEA), first applied to GVGP by Perez et al. [4] and compared with MCTS, appeared to be a good alternative to tree search methods. The algorithm has much potential due to its simplicity, but still requires improvement for optimal results.

Due to the requirement placed on the algorithm to provide a solution in very limited time budget, it does not always yield optimal results. The algorithm requires more time to evolve a good strategy from a random generated sequence of actions.

Vanilla RHEA lacks a way to store and use knowledge obtained during the gameplay. The information about the virtual surroundings acquired by previous generations is lost as all individuals are discarded before the start of the next planning stage. This project attempts to improve upon the vanilla version of RHEA.

1.4 Research question

Can population seeding from the statistical tree combined with replacements of individuals improve performance of Vanilla Rolling Horizon Evolutionary Algorithm applied to General Video Game Playing?

1.5 Approach and Metrics

The enhancement of Vanilla RHEA proposed in this work consists of combination of two techniques: population seeding from statistical tree and replacement of individuals in the evolving population. It is suggested to initialize population of the evolutionary algorithm with a solution recommended by statistical tree that is used to gather information throughout the gameplay. The seeding approach was combined with replacement of individuals in the evolving population. Seeding provides algorithm an opportunity to improve on the best sequence of actions found during previous game steps. This reduces a number of generations needed to find a good solution which is beneficial due to the shortage of computational time.

In order to test, assess and compare the proposed enhancement, intelligent agents were implemented in Java programming language using General Video Game AI (GVGAI) Framework. The agents implemented in this work represent the proposed approach, the baseline algorithm (Vanilla RHEA) and two recently studied enhancements of Vanilla RHEA. The correct work of evolutionary algorithm that serves as a base for the agents was verified by a black-box test. The test ensured the algorithm's convergence to the best solution.

In the experiment part of the project, each agent played 4000 times total on 20 games available in the GVGAI Framework. The agents were compared by the number of victories and scores they achieved. The Mann-Whitney U test was carried out on the results to determine statistical significance of algorithms' performance.

Chapter 2: Background

2.1 General Video Game Playing

Video games attract attention of many researchers specializing in General Game Playing (GGP) due to their richness of challenges for AI. Most of the video games belong to the real-time domain; they are not turn-based and require prompt reactions from the player. The game environment can be unpredictable and can change in a blink of an eye. When in some cases a video game agent has milliseconds to fire an action to prevent its defeat, a classical board game agent can take more time to finish its turn. Furthermore, video games feature different virtual-world entities like non-player characters or collectable items that make a gameplay more complex. These aspects of video games make them an excellent benchmark for AI software.

The focus of this work is on creation of versatile planning agents. Planning agents are the agents that learn how to play a game by planning their paths in the virtual environment. When playing a game they complete a series of steps, and in order to choose the most optimal first step, an agent needs to look a few steps into the future. This can be accomplished by simulation of the game before deciding what step to take.

Today, a majority of GVGP planning agents employ tree-based approaches like Monte-Carlo Tree Search (MCTS) algorithm, which has shown to yield results superior to other approaches[3]. MCTS is well known for its employment in the famous AlphaGo program. Recently, evolutionary algorithms started to gain attention in this area and evolution-based agents proved themselves to be successful competitors to MCTS variants[4]. Specifically, Rolling Horizon Evolutionary Algorithm (RHEA) introduced by Perez et al. [4] and implemented alongside MCTS showed very good results and potential.

Today, it is possible for GVGP agents to compete with each other. One of the most popular competitions among AI software creators is “The General Video Game AI Competition” that has been running since 2014[5]. In order to participate in this competition, an agent has to be implemented in the General Video Game AI (GVGAI) Framework [6].

2.2 General Video Game AI Framework

GVGAI Framework written in Java provides a corpus of games as well as means for creating games that can be used as testbeds for various intelligent controllers. The corpus consists of two- and single-player 2D games of different genres: shooter, maze, survival, puzzle, etc. All games have their own rules, scoring and winning criteria. For example, in the game Aliens controller wins when it kills all of the aliens and it gets points for every killed alien and an obstacle destroyed. The graphics of the game can be observed in Fig. 1. The game Survive Zombies, illustrated in Fig.2, considers controller a winner if it is able to stay alive avoiding being eaten by zombies for a particular amount of time. The avatar in this game has a health bar that decreases every time a zombie catches a player. Collecting honey helps restore the health bar. Furthermore, there are obstacles on the player’s way that it needs to avoid while running away from zombies and collecting honey.

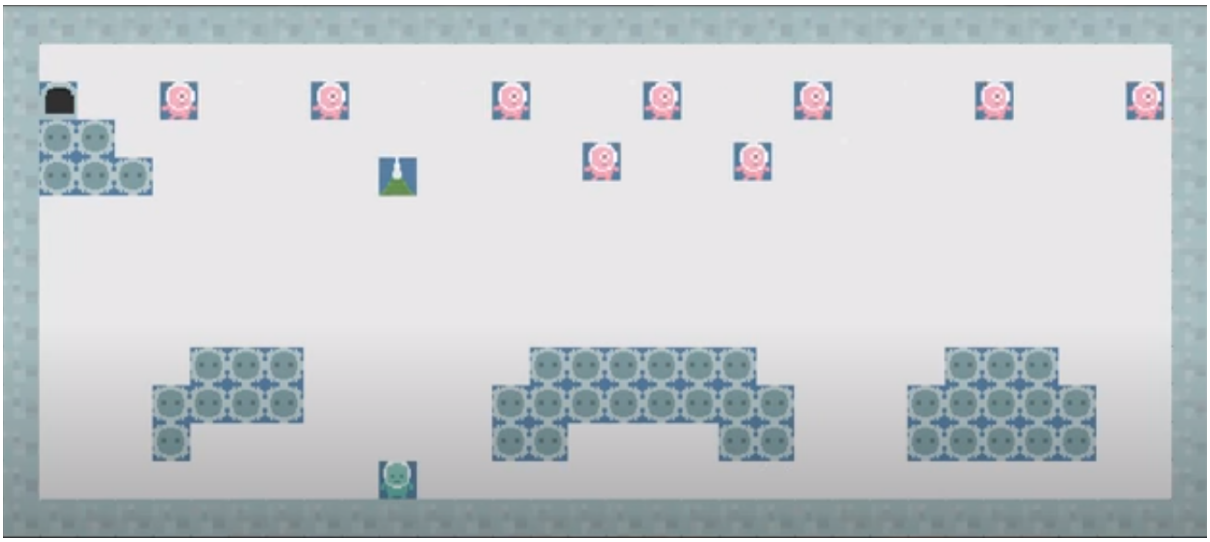


Figure 1 The game Aliens from GVGAI Framework

Figure 2 The game Survive Zombies from GVGAI Framework

Different games have different actions that are legal for a player to execute. Total there are six actions in the framework: *left*, *right*, *up*, *down*, *nil*, *use*, *escape*. In the planning track of the GVGAI Framework, controllers have access to the current game state; they can copy it and execute actions on the copy by using a Forward Model (FM), thus simulating the game. Games can be deterministic or stochastic in nature. In stochastic games, FM can produce different future game outcomes from the current game state with each simulation.

All games can be played on 5 levels. Each level can be played independently; there is no connection between levels and how controller performs, as there is no requirement for a controller to win a lower level to advance to the next level. A higher level can increase a complexity of the game played, it can introduce new challenges to the player. Like some levels may require better exploration from the controller.

In the general setting, a controller has a budget of 40ms to simulate the game and decide on an action to fire. Controller can face disqualification if it exceeds time allotted for decision-making. Thus, it is important for controllers to explore the game environment efficiently having limited resources.

2.3 Evolutionary Algorithms

Evolutionary Algorithms (EA) are heuristic search algorithms that are based on natural evolution principles [7]. As a rule, EAs consist of four main operators: selection, crossover (reproduction), mutation, the creation of a new generation. The cycle of selection, crossover and mutation followed by fitness assessment is called a generation. A classical algorithm of a new generation creation can be represented as follows:

Step 1. Create an initial population of N chromosomes.

Step 2. Assess the degree of fitness of each individual.

Step 3. Select N parents from the population using the selection method (the probability of choosing a parent should depend on its fitness).

Step 4. Select a pair of parents for reproduction from the parent pool. Using the crossover operator, get a descendant.

Step 5. Subject descendants to the mutation operator.

Step 6. Form a new generation of individuals.

Step 7. Assess fitness of each individual in the new population.

Step 7. Go to step 3 if the number of generations does not exceed permissible.

It is important to formalize a problem in such a way that its solution can be encoded into a chromosome - a vector of genes. With regards to GVGP, the genes in the chromosome should be represented by actions, legal in the game played. When applied to GVGAI, EA needs to be modified so that it is possible to stop it at any moment of time as the time budget for decision-making is limited. Instead of creating a pool of parents, we prioritize creation of children by selecting two parents N amount of times.

Rolling Horizon Evolutionary Algorithm (RHEA) appears to be a suitable version of EA for employment in GVGP. The vanilla version of the algorithm was described and applied to GVGP by Perez et al. [4]. Vanilla RHEA is a variant of EA that consists of cycles of constant reevaluation of available solutions – game strategies. Instead of preserving the initial population throughout the game, the algorithm creates a new population from scratch in the beginning of every game tick and does not use any information acquired from previous game steps. Each game tick an agent has to decide on an action to fire, so it creates a population of individuals that are represented by sequences of actions. This representation is illustrated in Fig. 3.

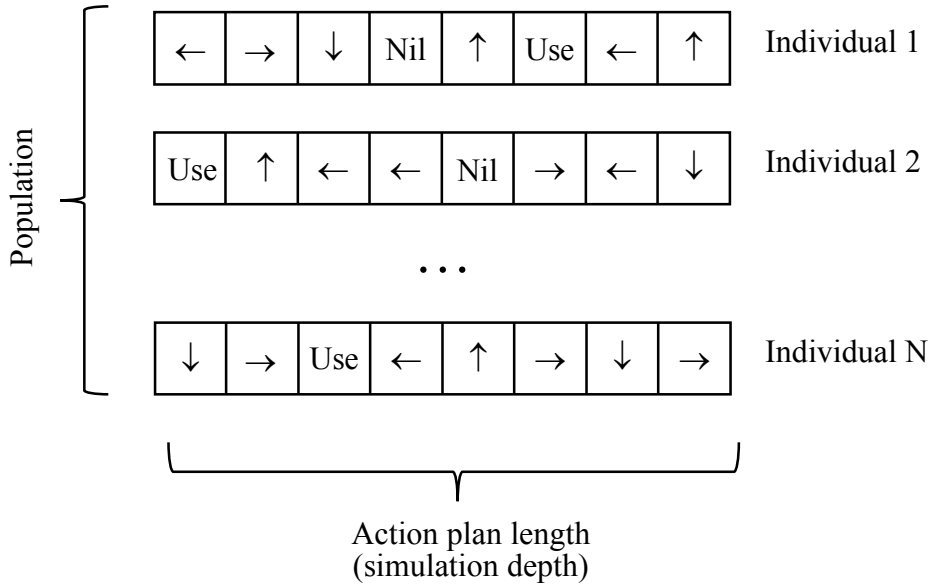


Figure 3 Representation of population used in Rolling Horizon Evolutionary Algorithm for GVGP

The algorithm evolves the population and assesses the individuals by using simulations of the game. In GVGAI Framework, Forward Model (FM) is available to simulate a game. The FM advances the copy of current state of the game by executing every action of the individual's chromosome. The result state of the game is assessed by using a heuristic function and the individual's fitness is assigned the function's value. When the last action is applied, the heuristic function is used to evaluate the produced state and its value is assigned to the individual's fitness. In case of the controller's victory, heuristic function can return a huge positive value, and if controller loses — huge negative value. Otherwise, it can return the score achieved by the controller after the plan of actions was performed. There are various enhancement of vanilla RHEA proposed to date. Two the most successful ones will be observed in this work.

2.4 Shift Buffer Enhancement

Shift Buffer algorithm was introduced by Santos et al. [8]. The idea of Shift Buffer enhancement is, once a population is initialized, to keep it throughout the game play. It does not discard evolved population at every game tick. Instead, it moves each individual's action plan to the left, dropping the first outdated action and adding a new random action at the end as illustrated in Fig. 4. This technique allows the algorithm to keep already found good solutions. Shift Buffer enhancement has been shown to be the most successful modification of Vanilla RHEA to date [10].

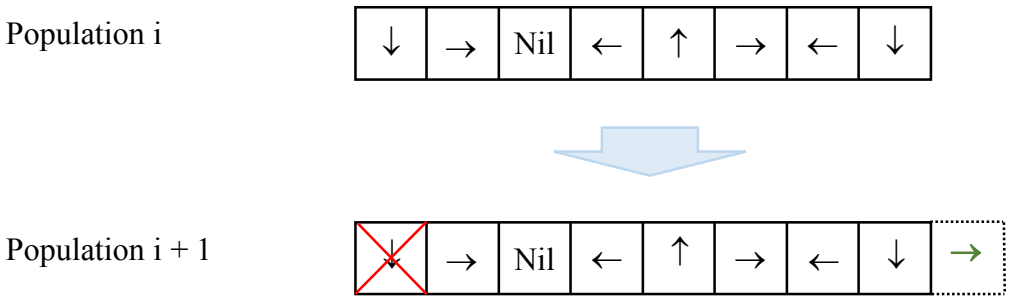


Figure 4 A preservation of an individual across populations with a left shift of its action plan

2.5 Statistical Tree Enhancement

The Statistical Tree enhancement inspired by the tree search algorithms was devised by Perez et al. [9]. It targets the shortcoming of Vanilla RHEA of not using knowledge obtained from previous the game cycles. The idea of this approach is to build a tree with the statistics on scores acquired during the game simulations. The tree is built while evaluating individual's action plan. Each action is added as a node to the tree starting from the root. Root represents a node which action was chosen on the previous game step. Each node contains information of how many times it has been visited, the action assigned to it and accumulated reward. This approach is called an Open Loop approach. It is utilized by many MCTS-like algorithms and it means that the game states (generative models) are not stored in the nodes. It is important because the game states can become obsolete and provide irrelevant information.

The tree is built by evaluating the first action of the individual and this action is added as a child to the root node. All consequent actions are added in the similar manner. When the last action is applied using FM, the game state is evaluated and the result of evaluation is assigned to individual's fitness. This value is then backpropagated to the root, incrementing the node visits count and augmenting stored rewards with the fitness value. An example of this process is presented in Fig. 5. Before advancing to the next step, irrelevant branches of the tree are pruned, and a node with the selected action becomes a new root.

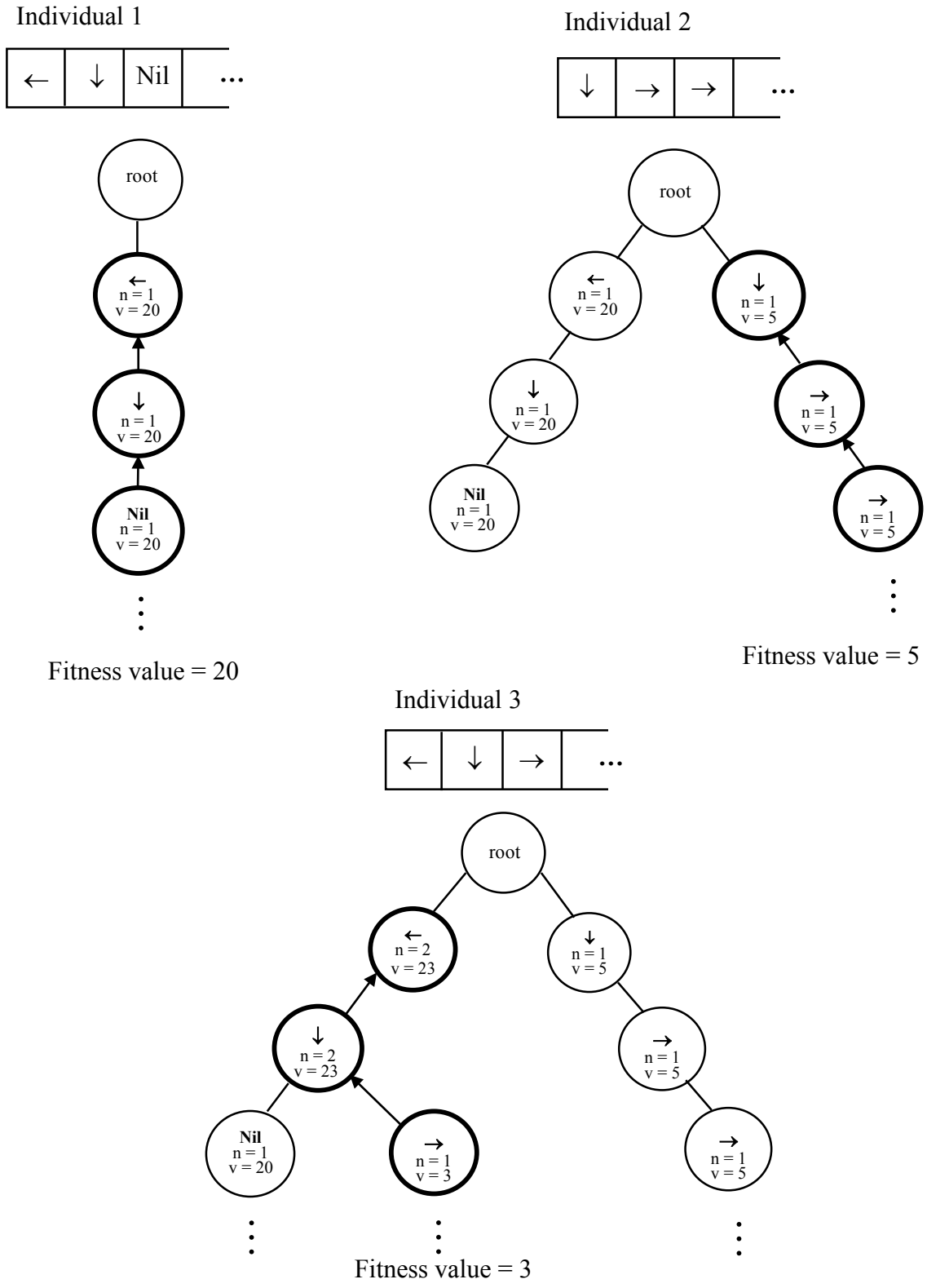


Figure 5 An example of the statistical tree building process while evaluating individuals, where v is accumulated reward of the node, n is the number of times it was visited

Chapter 3: Shortcomings of Vanilla Rolling Horizon Evolutionary Algorithm

Rolling Horizon Evolutionary Algorithm (RHEA) appeared to be a suitable and competitive approach employed by intelligent agents because it is able to perform well in the face of uncertainty caused by changes in game environment. It decomposes a big problem into smaller ones and tackles one of them at a time. The algorithm evaluates its strategies to make just one step into the future. It is done by applying the first action of the best found action sequence. The cycle of planning, evaluation and pushing the planning horizon one step forward is called "rolling horizon". That means that every game step the algorithm starts planning from scratch given updated circumstances. The visual representation of the process is illustrated in Fig 6.

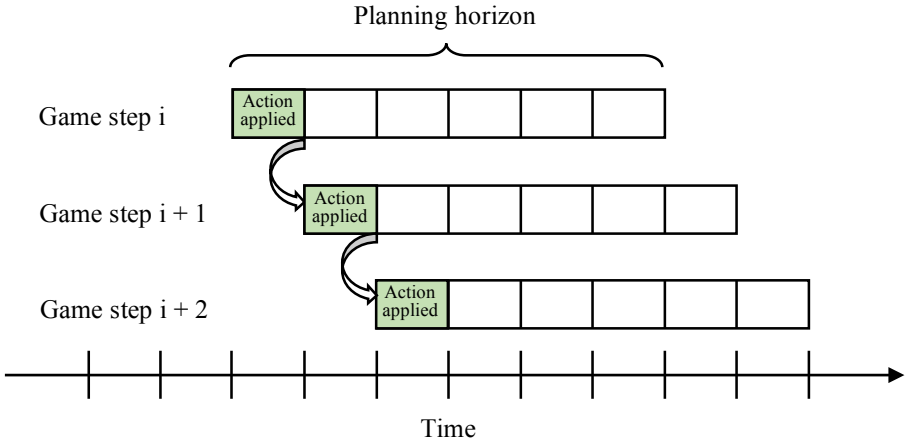


Figure 6 The process of "rolling horizon"

Introduced in 2013 [4], Vanilla RHEA remains to have much room for improvement. Applied to GVGP, a real-time domain, strict limitation in computational time makes it particularly challenging for the algorithm to perform its best. Due to its random nature, Vanilla RHEA is often unable to explore search space efficiently enough and find good solutions. Research showed [11] that Vanilla RHEA yields best results when configured with a bigger population. However, configuring the algorithm with a population of a big size means that there is very little time left for the evolution process, making Vanilla RHEA a random search algorithm. This means, that the algorithm is confined to a relatively small population size in order to save budget for evolution. It has been cognized before [12] that when the population size is relatively small and the initial pool of individuals is not optimal, EA can produce poor results. Populations of small size perform best at refining a solution, while bigger populations are suitable for exploring the search space. It is also known that tree search algorithms like MCTS outperform evolutionary counterparts in planning tracks of GVGAI [3]. One of the reasons for that might be their employment of various policies to target promising game states, balancing between refinement and exploration.

The vanilla variant of RHEA discards evolved population before proceeding to the next game step. This means that there is no information left about the past plans evaluations. The algorithm could use this knowledge to make better decisions. If we look at tree search algorithms, they do not have this drawback. They store statistics about the game states in the tree structure that they carry from one game step to another. They use the knowledge saved before they make a final decision.

Chapter 4: Statistical Tree Seeding with Replacement of Individuals

The enhancement of Vanilla RHEA proposed in this work consists of combination of two techniques: population seeding from statistical tree and replacement of individuals in the evolving population. The proposed enhancement is based on the idea of building and keeping a statistical tree alongside the evolving population in the same manner as described in chapter two, section 2.5. Keeping the tree provides some knowledge of the game space explored by previous populations. However, instead of exploiting EA to build a statistical tree that dictates final action selection as proposed by Perez et al. [9], it is suggested to utilize the tree to start EA with a more optimal pool of individuals. The final action selection of the proposed enhancement is done in the same way as in Vanilla RHEA: by choosing the first action of the fittest individual, while the statistical tree is used to guide evolution.

4.1 Population Seeding from the Statistical Tree

The technique described in this section addresses the shortcoming of Vanilla RHEA of having difficulty to find a good solution in limited time. Instead of starting with a randomly initialized population, the proposed algorithm works on improvement of the best action plan explored so far. Seeding the population with an already good solution helps reduce the number of generations needed to produce better results.

The statistical tree is built in the same way as in Statistical Tree enhancement. The scores stored in tree nodes are updated each time an individual is evaluated. If there was no node that corresponds to an action of an individual being evaluated, the node is added to the tree. The pseudocode for evaluation strategy can be found in Algorithm 1 starting from line 13.

The first individual of a new population is initialized by traversing the tree and selecting nodes that have the highest mean reward. This is done in line 4 of the pseudocode of Algorithm 1 that can be found at the end of this chapter. That gives the algorithm a chance to improve on the best action plan explored so far. To target more efficient exploration of the game space, it is suggested to initialize other N-1 individuals using the UCB (Upper Confidence Bound) algorithm (line 6 of the Algorithm 1). One of the simple, yet efficient, versions of the algorithm is UCB1, proposed by Auer et. al [13]. The selection policy's objective is to opt for a node that maximizes the value

$$UCB1 = \overline{x_j} + c\sqrt{\frac{2\ln n}{n_j}}, \quad (1)$$

where $\overline{x_j}$ is a mean reward of the jth child node, n_j is the number of times when the jth child node was visited, n is the number of times the parent (current) node was selected, c is an exploration constant. The reward value of the child node x_j is required to belong to interval [0,1] for the UCB1 algorithm to work correctly. The left term of Equation (1) encourages selection of the nodes with high reward values, while the right term stimulates visiting less explored but potentially profitable nodes. This policy also helps avoid further exploration of child nodes that yielded the least scores. Equation (1) is widely used as a tree selection policy in MCTS-like algorithms and ensures that only promising nodes are visited, encouraging better exploration of the search space [14]. Implementing this policy, RHEA works on exploring promising branches of the statistical tree expanding unvisited paths. This way the tree provides richer information for a better choice of individuals in subsequent game steps. It is important

to evaluate freshly created individual before initializing a next one. Evaluation of individual updates stored statistics. This ensures that the tree provides up-to-date information for the seeding of the next individual.

Two other techniques of population initialization of Vanilla RHEA have been studied before [15], showing that seeding can significantly improve performance of the vanilla version of the algorithm. One of the techniques is to run MCTS algorithm, which uses half of the budget to build a tree from which the first individual of a new population is created. The second is One Step Look Ahead algorithm that employs exhaustive search to create action plan of the first individual. Both of these approaches significantly reduce time for the evolution process, preventing RHEA from realizing its full potential. Keeping the statistical tree is a much cheaper operation and it does not require additional simulations of the game.

4.2 Replacement of Individuals

After some initial testing, it has been observed that the proposed seeding technique yields best results when combined with injection of new individuals using a replacement method. Generally, it is considered a good practice to remove the worst individual among the most similar ones to diversify population. In this work, a simpler version is chosen: a new individual substitutes the worst one if its fitness value is greater.

As the statistical tree represents the search space explored during the evolution cycles, it might suggest some good variants of action plans for the evolving population. In this work, starting from when the population is half evolved, a new individual using UCB1 policy is created at the end of each generation. Its fitness is compared to the fitness of the worst individual in the population. If the new individual's action plan yields a better score, it replaces the worst individual. This technique can increase diversity in the population and provide good solutions by examining potentially profitable branches of the statistical tree.

Algorithm 1. Population Seeding Algorithm

```
Procedure InitializePopulation()
For i is 0, i is less than POPULATION_LENGTH, i increments by 1
      If i is equal to 0
            individual = CreateIndividual(BestAverageReward)
      Else
            individual = CreateIndividual(UctSearch)
      EndIf
      Evaluate(individual)
      population.add(individual)
EndFor
End Procedure

Procedure Evaluate(individual)
parent = statisticalTree.root
For i is 0, i is less than INDIVIDUAL_LENGTH and not gameState.GameOver, i
increments by 1
      gameState.applyAction(individual.actions[i])
      parent = parent.addChild(individual.actions[i])
EndFor
score = heuristics.evaluateState(gameState)
parent.backpropagate(score)
individual.fitness = score
End Procedure
```

```
Procedure CreateIndividual(policy)
individual = new individual()
parent = StatisticalTreeContext.root
chosenChild = nil
For i is 0, i is less than INDIVIDUAL_LENGTH, i increments by 1
	chosenChild = policy(parent)
	individual.appendAction(chosenChild.action)
	parent = chosenChild
EndFor
Return individual
EndProcedure

CallBack UctSearch(node)
selectedChild = nil
bestValue = -MAX_VALUE
Foreach child in node.children
	childValue = normalize(child.accumulatedScore / child.nVisits)
	uctValue = childValue + c * sqrt(Math.log(node.nVisits/child.nVisits))
	applyNoise(uctValue)
	If uctValue is greater than bestValue
		selectedChild = child
		bestValue = uctValue
	EndIf
EndForeach
Return selectedChild
EndCallBack

CallBack BestAverageReward(node)
selectedChild = nil
bestValue = -MAX_VALUE
Foreach child in node.children
	If child.nVisits is greater than 0
		childValue = child.accumulatedScore / child.nVisits
		applyNoise(childValue)
		If childValue is greater than bestValue
			selectedChild = child
			bestValue = childValue
		EndIf
	EndIf
EndForeach
Return selectedChild
EndCallBack
```

Chapter 5: Development of Intelligent Agents

5.1 Design and Implementation

For the experiment part of the project, I implemented four controllers that represent four evolutionary approaches. The algorithms are Statistical Tree Seeding with Replacement of Individuals proposed in chapter 4, Shift Buffer described in chapter 2 section 2.4, Statistical Tree described in chapter 2 section 2.5 and Vanilla Rolling Horizon Evolutionary Algorithm (RHEA). Controllers used in this work were implemented from scratch. They are designed to work in the environment of General Video Game AI (GVGAI) Framework, described in chapter 2. They implement an interface that allows them to communicate with the framework. Controllers submit actions, access games states and simulate games by using the framework's mechanics.

Because all algorithms have Vanilla RHEA as their base, instead of implementing four different classes for each controller, I created one that changes its behavior based on the type of algorithm chosen. All algorithms differ in the way they initialize population and evaluate individuals. To avoid code repetition and to increase flexibility, I implemented a Strategy pattern[16] for these operators. Strategies can be substituted one for another. This provides flexibility to test and mix different approaches. Adoption of the Strategy pattern can be useful for future work on extension and improvement of Vanilla RHEA. Using this pattern helps avoid creation of a new class for every combination of operator variants. Inheritance in this case can result in code repetition and/or exploding class hierarchy. The UML diagram of EA classes is presented in Fig. 7.

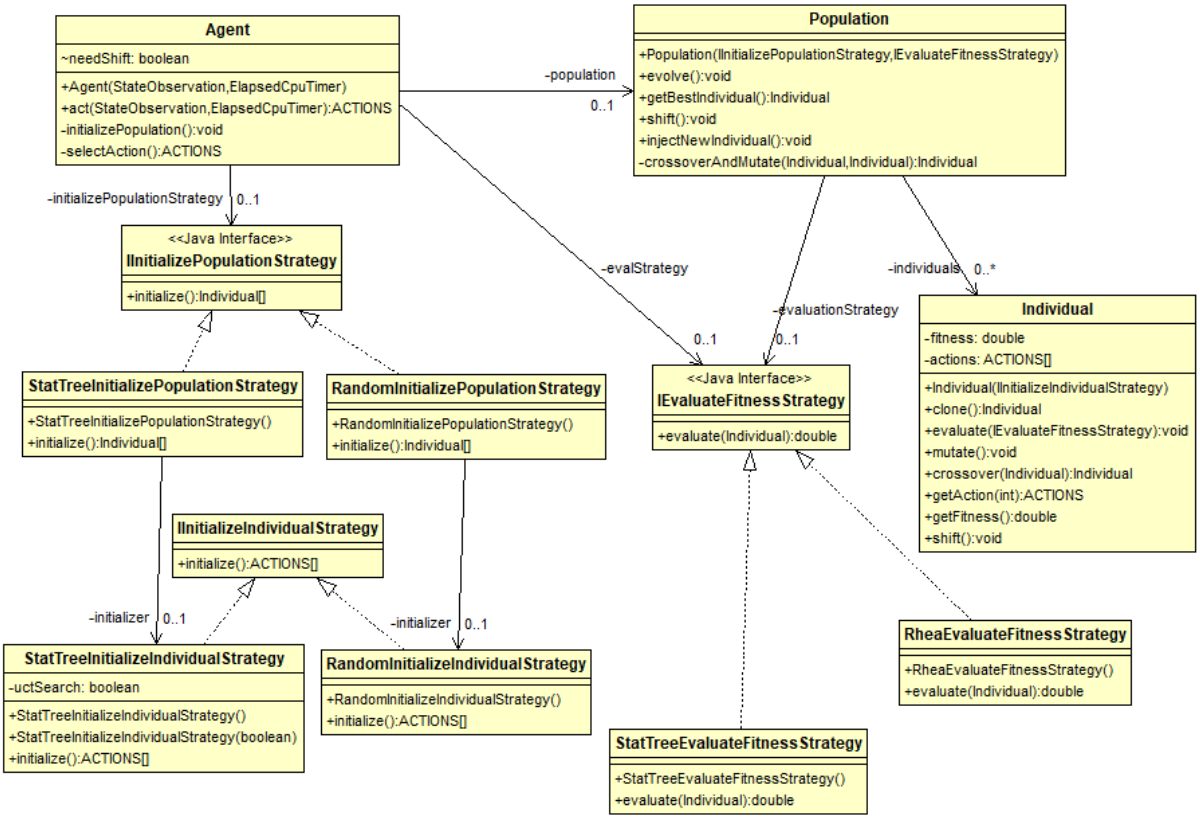


Figure 7 UML diagram of EA classes

To store all configuration of the algorithms in one place, I created a context class that contains parameters like budget for simulation, population size, mutation rate etc. The context also stores the current game state and heuristic that is used to evaluate individual's action plan. This context can be accessed by all strategy classes as well as from the core classes such as Population and Individual. This context ensures that all variants of algorithms have the same parameters thus fair conditions. It also makes it easy to change configuration because it is stored in one place and applies to every variant of the base algorithm. The context classes are illustrated in Fig. 8. The project file structure can be observed in Appendix A.

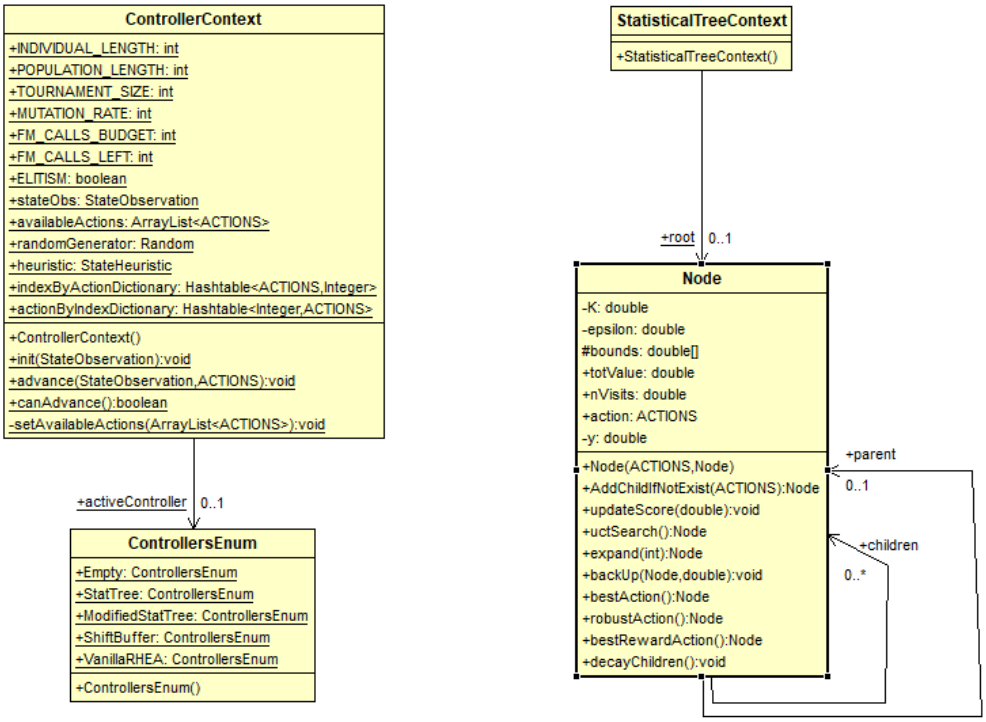


Figure 8 UML diagram of context classes

5.2 Software Verification

Testing EA is a challenging task due to its random nature. Testing a controller in the setting of GVGAI Framework is even more complex as it depends on the games and Forward Model that is used to simulate games. The class structure I designed allows implementing different strategies for evaluation of individuals. I implemented a test strategy that makes the evolutionary algorithm (base for all controllers) independent from the GVGAI framework. I chose to evaluate individuals based on their resemblance to the randomly created reference action plan. The more actions an individual's action plan has at the same position as the reference plan, the higher its fitness. To test that the algorithm works as expected I wrote a black-box test using JUnit that checks convergence of the algorithm to the reference action plan. Because the algorithm depends on randomness, I repeated the test 10000 times to verify it works correctly. The details of the test can be seen in Table 1.

Table 1. Functional test details

Input	Randomly created reference action plan
Expected Output	Best individual's action plan is identical to the reference action plan
Actual Output	As expected
Number of Generations	500
Number of Test Runs	10000
Pass/Fail	10000/10000 Pass, 0/10000 Fail

5.3 Collection and Processing of Output Data

One of the most challenging parts of this project was to collect and analyze all the results that agents achieved. To assess performance of the algorithms, controllers had to run 40 times on 5 levels of 20 chosen games. Each controller had to play 4000 times, resulting in total of 16000 runs by all controllers in one configuration. To perform this amount of executions, additional computational resources were required. I was kindly granted an access to the supercomputer Kay[17], that is provided to academic researchers by ICHEC. Kay runs a job scheduling system SLURM[18] that provides easy means to execute parallel software.

Because controllers can play games simultaneously and do not depend on each other, I adopted an embarrassingly parallel execution model for my program. The entry point was designed to accept a game index and a controller name for the specified game to be played 200 times (40 times on 5 levels). I submitted a job to SLURM via a bash script that contained a reference to a runnable JAR file of my program. An example of a script is provided in Fig. 9. The bash script specifies job steps, each step launches a jar file with parameters that indicate an index of a game and a name of a controller. Each step produces a file as an output. The file name contains a name of the controller and a name of the game that were passed as parameters. Each row of the file is attributed to one execution of the game and contains two values: a win value and a score value. A win value can be either 1 for a victory or 0 for a loss.

To collect all the data from the files and perform initial statistical analysis on it, I wrote a parser in the C# programming language. The parser reads all results files, structures data and outputs it into Excel files. I performed the rest of the statistical analysis, including the Mann-Whitney U test, in Excel by writing macros and executing them on the data. The parser and all result files, including Excel files, are submitted with the code.

```
#SBATCH --job-name=katzenklo_test
#SBATCH --time=19:40:00
#SBATCH --nodes=1
#SBATCH -A nuim01
#SBATCH -p ProdQ
#SBATCH --output="output"
#SBATCH --workdir="/ichec/home/users/katzenklo/controllers"

controller1="StatTree"
module load java/11

srun --ntasks=1 java -jar controllers.jar 1 $controller1 &
srun --ntasks=1 java -jar controllers.jar 2 $controller1 &
srun --ntasks=1 java -jar controllers.jar 5 $controller1 &
srun --ntasks=1 java -jar controllers.jar 8 $controller1 &
srun --ntasks=1 java -jar controllers.jar 9 $controller1 &
srun --ntasks=1 java -jar controllers.jar 10 $controller1 &
wait
```

Figure 9 An example of a bash script used to execute games on Kay via SLURM

Chapter 6: Evaluation

The hypothesis that the proposed approach can improve the performance of Vanilla RHEA was tested using the GVGAI Framework. The suggested enhancement was compared to the baseline algorithm Vanilla RHEA as well as its previously studied enhancements: Statistical Tree and Shift Buffer, described in chapter two. The measured output values are number of victories and scores achieved by algorithms in one-player games. In order to reject the null hypothesis, a statistical significance test was carried out on the obtained results.

6.1 Experimental Setup

To ensure that all algorithms have fair conditions, they were tested with the same configuration: population size was set to 10 and the length of the individual (action plan) — to 14. All algorithms used uniform crossover operator and a tournament selection with the tournament size of 3. The mutation rate was set to 80%. The use of elitism ensured that the best individual was carried to the next generation. Algorithms were not limited in the number of generations, the evolution process continued until the algorithm was out of given budget. To make results of the experiment machine independent, the budget for an action decision was set to 900 FM calls. FM calls are used as a measure of time, because simulating the game is the most computationally expensive operation. 900 is an average number of FM calls that Vanilla RHEA performs in 40ms in the GVGAI Framework [15].

The algorithms were run 40 times on all 5 levels of each game, resulting in 200 runs total per controller per game. All results were converted into Formula-1 (F1) ranking scheme. For each game, controllers are compared by their mean win rate. The controller with the highest average win rate gets 25 points, second best controller is granted 18 points, and others get 15 and 12 points accordingly. If controllers achieved the same number of victories, they are compared by the scores they acquired, and the best one gets the most points.

The Mann-Whitney U test at 5% significance level was used to compare controllers with each other to determine if the difference in scores and wins is statistically significant.

6.1.1 Games

For the evaluation of the algorithms, a set of 20 games from GVGAI Framework was chosen. The set contains 10 stochastic and 10 deterministic games and is presented in Table 2. The indexes provided are used in the results tables to refer to a particular game.

Table 2. A set of 20 games used in experiments

Index	Stochastic	Index	Deterministic
1	Aliens	11	Bait
2	Butterflies	12	Camel Race
3	Chopper	13	Chase
4	Crossfire	14	Escape
5	Dig Dug	15	Hungry Birds
6	Infection	16	Lemmings
7	Intersection	17	Missile Command
8	Roguelike	18	Modality
9	Sea Quest	19	Plaque Attack
10	Survive Zombies	20	Wait for Breakfast

This particular games set is chosen because it was used in many previous research works like [10], [8], [11] and it is considered to be a representative assortment of various challenges present in GVGAI corpus. The results shown in this section can be compared to ones already acquired by algorithms used by other researchers.

6.1.2 Controllers

Four controllers were used in this experiment. The baseline algorithm Vanilla RHEA is implemented as a *Vanilla* controller, controller representing the proposed enhancement is called *TreeSeeding*. The other two controllers used for results comparison are *StatTree* and *ShiftB*. *StatTree* controller implements the enhancement described in chapter 2, section 2.5. As a recommendation policy, *StatTree* selects an action to play by opting for a child with the highest average reward. Statistical trees used by *StatTree* and *TreeSeeding* are carried from one game step to another. At the end of each game step, the values stored in the nodes – visit count and accumulated reward – are multiplied by a decay factor of 0.99. This reduces weight of old statistics, as it becomes outdated due to the dynamic environment of the games. *ShiftB* controller represents enhancement described in chapter 2, section 2.4. The whole population is shifted one action to the left, a random action is appended to the end of each individual's action plan. The shifted population is then carried to the next game step.

6.2 Results and Discussions

The controllers were assigned Formula-1(F1) points and ranked by the total points achieved. The idea of the F1 ranking scheme is described in section 6.1. The ranking and results for each game in F1 points are presented in the Table 3. The higher the number in the cell and the darker its color, the better results achieved by a controller.

Table 3. F1 points achieved by controllers, where algorithms in the first column are 1 – *TreeSeeding*, 2 – *StatTree*, 3 – *ShiftB*, 4 – *Vanilla*. The darker the color of the cell, the better results achieved by a controller.

	F1 Points	Avg. Wins	G-1	G-11	G-2	G-12	G-13	G-3	G-4	G-5	G-14	G-15	G-6	G-7	G-16	G-17	G-18	G-19	G-8	G-9	G-10	G-20
1	412	49.67%	15	25	15	25	18	25	25	15	25	18	15	25	12	18	18	25	18	25	25	25
2	382	48.37%	18	18	25	18	25	12	15	18	12	25	25	15	25	25	25	18	15	15	15	18
3	315	45.92%	25	12	18	15	12	18	12	25	15	12	18	12	18	15	12	12	25	12	12	15
4	291	46.77%	12	15	12	12	15	15	18	12	18	15	12	18	15	12	15	15	12	18	18	12

Results show, that *Vanilla* controller gained the least amount of points and the proposed algorithm showed the best results, ranking first in 10 games. For more detailed results achieved by controllers in each game in terms of wins percentage and scores, the reader is referred to Appendix C1-4. Grouped bar charts with confidence intervals that show win rates across games are presented in Fig. 10 and Fig. 11.

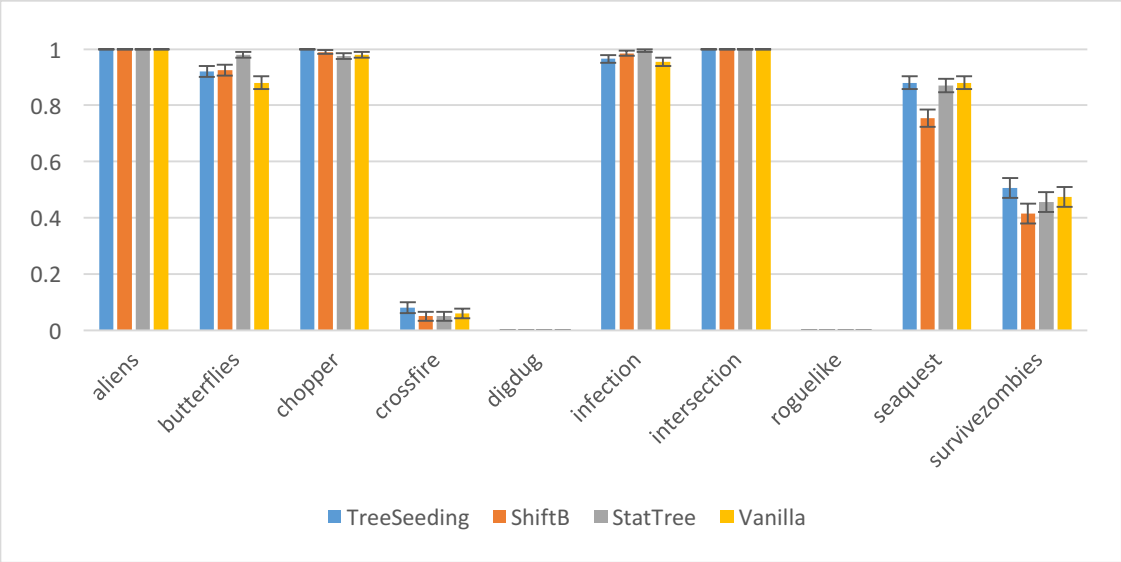


Figure 10 Win rate of controllers in stochastic games. Confidence interval is expressed in terms of standard error.

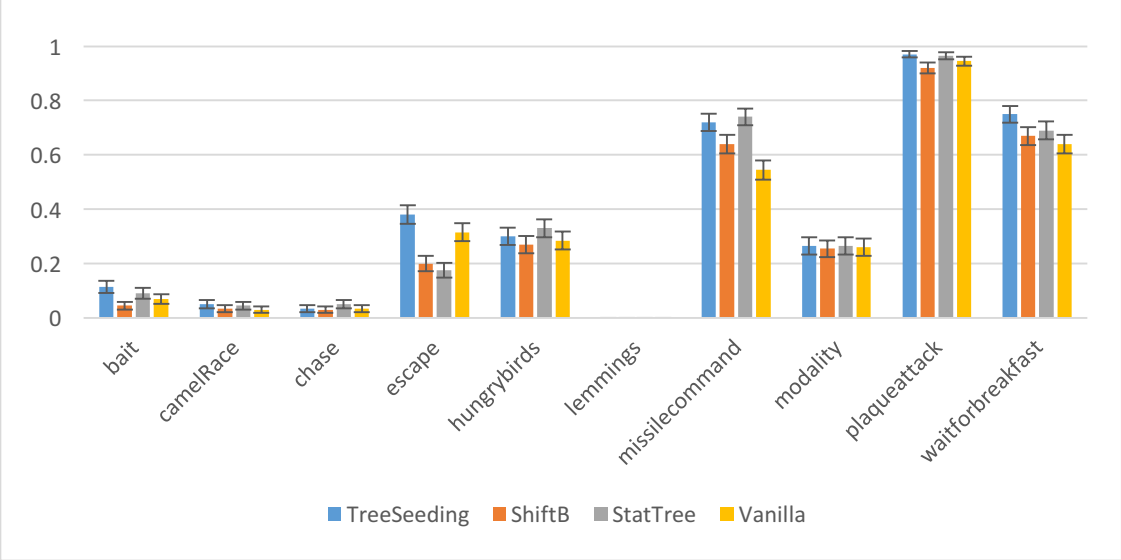


Figure 11 Win rate of controllers in deterministic games. Confidence interval is expressed in terms of standard error.

The win rates indicate that all controllers performed better in stochastic games on average, but it is not convincing. This might be attributed to the fact that games in stochastic section of the test set are less challenging than the ones in deterministic section. In games like *Lemmings*, *Dig-Dug* and *Roguelike* none of the controllers was able to win. That was expected, as in previous research [10], [8], [11] RHEA performed the same way in these games. The controllers' inability to succeed can be attributed to the difficulty of the games. They require from algorithms high ability to explore the game while avoiding entities that can kill the player. To achieve victory in these games, controllers have to focus on multiple goals.

Mann-Whitney U test was performed to determine the significance of the results. The number of games, in which controllers achieved significantly higher win rates than algorithms they are compared to are presented in Table 4.

Table 4. Significance comparison of controllers by win percentage. The number indicates the number of games in which controller specified in the first column significantly **outperformed** a controller in first the row.

Compared to / Algorithm	TreeSeeding	ShiftB	StatTree	Vanilla
TreeSeeding		4	2	3
ShiftB	0		0	0
StatTree	2	3		3
Vanilla	0	2	1	

Based on significance analysis, we can say with 95% confidence, that the proposed agent outperformed *Vanilla* in 3 games in terms of win rates. These games are *Wait For Breakfast*, *Missile Command* and *Chopper*. The *StatTree* controller has the same number of games in which it outperformed *Vanilla*, but they are *Infection*, *Butterflies*, *Missile Command*. The two approaches share similarity in the fact that they use statistical tree for information keeping, and that seems to be the reason why they outperform *Vanilla* in *Missile Command.* Worth mentioning, that there are no games in which *TreeSeeding* is worse than *Vanilla* in terms of victories while *StatTree* is inferior to Vanilla in game *Escape*. In fact, even *ShiftB* showed worse results in *Escape*.

Interesting to note the difference between *StatTree* and *TreeSeeding*. *TreeSeeding* performs significantly better than *StatTree* in games *Chopper* and *Escape* while *StatTree* leads in games *Butterflies* and *Infection*. The difference in these games applies to both win rate and scores. The result of significance comparison in terms of achieved scores is shown in Table 5.

Table 5. Significance comparison of controllers by achieved scores. The number indicates the number of games in which controller specified in the first column significantly **outperformed** a controller in first the row.

Compared To / Algorithm	TreeSeeding	ShiftB	StatTree	Vanilla
TreeSeeding		4	3	6
ShiftB	2		0	5
StatTree	4	4		5
Vanilla	1	1	0	

Although *TreeSeeding* has not shown results inferior to *Vanilla* in terms of win rate, there is a game *Dig-Dug* in which it gained significantly less scores. Nevertheless, *TreeSeeding* has

shown the most number of games (6) compared to other enhancements in which it is superior to *Vanilla*. These games are *Wait for Breakfast*, *Roguelike*, *Crossfire*. *Chopper*, *Intersection*.

The results show, that although *StatTree* and *TreeSeeding* have similarities in their nature, there is a significant difference in the way they perform in some games. In the game *Chopper*, the scene changes very rapidly and the player has to shoot flying objects to get points and avoid moving tanks. Poor performance of *StatTree* can be caused by information in the tree getting outdated quickly. The proposed approach does not have this shortcoming because it evolves the best action instead of following a recommendation of the tree.

The external validity threat in this work lies in the relatively small amount of games in the test set. The threat can be reduced by using more games of different genres in the experiment. Another external validity threat is caused by testing algorithms in single configuration of population size and individual length. The threat can be reduced by testing solution on various configurations of EA.

Chapter 7: Conclusion

7.1 Contribution to the state-of-art

The proposed enhancement of the vanilla version of Rolling Horizon Algorithm (RHEA) showed significantly improved performance of the algorithm in some games of the chosen test set. It appeared, that in some games, the proposed algorithm of tree seeding with replacement of individuals outperforms other popular enhancements applied to the Vanilla RHEA.

7.2 Future Work

For deeper investigation, the performance of the proposed enhancement can be tested on more configurations of the evolutionary algorithm. This can help to better determine its weak and strong sides.

Two techniques that constitute the proposed approach, namely population seeding and replacement of individuals, can be tested separately to determine their contribution to the improved performance. In this work, replacement of individuals with the worst fitness was tested. To better target population diversity, replacement of the least fitted individual out of the most similar individuals can be adopted in future investigations.

The population seeding technique proposed in this work can be compared to other seeding techniques introduced by Gaina et al [15] and mentioned in chapter 4. This requires implementation of additional agents.

In the experiment part of the work, 20 games were chosen to observe the effect of employed enhancement of Vanilla RHEA. For stronger conclusions on the improved performance of the baseline algorithm, more testing can be done on a wider range of games.

References

[1] M. Campbell, A. J. Hoane, and F. Hsu, “Deep Blue ,” *Artificial Intelligence* , vol. 134, no. 1. Elsevier B.V , AMSTERDAM , pp. 57–83, 2002.

[2] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search ,” *Nature* , vol. 529, no. 7587. NATURE PUBLISHING GROUP , LONDON , pp. 484–489, 2016.

[3] P. García-Sánchez and P. García-Sánchez, “Georgios N. Yannakakis and Julian Togelius: Artificial Intelligence and Games: Springer, 2018, Print ISBN: 978-3-319-63518-7, Online ISBN: 978-3-319-63519-4, https://doi.org/10.1007/978-3-319-63519-4 ,” *Genetic Programming and Evolvable Machines* , vol. 20, no. 1. Springer US , New York , pp. 143–145.

[4] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, “Rolling horizon evolution versus tree search for navigation in single-player real-time games ,” *Proceedings of the 15th annual conference on genetic and evolutionary computation* . ACM , pp. 351–358, 2013.

[5] D. Perez-Liebana *et al.*, “The 2014 General Video Game Playing Competition ,” *IEEE Transactions on Computational Intelligence and AI in Games* , vol. 8, no. 3. IEEE , PISCATAWAY , pp. 229–243, 2016.

[6] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms .” 2018.

[7] T. Bäck, “Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms .” Oxford University Press , New York , 1996.

[8] B. Santos, H. Bernardino, and E. Hauck, “An Improved Rolling Horizon Evolution Algorithm with Shift Buffer for General Game Playing,” in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 2018, pp. 31–316, doi: 10.1109/SBGAMES.2018.00013.

[9] D. Perez Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. Lucas, “Open Loop Search for General Video Game Playing ,” *Proceedings of the 2015 Annual Conference on genetic and evolutionary computation* . ACM , pp. 337–344, 2015.

[10] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, “Rolling horizon evolution enhancements in general video game playing ,” *2017 IEEE Conference on Computational Intelligence and Games, CIG 2017* . pp. 88–95, 2017.

[11] R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, “Analysis of vanilla rolling Horizon evolution parameters in general video game playing ,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* , vol. 10199. pp. 418–434, 2017.

[12] P. Diaz-Gomez and D. Hougen, *Initial Population for Genetic Algorithms: A Metric Approach.* 2007.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem ,” *Machine Learning* , vol. 47, no. 2. Kluwer Academic Publishers , Boston , pp. 235–256, 2002.

[14] C. B. Browne *et al.*, “A Survey of Monte Carlo Tree Search Methods ,” *IEEE Transactions on Computational Intelligence and AI in Games* , vol. 4, no. 1. IEEE , PISCATAWAY , pp. 1–43, 2012.

[15] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, “Population seeding techniques for Rolling Horizon Evolution in General Video Game Playing ,” *2017 IEEE Congress on Evolutionary Computation (CEC)* . IEEE , pp. 1956–1963, 2017.

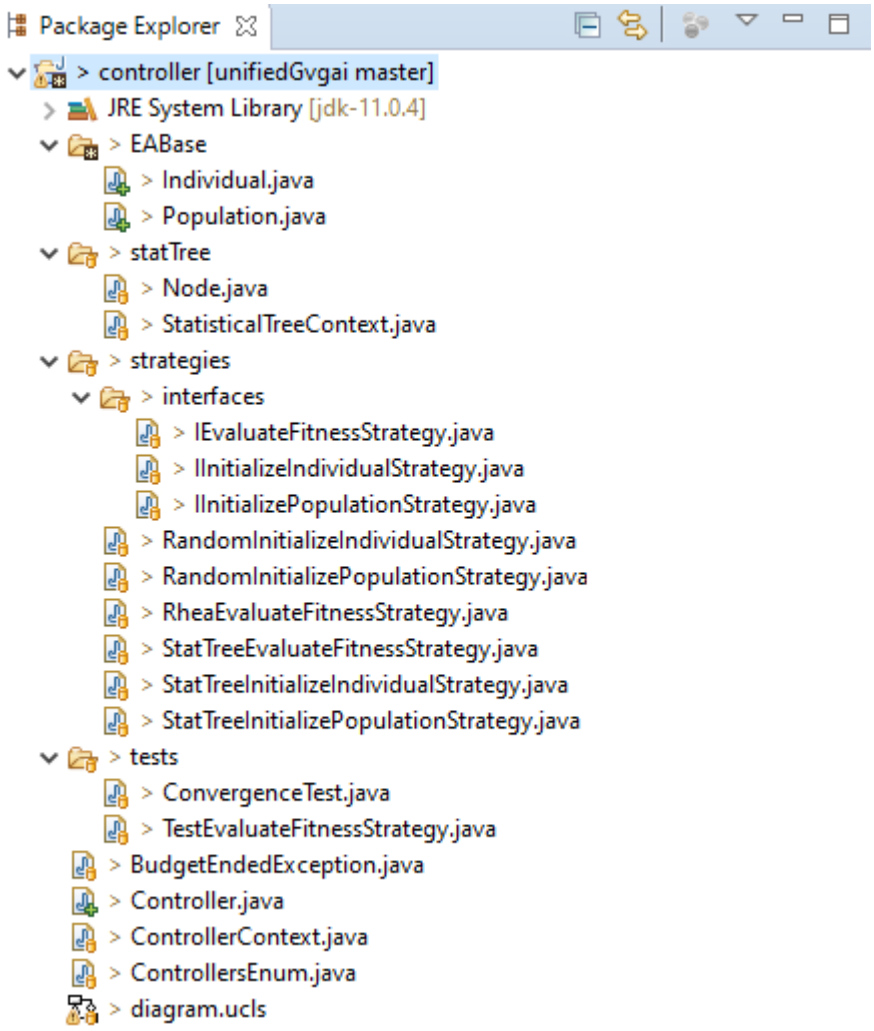
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*. Addison-Wesley, 1994.

[17] “Kay | ICHEC.” [Online]. Available: https://www.ichec.ie/about/infrastructure/kay. [Accessed: 18-Jun-2020]

[18] “Slurm Workload Manager - Overview.” [Online]. Available: https://slurm.schedmd.com/overview.html. [Accessed: 18-Jun-2020]

Appendix A

The structure of the source code.



Appendix B

Convergence test of the EA

```
package tracks.singlePlayer.advanced.controller.tests;

import static org.junit.Assert.*;
import java.util.ArrayList;
import java.util.Arrays;
import org.junit.Test;
import org.junit.jupiter.api.RepeatedTest;
import ontology.Types.ACTIONS;
import tracks.singlePlayer.advanced.controller.ControllerContext;
import tracks.singlePlayer.advanced.controller.Population;

public class ConvergenceTest {

	final static int NUMBER_OF_GENERATIONS = 500;

	/**
	 * Tests convergence of genetic algorithm
	 *
	 * @result Best individual's actions should be identical to
reference actions by
	 *         the end of the evolution
	 */
	@Test
	@RepeatedTest(10000)
	public void test() {
		// Arrange
		ControllerContext.availableActions = new
ArrayList<ACTIONS>(Arrays.asList(ACTIONS.values()));
		// so code works properly and canAdvance() returns true
		ControllerContext.FM_CALLS_LEFT =
ControllerContext.FM_CALLS_BUDGET;
		var evalStrategy = new TestEvaluateFitnessStrategy();
		var referenceActions = evalStrategy.referenceActions;
		var population = new Population(new
TestInitilizePopulationStrategy(), evalStrategy);
		var initialBestIndividual = population.getBestIndividual();

		// Act
		for (int i = 0; i < NUMBER_OF_GENERATIONS; i++) {
			population.evolve();
		}

		var bestIndiv = population.getBestIndividual();

		// Assert
		// Check if the initial best individual was random
		// makes sure that it evolved during GA execution
		boolean allEqual = true;
		for (int i = 0; i < ControllerContext.INDIVIDUAL_LENGTH;
i++) {
			if (referenceActions[i] !=
initialBestIndividual.getAction(i)) {
				allEqual = false;
			}
		}
```

```
                assertEquals("Best individual was already the fittest",
false, allEqual);

                // Check if the best individual's actions match the
reference actions
                for (int i = 0; i < ControllerContext.INDIVIDUAL_LENGTH;
i++) {
                        assertEquals("Action with index " + i + "differs",
referenceActions[i], bestIndiv.getAction(i));
                }

                assertEquals("The fitness is incorrect",
ControllerContext.INDIVIDUAL_LENGTH, bestIndiv.getFitness(), 0.000005);
        }

}
```

Appendix C1

Detailed results in each game achieved by *TreeSeeding* controller

TreeSeeding						
Game name	Mean wins	Wins standard deviation	Wins standard error	Mean scores	Scores standard deviation	Scores standard error
aliens	1	0	0	62,275	12,80427	0,905399
bait	0,115	0,319022	0,022558	6,265	8,295467	0,586578
butterflies	0,92	0,271293	0,019183	33,09	16,27857	1,151069
camelRace	0,05	0,217945	0,015411	-0,75	0,53619	0,037914
chase	0,035	0,18378	0,012995	2,49	2,229327	0,157637
chopper	1	0	0	17,745	2,896545	0,204817
crossfire	0,08	0,271293	0,019183	0,135	1,498925	0,10599
digdug	0	0	0	15,54	11,71744	0,828548
escape	0,38	0,485386	0,034322	0,35	0,53619	0,037914
hungrybirds	0,3	0,458258	0,032404	30,2	45,78166	3,237252
infection	0,965	0,18378	0,012995	14,62	8,362751	0,591336
intersection	1	0	0	2,085	3,359728	0,237569
lemmings	0	0	0	-0,55	0,89861	0,063541
missilecommand	0,72	0,448999	0,031749	5,3	4,749737	0,335857
modality	0,265	0,441333	0,031207	0,265	0,441333	0,031207
plaqueattack	0,97	0,170587	0,012062	54,805	19,42465	1,37353
roguelike	0	0	0	6,995	6,310703	0,446234
seaquest	0,88	0,324962	0,022978	2724,535	2174,111	153,7329
survivezombies	0,505	0,499975	0,035354	3,12	3,490788	0,246836
waitforbreakfast	0,75	0,433013	0,030619	0,75	0,433013	0,030619

Appendix C2

Detailed results in each game achieved by *ShiftB* controller.

ShiftB						
Game name	Mean wins	Wins standard deviation	Wins standard error	Mean scores	Scores standard deviation	Scores standard error
aliens	1	0	0	64,485	13,49814	0,954463
bait	0,045	0,207304	0,014659	6,23	8,86268	0,626686
butterflies	0,925	0,263391	0,018625	30,8	15,06386	1,065176
camelRace	0,035	0,18378	0,012995	-0,765	0,499775	0,035339
chase	0,03	0,170587	0,012062	2,79	2,205879	0,155979
chopper	0,99	0,099499	0,007036	17,385	2,914923	0,206116
crossfire	0,05	0,217945	0,015411	0,085	1,190704	0,084195
digdug	0	0	0	16,655	12,06922	0,853422
escape	0,2	0,4	0,028284	0,145	0,48371	0,034203
hungrybirds	0,27	0,443959	0,031393	27,4	44,33103	3,134677
infection	0,985	0,121552	0,008595	14,89	8,575424	0,606374
intersection	1	0	0	1,09	0,895489	0,063321
lemmings	0	0	0	-0,505	0,894413	0,063245
missilecommand	0,64	0,48	0,033941	4,61	5,088015	0,359777
modality	0,255	0,435861	0,03082	0,255	0,435861	0,03082
plaqueattack	0,92	0,271293	0,019183	50,665	21,51099	1,521057
roguelike	0	0	0	7,13	6,567579	0,464398
seaquest	0,755	0,430087	0,030412	2187,915	1931,65	136,5883
survivezombies	0,415	0,492722	0,034841	2,7	3,559494	0,251694
waitforbreakfast	0,67	0,470213	0,033249	0,67	0,470213	0,033249

Appendix C3

Detailed results in each game achieved by *StatTree* controller.

StatTree						
Game name	Mean wins	Wins standard deviation	Wins standard error	Mean scores	Scores standard deviation	Scores standard error
aliens	1	0	0	63,49	13,0073	0,919755
bait	0,09	0,286182	0,020236	3,94	4,268067	0,301798
butterflies	0,98	0,14	0,009899	31,16	15,30276	1,082068
camelRace	0,045	0,207304	0,014659	-0,755	0,524381	0,037079
chase	0,05	0,217945	0,015411	2,77	2,144085	0,15161
chopper	0,975	0,156125	0,01104	17,525	3,517012	0,24869
crossfire	0,05	0,217945	0,015411	0,21	1,116199	0,078927
digdug	0	0	0	15,815	11,52305	0,814803
escape	0,175	0,379967	0,026868	0,165	0,397209	0,028087
hungrybirds	0,33	0,470213	0,033249	33	47,02127	3,324906
infection	0,995	0,070534	0,004987	17,08	9,733632	0,688272
intersection	1	0	0	1,41	2,114687	0,149531
lemmings	0	0	0	-0,435	0,869353	0,061473
missilecommand	0,74	0,438634	0,031016	5,74	4,7236	0,334009
modality	0,265	0,441333	0,031207	0,265	0,441333	0,031207
plaqueattack	0,965	0,18378	0,012995	53,31	18,76417	1,326827
roguelike	0	0	0	5,75	5,943694	0,420283
seaquest	0,87	0,336303	0,02378	2884,71	2286,489	161,6792
survivezombies	0,455	0,497971	0,035212	3,055	3,627944	0,256534
waitforbreakfast	0,69	0,462493	0,032703	0,69	0,462493	0,032703

Appendix C4

Detailed results in each game achieved by *Vanilla* controller.

VanillaRHEA						
Game name	Mean wins	Wins standard deviation	Wins standard error	Mean scores	Scores standard deviation	Scores standard error
aliens	1	0	0	62,14	12,68032	0,896634
bait	0,07	0,255147	0,018042	6,35	8,602761	0,608307
butterflies	0,88	0,324962	0,022978	30,96	15,03989	1,063481
camelRace	0,03	0,170587	0,012062	-0,77	0,486929	0,034431
chase	0,035	0,18378	0,012995	2,43	2,2924	0,162097
chopper	0,98	0,14	0,009899	15,48	4,292971	0,303559
crossfire	0,06	0,237487	0,016793	-0,11	1,381268	0,09767
digdug	0	0	0	14,32	10,18517	0,7202
escape	0,315	0,464516	0,032846	0,23	0,589152	0,041659
hungrybirds	0,285	0,451414	0,03192	29,5	44,94163	3,177853
infection	0,955	0,207304	0,014659	13,48	8,012465	0,566567
intersection	1	0	0	1,465	1,915405	0,13544
lemmings	0	0	0	-0,51	1,029514	0,072798
missilecommand	0,545	0,497971	0,035212	3,88	5,007554	0,354088
modality	0,26	0,438634	0,031016	0,26	0,438634	0,031016
plaqueattack	0,945	0,22798	0,016121	53,105	21,17532	1,497321
roguelike	0	0	0	5,235	6,021609	0,425792
seaquest	0,88	0,324962	0,022978	2394,57	1882,27	133,0966
survivezombies	0,475	0,499375	0,035311	3,225	3,672108	0,259657
waitforbreakfast	0,64	0,48	0,033941	0,64	0,48	0,033941

Appendix D

Repository with source code and experiment results: https://gitlab.cs.nuim.ie/p200123/cs646a