Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Evolutionary Learning and Search-Based Content Generation in Computer Games

Tesi di dottorato di:
Luigi Cardamone
738810

Relatore:
Prof. Pier Luca Lanzi
Tutore:
Prof. Fabrizio Ferrandi
Coordinatore del programma di dottorato:
Prof. Carlo Ettore Fiorini

XXIV - 2011

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Evolutionary Learning and Search-Based Content Generation in Computer Games

Doctoral Dissertation of:
Luigi Cardamone
738810

Advisor:
Prof. Pier Luca Lanzi
Tutor:
Prof. Fabrizio Ferrandi
Supervisor of the Doctoral Program:
Prof. Carlo Ettore Fiorini

XXIV - 2011

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 I 20133 — Milano

To Valentina and to my family.

Abstract

Modern computer games achieved an impressive level of realism featuring advanced 3D graphics and sophisticated physics engine able to simulate complex game environments. However, the increasing complexity of the game environment poses many challenges for developing the Artificial Intelligence which controls the non-player characters. In many games, the non-player characters show very simple and predictable behaviors which are not able to evolve over time or to adapt to the user's gameplay. Another problem, caused from the complexity of the game environment, is the generation of a large quantity of game content, e.g., levels, maps, racing tracks, etc. In most complex computers games, the content generation can require the work of several designers and can last even for months. Thus, there is the necessity of techniques that can automate or support some aspects of the game development process.

In this context, Computational Intelligence can be a valid approach to deal with some of the issues in computer games. Computational Intelligence is a field of the Artificial Intelligence which includes techniques and mathematical models which are inspired by nature, such as Evolutionary Algorithms and Neural Networks. In the recent years, the number of scientific works involving Computational Intelligence and Game is quickly growing and suggests that this is a promising research direction.

In this thesis, we focus on the application of Computational Intelligence, and in particular Evolutionary Algorithms, to computer games. In the first part, we investigate different learning paradigms which can support the development of non-player characters. In particular, we investigate on-line learning, i.e., a learning process applied during the game to generate sophisticated non-player characters that can evolve over time. In the second part, we focus on game content generation. We investigate several representations and different approaches for evaluating the game content. In our experimental analysis we take into account different computer games: racing games, shooters and platform games.

The results show that all the proposed methods are promising and Computational Intelligence represents an effective approach to support the game development process.

Sommario

I videogiochi di ultima generazione hanno raggiunto notevoli livelli di realismo. Questo risultato è stato possibile grazie agli elevati progressi ottenuti nella realizzazione di motori grafici tridimensionali e simulatori della fisica sempre più sofisticati che permettono di rappresentare degli ambienti di gioco con un elevato grado di complessità.

Tuttavia, la crescente complessità degli ambienti di gioco ha reso sempre più difficile la programmazione dell'intelligenza artificiale dei giocatori controllati dal computer, spesso indicati come *non-player characters (NPCs).* Da una parte vi è la necessità di avere tecniche in grado di automatizzare lo sviluppo dei comportamenti di gioco dei non-player characters. Dall'altra si rendono necessari dei meccanismi per realizzare non-player characters più sofisticati, in grado di apprendere dall'esperienza ed adattarsi a nuove condizioni di gioco.

Un altra problematica, relativa alla crescente complessità degli ambienti di gioco, riguarda la generazione del contenuto di gioco. Per contenuto si intendono tutti quegli elementi che non fanno strettamente parte del motore di gioco, ma ne estendono la giocabilità e la durata: per esempio i livelli di gioco, le mappe, le armi, i personaggi, le piste da corsa, ecc. Nei videogiochi più complessi, la generazione delle mappe di gioco è una attività costosa che può richiedere diversi mesi di lavoro. Da qui la necessità di avere degli strumenti in grado di automatizzare parzialmente o totalmente la generazione del contenuto di gioco.

In questo contesto, la *Computational Intelligence* può rappresentare un approccio interessante per affrontare alcune delle problematiche presenti nell'ambito dei videogiochi. La Computational Intelligence è un area di ricerca dell'Intelligenza Artificiale, che comprende tecniche e modelli matematici ispirati alla natura, come *Algoritmi Evolutivi*, *Reti Neurali* e *Apprendimento per Rinforzo.* Negli ultimi anni, la Computational Intelligence è stata applicati ai giochi e a i videogiochi in molti lavori di ricerca. Il numero di pubblicazioni scientifiche e i risultati ottenuti in questo ambito suggeriscono che l'applicazione della Computational Intelligence nei videogiochi rappresenta una direzione di ricerca promettente.

In questa tesi ci proponiamo di affrontare alcune problematiche presenti nell'ambito dei videogiochi, usando tecniche di Computational Intelligence ed in particolare Algoritmi Evolutivi. Prenderemo in esame due problematiche dei videogiochi moderni: l'apprendimento dei comportamenti di gioco dei non-player characters e la generazione del contenuto di gioco. Per quanto riguarda l'apprendimento, verranno studiati due aspetti: da una parte tecniche per automatizzare l'apprendimento dei comportamenti di gioco; dall'altra tecniche per creare dei non-player characters sofisticati in grado di apprendere on-line, cioè durante il gioco stesso. Per quanto riguarda la generazione del contenuto di gioco studieremo diversi approcci: generazione basata su cifre di merito teoriche e generazione basata sulle preferenze dell'utente. I vari approcci verranno studiati in videogiochi che richiedono contenuto di gioco molto diverso tra loro: piste per un gioco di guida, mappe per uno sparatutto in prima persona e livelli per un gioco platform. I risultati ottenuti mostrano che gli approcci proposti hanno un buon potenziale e che la Computational Intelligence rappresenta una metodologia efficace per supportare diversi aspetti dello sviluppo di videogiochi.

Acknowledgments

My first thanks is to my advisor, Prof. Pier Luca Lanzi, and to Ing. Daniele Loiacono for guiding my efforts during this Ph.D.
I would also like to thank Prof. Georgios Yannakakis, Prof. Julian Togelius, and all the Game AI group which gave me the opportunity to work with them during my research period at the IT University of Copenhagen.
Then, I would like to thank all my colleagues which worked with me at Politecnico di Milano: Marco Lattuada, Chiara Sandionigi, Antonio Miele, Paolo Grassi, Christian Pilato, Enrique Onieva and Antonino Tumeo.
Thanks also to the students of the Politecnico di Milano for their contribution to my research work. Among them, I would like to mention Alessandro Bardelli, Marco Colombo, Luca Galli, Matteo Botta, Vincenzo Gautieri, Antonio Caiazzo and Stefano Suardi.
I would like to thank my family for their support in every moment.
Finally, a big thanks to Valentina for her love and for making me smile whenever I need it.

Milan, 16^{th} January, 2012.

Luigi Cardamone

Contents

List of Figures

List of Tables

1. Introduction

It has been almost 60 years since 1952, when one of the first videogames, *Tennis for two*, was played using an oscilloscope. During these years, videogames have been going through a constant and rapid evolution towards realism and playability. Modern computer games feature realistic 3D graphics and sophisticated physics engines which reproduce real or fictional environments with an impressive level of detail. The majority of new titles are multi-player oriented and are played on the internet against or in cooperation with other players.

However, the level of complexity achieved in computer games poses several challenges for developing the *Artificial Intelligence (AI)* of the *non-player characters (NPCs).* Usually, the non-player characters present very simple and predictable behaviors. Once the player discovers a weakness in the game AI, this weakness can be exploited indefinitely without any possibility that the behavior will change. In general, modern computer games need an AI which is adaptive and able to show human-like behaviors. The AI should be able to adapt both to new environments (e.g. new maps not available in the original game) and to the player to improve the user's fun or to dynamically balance the challenge.

Another problem of the increasing complexity in computer games is related to the *game content*, e.g., maps, levels, racing tracks available within a game. Game content represents an important element for the game longevity, i.e., the time required to complete the game. Unfortunately, in modern games levels and maps have a huge amount of detail and can require many months of work by several designers to generate the few levels that will be shipped with the game.

In this context, Computational Intelligence might represent an interesting approach for dealing with some of the open problems in computer games. Computational Intelligence is a field of Artificial Intelligence which includes techniques and mathematical models which are inspired by nature, like *Evolutionary Algorithms*, *Neural Networks* and *Reinforcement Learning.* Computational Intelligence has already been applied to computer games in several academic works. The number of scientific publications on this topic is fast growing and the results show that Computational Intelligence can be a valid tool to support the development of computer games.

A large number of research works focus on learning behaviors for non-player characters. Almost all the early works in this area focused on board games like *Go* [74, 70, 5, 16], *Checkers* [71, 73, 32], *Backgammon* [91, 90, 63], *Chess* [92, 40, 34]. More recently, the application domain has broadened to include many game types: arcade games [57, 48], first person shooters [15, 33, 64], racing games [6, 95], real time strategies [25, 53]. Overall, many games have been used and a variety of techniques have been investigated: evolutionary computation, fuzzy logic, neural networks and reinforcement learning. However, very few works focus on (i) on-line learning, i.e., a learning process that is performed *in-game*; (ii) transfer learning, i.e., a set of techniques for exploiting the knowledge of a source domain to facilitate learning in a target domain.

Computational intelligence for game content generation, is a more recent research topic and some of the early examples [37, 50, 97] were only published as recently as 2006. Thus, many research directions are still open or explored by only a few works, such as, for example, first person shooters. Among the others, an interesting research direction is the generation of personalized game content which uses the feedback of the user to generate content which can maximize his/her preferences.

1.1. Thesis Objectives

In this thesis, we focus on the application of Computational Intelligence to computer games. In particular we focus on Evolutionary Algorithms and Neural Networks. We take into account two main topics in the design of computer games: the development of non-player characters and the generation of game content.

The goal of the first part of the thesis, is to investigate different learning paradigms which can support the development of non-player characters. In particular we investigate (i) off-line learning, a learning process applied out-of-game to generate strong and competitive non-player characters with a static behavior; (ii) on-line learning, a learning process applied in-game to generate non-player characters which can evolve over time by learning new behaviors from scratch or by adapting an existing one; (iii) transfer learning, a set of techniques for transferring behaviors across different games, by exploiting the knowledge of a source domain in order to accelerate the learning in a target domain.

The goal of the second part of the thesis, is to investigate search-based procedural content generation to automatically generate content in different game types. We investigate two key aspects: the representation of the game content and the fitness function. In particular we consider two types of fitness functions: (i) theory-driven fitness function, which comes from a theoretical study of the problem and aims to generate content which presents some interesting characteristics like variety and diversity; (ii) user-driven fitness function, which aims to generate personalized game content that maximizes user preferences. Finally, since content representation is game dependent, we propose and investigate representations in different game types: racing games, first person shooters and platform games.

1.2. Outline

This thesis is organized as follows.

In Chapter 2, we present the field of computer games and the field of Computational Intelligence. We analyze some open problems in modern computer games and we discuss why they represent an interesting domain of application for Computational Intelligence.

In Chapter 3, we review the main approaches and the most important related works which involve the application of Evolutionary Algorithms to computer games. In particular, in the first part we describe which elements should be considered when applying evolutionary computation to evolve NPCs behaviors. In the second part, we describe the main concepts behind search-based procedural content generation.

In Chapter 4, we present the genre of racing games and we explain why it is appealing for research purposes. Then, we present the major research works which involve racing games and the academic competitions organized in this field. Finally we introduce the game platform used for the experiments of this thesis.

In Chapter 5, we study the off-line learning of NPCs behaviors. In particular we consider the problem of evolving a competitive driver for a racing game. The evolved behavior is finally evaluated in a complex racing task against other drivers.

In Chapter 6, we introduce the concept of on-line learning and the additional challenges related. We present our approach which is based on on-line neuroevolution combined with fine grained evaluation episodes.

The proposed approach is validated in a driving task and we study both the learning from scratch and the adaptation of an existing policy.

In Chapter 7, we apply transfer learning to transfer driving behaviors across two racing games with different engines and game dynamics. In particular we analyze three methods for transfer learning: direct copy of the behavior; transfer of the learning process; direct copy of the behavior combined with an adaptation process.

In Chapter 8, we apply search-based procedural content generation to evolve racing tracks. In particular, we study two different approaches. The first one consists of evolving tracks with a good amount of variety and challenges. The second one aims to evolve tracks which maximize the players' preferences through a framework which implements an interactive genetic algorithm.

In Chapter 9, we apply search-based procedural content generation to evolve multi-player maps for a First Person Shooter. We introduce four representations for single-floor maps and we design a fitness function for evolving maps with interesting game-play. Finally, we show how one of the representation can be extended to evolve maps with more than one floor.

In Chapter 10, we exploit a user preference model to generate levels for a platform game. In particular, we study the problem of adapting the model on-line to a given user. The content generation is studied using both a static model and a dynamically adapted model. The approach is validated through an experimental study involving human players.

In Chapter 11, we finally give a short balance of the work done and we discuss the possible future research directions.

1.3. Related Publications

Most of the results presented in this thesis also appeared in the following publications:

- Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Learning to drive in the open racing car simulator using online neuroevolution. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(3):176 –190, sep. 2010.

- Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Online neuroevolution applied to the open racing car simulator. In

Evolutionary Computation, 2009. CEC '09. IEEE Congress on, pages 2622–2629, May 2009.

- Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1179–1186, New York, NY, USA, 2009. ACM.

- D. Loiacono, P.L. Lanzi, J. Togelius, E. Onieva, D.A. Pelta, M.V. Butz, T.D. Lonneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, and J. Quadflieg. The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131 –147, jun. 2010.

- Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 395–402, New York, NY, USA, 2011. ACM.

- Luigi Cardamone, Georgios N. Yannakakis, Julian Togelius, and Pier Luca Lanzi. Evolving interesting maps for a first person shooter. In *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part I*, EvoApplications'11, pages 63–72, Berlin, Heidelberg, 2011. Springer-Verlag.

- Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Automatic track generation for high-end racing games using evolutionary computation. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):245 –259, sept. 2011.

- Luigi Cardamone, Antonio Caiazzo, Daniele Loiacono, and Pier Luca Lanzi. Transfer of Driving Behaviors Across Different Racing Games. In *Computational Intelligence and Games (CIG), 2011 IEEE Symposium on*, pages 227 –234, aug. 2011.

Part I.

Background

2. Computational Intelligence and Games

In this chapter, we introduce the field of videogames and we overview its evolution from the primitive prototypes to the modern titles of today. Then, we briefly introduce the field of computational intelligence. Finally, we analyze the motivations for applying computational intelligence to games and we give an overview of the research community working in this field.

2.1. History of Videogames

The origin of videogames dates back to 1947 when Thomas Goldsmith and Estle Ray Mann requested a patent for an invention called "cathode ray tube amusement device". This was a machine in which a person could use knobs and buttons to manipulate a cathode ray tube beam to simulate firing at airplane targets.

In 1952, Alexander Douglas made a game similar to *Tic-Tac-Toe* at the University of Cambridge. It was the first computer game to use a digital graphical display. It was designed for EDSAC computer, the world's first stored-program computer, and used a rotary telephone controller for game control.

In 1958, William Higinbotham made an interactive computer game named *Tennis for Two* for the Brookhaven National Laboratory's annual visitor's day. The game was developed using an oscilloscope and an analog computer and was played with two box-shaped controllers, both equipped with a knob for trajectory and a button for hitting the ball.

In 1961, a group of students at MIT, including Steve Russell, programmed a game titled *Spacewar!* on the DEC PDP-1, a new computer at the time. The game pitted two human players against each other, each controlling a spacecraft capable of firing missiles. The game was eventually distributed with new DEC computers and traded throughout the primitive computer networks. Spacewar! is credited as the first influential computer game.

In 1971, Nolan Bushnell, with help from the programmer Ted Dabney and arcade-machine manufacturer Nutting Associates, produced the first mass-produced coin-operated videogame. Its name was *Computer Space*. The genial idea was to transfer Spacewar! computer-programmed gameplay into easily available logic circuits, which contributed to reduce the cost of the product.

In 1972, Bushnell and Dabney founded their own company named Atari and they produced the iconic *Pong*. Problems with hardware producers, led Atari to build its own machine for Pong: within twelve months they sold around 8500 Pong units. In the same year, it was developed the first console, i. e., a TV-based system for home gaming. Its name was Magnavox Odyssey, and its first version, with twelve hard-coded games, sold around 100,000 units.

In 1976, Fairchild Camera & Instrument releases its Video Entertainment System (later renamed Channel F), the first home game console to use cartridges to load games, rather than hard-wiring them into the hardware. This system was copied by Atari, that in 1977 released its Video Computer System (VCS). After a slow start, Atari sold a million of consoles only in the 1979. In the same year, Atari developed its best selling arcade game: *Asteroids*. Deployed over 50,000 machines worldwide, Asteroids, became one of the first videogames to be copyrighted in the US.

In 1980, the Atari VCS became the first console to receive a licensed arcade game and hence the first to claim exclusive content. That game was the famous *Space Invaders* and was a porting of the original arcade game produced by Taito two years before. In the same year, Japan's Namco produced the iconic *Pac-man*. This arcade games quickly achieved an impressive popularity and became the second game in history to cause a coin shortage in japan.

In 1981, Nintendo developed *Donkey Kong*, an arcade game which quickly became a best seller. The game was a single-screen platform where the main character called Jumpman race to save his girlfriend Pauline from a crazed monkey. Jumpman is later named Mario, becoming the most famous character of the Nintendo.

In the early eighties, the console markets has a sudden slowdown mainly due to the strong the competition involving many game companies. In the same years, personal computers start to spread and in 1983 the *Commodore 64* was released to the market. It quickly became the best-selling single personal computer model of all time. The rise of personal computers caused a big explosions in the number of game titles published.

In 1983, consoles strikes back with Nintendo which released to the Asian market the 8-bit *Famicom* (short for Family Computer). By the end of 1984, the Famicon has sold around 3 millions of units. In 1985, the same console was launched in the United States with the name *Nintendo Entertainment System* (NES): it was a hit in a limited market release. Most of the success of this new console was due to new control policies on game released. Nintendo maintained a strict control over the game released, focusing more on the quality of the published titles than on the number. Nintendo introduced a system of licensing game produced by third part developer: the game developed had to be exclusive for Nintendo for two years. With this licensing system, to play a Nintendo game the player were forced to buy a Nintendo machine. One of the best selling game of NES (and one of the most influential game of all the time) was *Super Mario Bros* released in 1985.

In 1989, the *Game Boy*, released in Japan and the US, was the first handheld gaming device to use changeable cartridges, and remains the biggest selling handheld to date. Crucial for its success, was Nintendo's legal victory over Atari for the rights to publish the most addictive game of history: *Tetris*. This game, originally designed by from the Russian Alexei Pajitnov in 1985, sold over 33 millions of copies on the Game Boy.

In 1991, Nintendo released its 16-bit Super Nintendo Entertainment System (SNES) with some new games like Super Mario World, Legend of Zelda and Donkey Kong Country. In the meanwhile, the solid leadership of Nintendo, had been eroded by the more powerful Sega Mega Drive. The top game of the Mega Drive was the Sonic the Hedgehog, a spiky-haired ball that represented the antidote to Nintendo's Mario. The fast running Sonic showed off all the speed and power of the Mega Drive.

In 1994, Sega Saturn and Sony PlayStation are launched in Japan. The year after the are both launched in the US. Initially, the Saturn was blessed with a much great success than the Play Station. However, in less than two years, the ability of Sony to gather a wide number of game titles, like Tekken, Mortal Kombat, Formula One and Tomb Raider, brought to Play Station an impressive success with respect to all the other consoles.

2.2. Videogames Today

After more that 50 years from the early prototypes in this field, the world of videogames has been following a constant and rapid evolution in every aspect: from the graphical visualization to the interface used for playing. Nowadays, in the videogames field there are a variety of platforms: personal computers, consoles, pocket consoles, tablet and mobile phones. Even if consoles and personal computers are the most popular platforms, casual gaming is having a strong diffusion on pocket consoles (e.g., *Nintendo DS* and *Play Station Portable*) and on all kinds of mobile phones. For what concern the realism and the graphical visualization, modern computer games reached an impressive level. Many games have a standout 3D visualization and every aspect of the environment is represented with an high level of detail: e.g. water effects, smoke effects, explosions, dynamic lighting. The realism is also achieved from an accurate simulation of the physics of every object in the game scene.

The level of realism has pushed also to create specific controller interfaces: e.g. the racing wheel and the pedals for a racing game, the guitar of a musical game, the cloche for a flight simulator. In some cases, the physical interface disappears in order to offer a new gaming experience: it is the case of *Kinect XBox* where the player interacts with the game through a camera which detects the gestures and the movements of the body. In the case of the console *Nintendo Wii*, the player performs physical movements to interact with a wireless controller which is used to represent a variety of objects: e.g. a tennis racket, a sword, a gun.

Another important aspect of modern videogames is the availability of a multi-player mode and in particular the possibility for on-line gaming. In some games, the on-line play is the only available game mode. A particular class of these games is represented by Massively Multiplayer Online Role-Playing Games (MMORPG) in which a very large number of players interact within a virtual game world. Internet, has also allowed the spread of a particular class of casual games which are played within the web browser, and typically are connected to a social network. These games achieved an unexpected success due to their simplicity and through several social interaction mechanisms: e.g. break the score of friends. One of the most famous example is Farmville[1] from Zynga, which achieved over 80 million of users worldwide.

From an economic point of view, the videogame industry is growing with impressive performance. Currently, it is one of the leading forms of

[1]http://www.zynga.com/

entertainment and its revenues has surpassed the movie and the music industries. [2] In the last ten years, the videogames industry has more than doubled the yearly revenues: 7 billions of dollars in 1999, 25 billions of dollars in 2010.[3] [4]

2.3. Game Genres

During the years, thousands and thousands of games have been published. Even if each game has its unique characteristics, it is possible to identify several categories. Many different taxonomies have been proposed, but no one can be considered complete since there are many game titles which embraces more than one genre. In this section, we provide a simple taxonomy which aims to give an overview of the main genres in the field of videogames. In particular, we identify five main categories: action games, adventures, role-playing, simulations and strategy games. In the remainder of this section, we provide a quick overview for each game genre considered.

Action

In action games, the player typically controls a single character which has to overcame a series of challenge or fight against different opponents. This kind of games usually require quick reflexes and accuracy. A popular class of action games is represented from arcade games. Arcade refers to all that titles which have the similar style of the games typically played on arcade-machines. Some historically famous examples are Pac-man, Pong, Space Invaders and Asteroids. Another class, of action games is represented by the so called *Platforms*. This kind of games are characterized from side-scrolling levels. The most well know examples are Super Mario and Sonic.

A very important class of action games is represented from *Shooters* in which the player uses different weapons to kill the opponents which inhabit the game world. In particular, the most popular shooters are the so called First Person Shooters (FPS) in which the player sees the game world from a first person perspective. Some examples of the most

[2]http://vgsales.wikia.com/wiki/Video_game_industry
[3]http://www.newzoo.com/ENG/1575-Total_Consumer_Spend_2010.html
[4]http://www.theesa.com/gamesindailylife/economy.asp

popular FPS are the *Quake* series[5], *Unreal Tournament*[6], *Call of Duty*[7], *Halo*[8], *Half Life*[9].

Adventures

In adventure games the player assumes the role of protagonist in an interactive story driven by exploration and puzzle-solving. The genre's focus on story allows it to draw heavily from other narrative-based media such as literature and film. The first games of this type, date back to seventies, and were completely based on a textual representation while the user interaction was allowed by typing simple sentences. Some of earliest examples are *Colossal Cave* [31] and *Zork*.[10]

In its most modern form, adventure games have a graphical visualization and are usually called point-and-click games since the interaction is mainly based on the mouse. Some famous examples are the *Myst* series[11], *Day of the Tentacle*[12], and *Monkey Island*.[12]

Role-playing

Role-playing games (RPGs) originated from pen-and-paper role-playing games such as Dungeons and Dragons, using much of the same terminology, settings and game mechanics. The player controls one or more game characters, and achieves victory by completing a series of quests and reaching the conclusion of a central storyline. Players explore a game world, while solving puzzles and engaging in tactical combat. A key feature of the genre is that characters grow in power and abilities, and characters are typically designed by the player. Famous examples of role-playing games are *Neverwinter Nights*[13] and the *Baldur's Gate* series.[13] An important class of these games is represented by Massively Multiplayer Online Role-Playing Games (MMORPG) in which a huge number of players interact within a virtual game world. The most know example is *World of Warcraft*.[14]

[5]www.idsoftware.com
[6]www.epicgames.com
[7]www.activision.com
[8]www.microsoft.com/games/
[9]www.valvesoftware.com
[10]www.infocom-if.org
[11]www.myst.com
[12]www.lucasarts.com
[13]www.bioware.com
[14]www.blizzard.com

Simulation

This category includes all games that try to reproduce some real activity with the most possible level of accuracy. Some examples are *SimCity*[15], which simulate many aspects of the management of a city, and *The Sims*[15], which simulate the life and the social interactions of a human character. In this category, an important class is represented by the games which simulate the control of vehicles. Some examples are the simulators for cars, motorbikes, trains, flights, spaceships and speedboats.

Strategy and Real-Time Strategy

In strategy games, the gameplay require careful and skillful thinking and planning in order to achieve victory. In strategy games, the player usually manages limited resources and units in order to expand its power and defeat the opponent. It is possible to see that strategy games are very close to some board games like Chess or Risiko. A strategy game can be turn-based, like the very popular game *Civilization*, or real-time, like *Age of Empires*[16], *Startcraft*[17] and *Warcraft*[17].

Other Genres

Among the other games genres we reports puzzle games and sport games. Puzzle games require the player to solve logic puzzles. Maybe the most popular example is Tetris. Sport games are games which represent and actual sport like soccer, basket, tennis etc. Soccer games like the *Fifa*[18] or the *Pro Evolution Soccer* series[19] are undoubtedly among the most played titles in the category of sport games.

Finally, many popular board games and card games have a computer version. Some examples are Chess, Checkers, Othello, Backgammon, Go, etc. In this games it is usually possible to play against the computer AI or on-line against other players.

[15]www.ea.com
[16]www.microsoft.com/games/
[17]www.blizzard.com
[18]www.ea.com
[19]www.konami.com

2.4. Computational Intelligence and Games

Computational Intelligence is a field of the artificial intelligence devoted to the study of algorithms and techniques for learning and optimization that are somehow inspired by nature. Here we find evolutionary computation, which uses Darwinian survival of the fittest for solving optimization problems; neural networks, where various principles from neurobiology are used for machine learning; reinforcement learning, which borrows heavily from behaviourist psychology; fuzzy logic which is inspired from principles of the human reasoning.

In this section, we firstly present which are the motivations for applying computation intelligence techniques to computer games. Finally, we present the main conferences and journals in the field.

2.4.1. Motivations

Computer games did giant leaps in many aspects like the graphical visualization and the realism of the simulated environment. Although, the artificial intelligence of the non-player characters (NPCs) is not very sophisticated. Many NPCs present very simple and predictable behaviors. Usually, when the player discover a weakness of a NPC, this weakness can be exploited indefinitely without no hope that the behavior can change. This aspect affect the believability of the game and cause the player to loose interest and challenge. In this context, on-line learning, can be applied in order to develop a better AI.

In some games, in order to build the strongest AI, the developers use cheating allowing the NPC to perform non-realistic actions. Again, this is a problem since the NPC is not believable and may cause frustration when the player try to imitate its actions. To overcame this problem, a promising approach it to apply optimization in order to evolve a stronger AI without using cheating. Computational Intelligence can be also applied to evolve a non-player character from scratch. This can be very important when game developers need to quickly generate the Game AI for testing purpose.

Another common problem is the design and the generation of game content, i.e. levels, maps, racing tracks, etc. In many games, some levels are so complex that require the work of many designers and represent a very time consuming activity. In this context, computational intelligence and in particular evolutionary computation can be a promising approach for content generation.

From the other side, computer games are an excellent test bed for testing computational intelligence techniques: they allows to compare different techniques on complex simulated environments. Many simulators developed for research activities are too simple, mainly because a realistic simulator is costly and hard to realize. Often these simulators are not enough detailed to solve some difficult real-world problems. So, a great opportunity is to use last generation games which present a realistic environment made of thousands of objects of different materials, simulate many kind of forces like friction, viscosity, gravity, aerodynamics and give the possibility to the agents to interact in many complex ways with almost everything.

2.4.2. Conferences and Journals

In the recent years, a growing body of researchers have shown a great interest in the field of computational intelligence and games. This result is confirmed both from the number of papers published in the topic and from the conferences dedicated to this field.

Among the most important conferences almost completely devoted to the topic we have the IEEE Symposium on Computational Intelligence and Games (CIG) and Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE). Both conferences started in 2005 and are held every year. Among the other conferences, two of the major international conferences on evolutionary computation dedicate a full track to evolutionary algorithms applied to computer games: the ACM conference on Genetic and evolutionary computation (GECCO) and the IEEE Congress on Evolutionary Computation (CEC). More recently, has been founded a new journal completely dedicated to this topic: IEEE Transactions on Computational Intelligence and AI in Games (TCIAG). To analyze the trend of the publications about computational intelligence and games we performed a wide bibliographical research which took into account the main conferences and papers in this field. In figure 2.1 we show the papers published in the last 10 years (based on our bibliographical research). It is possible to see that there is a constant increment in the number of papers. In particular, the number of papers has a big increment from the 2005, year in which both the conference CIG and AIIDE were held for the first time.

2.5. Summary

In this chapter, we overviewed the field of computational intelligence and games. We presented a brief history of videogames, from the early

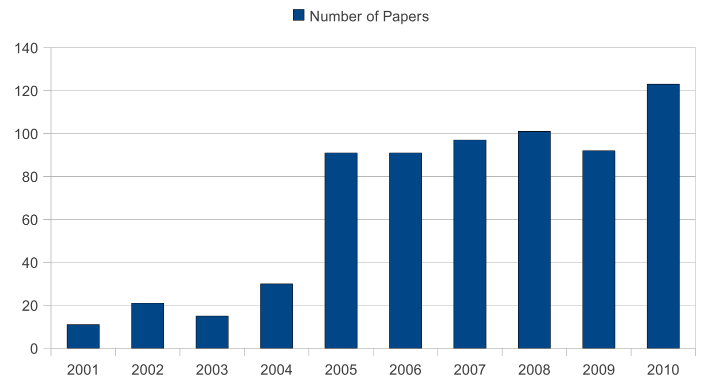


Figure 2.1.: Papers published in the field of computation intelligence and games

fifties until today, and we described the main game genres. Finally, we presented why games are an interesting application domain for computational intelligence and that a growing body of researchers is working in this field.

3. Evolutionary Computation in Games

In this chapter, we give an overview of the application of evolutionary computation (EC) to games. In the first part, we describe EC for evolving non-player characters (NPCs) in videogames and we discuss the most important works in the area. In the second part, we define game content and we introduce the area of Search-Based Procedural Content Generation (SBPCG) together with the most relevant research works.

3.1. Evolving Non-player Character Behaviors in Games

Evolutionary computation has been widely applied to computer games, mainly for learning the behaviors of non-player characters (NPCs). The overall behavior of the NPC is usually decomposed in a multi-level hierarchy of simpler behaviors [45]. In this hierarchy, each behavior is responsible for accomplish a specific task. For example, the NPC of a racing game can be decomposed in several behaviors like driving, overtaking and recovery. The decomposition in simpler behaviors has the primary goal of reducing the initial problem to smaller sub-problems where the use of evolutionary computation can be more effective. Once a target behavior has been identified, the application of EC requires two elements: (i) a representation for that behavior; (ii) a fitness function which can score the ability in solving the given task.

3.1.1. Representation

A behavior can be encoded using several representations. When evolutionary computation is used to optimize the performance of an existing NPC, the representation is typically a vector of parameters. In this case, the most important parameters are identified from the code of the NPC, and a genetic algorithm is used to tune the behavior for maximizing the performance. In [15] this approach was used to tune the behavior of a First Person Shooter Bot. The same approach was applied by Butz et al. [6] to optimize the performance of a driver in a racing game and in [57] to optimize a scripted agent of X-pilot.

A very common representation is based on neural networks. In this case, the behavior is encoded in the neural network, which receive as input some information from the game environment and outputs some actions. An evolutionary algorithm can be applied to modify both the weights or the structure of the network. This combination, usually called Neuroevolution, has been applied in many research works. In [56, 58], the authors use neural networks for representing different types of behavior in X-pilot. In [95, 98], Togelius et al. evolve a driving behavior for a racing game and investigated neural networks with several schema of sensory information (e.g., first person based, third person based, etc.). In [33] neural networks were used to evolve a behavior for navigating a map in Unreal Tournament. In [48], Lucas evolves neural networks to develop an agent that can play Ms. Pac-Man. In this work, the neural networks do not represent a behavior, but instead are used as function which evaluate the future locations of the maze. The agent will select the action which leads to the location with the highest score.

In some works, a behavior was represented as a computer program which is evolved using Genetic Programming. This approach was used in [1] and [18] to evolve a driving behavior for a racing game.

A behavior can be also represented as a list of rules. Rules are usually expressed as pair of states and actions. When the agent conditions matches a state, the respective action is applied. The complete set of rules is called rule-base. This approach was applied again for evolving a driving controller in [62] and to develop a competitive agent for playing Quake [64].

3.1.2. Fitness Function

In this context the fitness function is usually computed after a simulation in the game. Thus, the behavior to be evaluated is used in the game to solve a specific task: during this simulation, some features are extracted and then are used to compute the fitness function. A fitness based which use an in-game simulation is a critical issue for the application of evolutionary computation: (i) the simulated match need to be completely automatized but in many commercial games this is not possible; (ii) the simulation often requires a great amount of computational resources and a complete run of a genetic algorithm can last for several hours or even days. For this reason, some commercial games are not used in favor of open-source games which can be modified and adapted to perform batch experiments. In particular, in open-source games it is also possible to disable the graphical rendering to perform the experiments in accelerated time.

In many works, the fitness is designed to evolve a non-player character which can solve a given game task with the best performance possible. E.g., achieve the best lap in a driving task [6]; clear the level in shortest time possible [48]; inflict as much damage as possible in a FPS game [64]. However, pushing for best performance is not the only fitness which can be used. In particular, some research works investigated approaches in which the goal is to evolve a NPC that can maximize the player fun or a NPC which can resemble human style of play. Human like behaviors have been evolved in Quake [65, 66] and in racing games [105]. In [114, 113] Yannakakis et al. investigated how is possible to evolve opponents which can improve the player fun.

3.2. Off-line versus On-line Learning

Off-line learning consists in applying an evolutionary algorithm to search for a good policy that can solve a given task. When the evolutionary process is over, the policy with highest fitness can be deployed in the the game. In off-line learning the main goal is to find an individual with the highest fitness possible. It is not important if several individuals of the population has a very low fitness. All the works cited so far use an off-line learning approach.

On-line learning, is meant to be used after the game deploy, i.e., while the game is being played by the final user. While a NPC evolved off-line cannot modify its behavior when the game is running, the goal of on-line learning is to create NPCs that can always modify their behavior and adapt to new conditions of the environment. Such ambitious goal poses two challenging constraints: (i) the learning process should be fast; (ii) the learning process should be as transparent as possible to the final user. A fast learning process in this context means that the learning time should be comparable with the time necessary to play few matches. E.g., if in a racing game a race usually last 5-10 minutes, a reasonable learning time can be 30-60 minutes but not 1000 minutes. The learning process should be transparent in the sense that it should not have a negative impact on the user experience. A poor user experience can happen when the evolutionary learning algorithm selects for evaluation some individiduals with a low fitness. To overcame this problem, an idea is to increase the time dedicated to the evaluation of good individuals and reduce the evaluation time of poor individuals. This idea was applied in [108, 109] by Whiteson et al. which deal with the exploration/exploitation problem using some selection strategies taken from Reinforcement Learning. A different approach was applied in the game NERO [79] by Stanley et al. In NERO, a steady-state version of NEAT, rtNEAT, was applied to evolve in real-time a group of agents. The use of a steady-state algorithm avoids that one population is instantly substituted by another, and the evolution follows a more gradually process. Recently, rtNEAT has been applied also to a real-time strategy game [53] and to evolve a team of ghosts in Pac-Man [110]. In, [68] the authors applied both rtNeat and the selection strategy proposed in [108, 109] to perform on-line learning into a 2D spaceship game.

3.3. Transfer Learning

In the previous sections, we described the evolution of NPC behaviors. An interesting problem, is to study whether the evolved behaviors can be transfered to a another game of the same type. This type of problem belongs to the area known as *Transfer Learning* (TL) [93, 87]. This area studies how the knowledge learned on a *source* task can be exploited to speed up the learning on a new *target* task. Several methods of transfer learning have been specifically devised to be applied in Reinforcement Learning (RL) or Temporal Difference (TD) learning [84]. Such methods either focus on the transfer of value functions (e.g., [86, 88]) or experience instances (e.g., [39]). Only few works in the literature (see [87] for an overview) focused on transferring the learned policies and, thus, can be applied to the methods that perform a search in the policy space (like NEAT [83]). One of the earliest examples of transfer learning in on-line evolution is due to Lanzi [38] who used populations of condition-action-prediction rules (called classifiers) evolved for simple tasks to seed the evolution of solutions in similar problems involving more complex state-action spaces.

In [89], Taylor et al. show that is possible to transfer the knowledge evolved using NEAT, when the target task has either a different state space or a different action space. In particular, they used the population evolved by NEAT in the source task to seed the initial population for the target task. At the best of our knowledge there is no work in the literature which studies transfer learning of policies across similar computer games.

3.4. Evolving Game Content

In this section, we firstly introduce a definition of game content. Then, we present what is procedural content generation and we cite some of the most important examples of computer games which use this technique. Finally, we present search-based procedural content generation and we review the most important related works available in the literature.

3.4.1. Game Content

Accordingly to the definition introduced by Togelius et al. in [102], *game content* means all aspects of a game that affect gameplay but are not non-player character (NPC) behaviour or the game engine itself. This is a broad definition which includes such aspects as terrain, maps, levels, stories, dialogue, quests, characters, rulesets, dynamics and weapons. A more strict definition of content may exclude rules and game dynamics but can also introduce ambiguity (e.g. a weapon which modifies the game dynamics is part of the game content or not?). The definition introduced by Togelius et. al. does not explicitly includes all the aspects which affects the player experience of the game, like sounds, lights, textures, etc. Even if these aspects does not affect the gameplay, they are very important for the success of a videogame and may take a non negligible amount of resources during the development process. Thus, in this thesis we use a wider definition of game content, that includes all aspects of a game that affect the gameplay and the game experience.

The content of a game can be divided in two big category: *interactive* content and *non-interactive* content. With interactive we mean all the content of the game with whom the player can interact when playing the game. E.g. the tracks of a racing game, the maps of a First Person Shooter or the level of a platform game like Mario. The interactive game content directly affects the gameplay and must be correct: e.g. a racing track must be closed or the exit point of a map must be reachable. This is an important constraint to be taken into account when the game content is generated automatically. With non-interactive we mean all the other content which does not allow any interaction with the player: the background of a platform game, the environment around a racing track, and in general all the sounds, the textures and the lights of the game. These aspects are generally used to enhance the level of realism of the game or to influence some emotions like fear, suspense, engagement, etc.

3.4.2. Procedural Content Generation

Procedural content generation (PCG) refers to algorithmic procedures which automatically generate game content. Procedural content generation is usually based on a constructive approach which is an alternative to the use of an explicit representation of the game content. Thus, procedural content generation started with the goal of saving memory space, but soon became a way for generate original and diversified game content.

PCG algorithms can be divided in *non-stocastic* and *stochastic*. In the first case we have a deterministic procedure which generate always the same piece of content: e.g. a procedure which generate a forest that is a grid 4 by 4. In the second case, for each run of the algorithm we will get a different piece of content: e.g. a procedure which generated a forest with randomly placed trees.

A PCG algorithms usually can take one or more parameters which determines the generation process: e.g. for a random forest of trees, possible parameters can be the minimum and the maximum distance between a tree. Usually, if a PCG algorithm is deterministic and takes as input a list of parameters, this algorithm is a *mapping procedure*: in this case the list of parameters represents an implicit compact representation which is then translated into an explicit representation of the actual content.

Procedural content generation can be applied both on-line or off-line. When applied off-line, PCG is a tool which helps the game designer to create a piece of game content. For example, the tool *SpeedTree*[1], can be used to create a forest with a wide variety of vegetation: then this forest can be imported inside a bigger piece of content like a map for an RTS game. When used on-line, the procedural content algorithms are integrated into the game and are used on-the-fly to generate new piece of content. As an example, some RTS game allows to start a new match in a new random map, which contains forests, lands, lakes generated on-the-fly.

Procedural content generation has been widely applied to commercial videogames and the early examples date back to the eighties. One of the early example is *Rogue*[2], an ASCII role-playing game by Michael Toy and Glenn Wichman in which an unlimited number of dungeon maps,

[1]http://www.speedtree.com/
[2]http://en.wikipedia.org/wiki/Rogue_(computer_game)

filled with monsters and treasures, were procedurally generated on-the-fly [37]. In 1984, the space trading game *Elite*[3] by Ian Bell and David Braben (Acornsoft 1984) provided an expansive environment with eight galaxies, each containing 256 stars. At each location, the player could fight pirates, dock at a space-station, and trade goods carried aboard their ship. Apart from providing (for that time) stunning 3D graphics, *Elite* also used a procedural approach to generate the content of each galaxy and needed only few kilobytes to store the entire universe. In 1989, *MidWinter*[4], a first-person action role-playing game by Microplay, required only less than half a megabyte to store a huge post-apocalyptic scenario spanning for 410000km^2. Nowadays, although the memory limitations are long forgotten, procedural content generation is still widely used both to reduce the design costs and to generate those immense scenarios which would be infeasible for human designers. Recent examples include, *Diablo*, *Diablo II*[5], and *Sid Meyer's Civilization*[6] which apply level-generation methods similar (in principle) to the one used in *Rogue* [37]. The massively multi-player on-line space-game *Eve Online*[7] involves a universe generated using fractal methods, conceptually similar to what was done in *Elite*. In the game *Spore*[8], players can create their own "creature" and procedural methods are applied to animate the huge variations of possible creature creations. The game *.kkrieger*[9], developed by the *.theprodukkt* team, is an extreme example of procedural content generation as the entire demo game and all its game content fits into only 97,280 bytes. In *Left4Dead*[10], the enemies and the objects are generated based on the players movements so that they will appear outside its view so as to scare the player. In the sequel, *Left4Dead 2*, the geometry and the content of the levels evolve according to the gameplay. *Borderlands*[11] provides a huge arsenal of three millions weapons generated using a single parameter vector [22]; *Far Cry 2*[12] applies procedural content generation for the weather, the day cycle, and the fire propagation. *Subversion* exploits a procedural city generator to create the game world.

[3]http://en.wikipedia.org/wiki/Elite_(video_game)
[4]http://en.wikipedia.org/wiki/Midwinter_(video_game)
[5]http://www.blizzard.com (Blizzard)
[6]http://www.civilization.com/
[7]http://play.eveonline.com/ (CCP)
[8]http://www.spore.com (Maxis 2008)
[9]http://www.theprodukkt.com/
[10]http://www.l4d.com (Valve Corporation)
[11]http://www.borderlandsthegame.com (Gearbox Software, 2009)
[12]http://www.fracry2.com (Ubisoft)

3.4.3. Search-Based Procedural Content Generation

PCG is usually based on a constructive approach: starting from a vector parameters an algorithmic procedure generate a piece of game content. A particular class PCG algorithms is based on a generative-and-test approach: after the generation of a piece of content, a procedure checks if some properties hold; if the content is not good, the generation mechanism is invoked again until the desired properties hold. In the recent years however, researchers have been investigating generative-and-test approaches [102] in which different types of search algorithms can be used to automatically discover innovative and interesting content. This research area has been recently dubbed Search-Based Procedural Content Generation (SBPCG) [102]. In Search-Based Procedural Content Generation we can define three basic elements:

1. A PCG procedure: an algorithm which, from a input vector of parameters X generate a piece of game content.

2. A scoring function: a function, usually called fitness, which gives a score to the content generated

3. A search algorithm: an algorithm which searches the space of parameters X to find the best piece of content

Even if the general approach is not based a specific search algorithms, the large majority of works in the literature [102] apply evolutionary algorithms (EAs) to search the space of solutions. Thus, many terms used in SBPCG belongs from the EAs field. In particular, the input vector of parameters X is called *genotype* while the piece of content generated by executing the PCG algorithm is called *phenotype*. In this context, the PCG algorithm, is a mapping procedure which maps the implicit representation encoded into the genotype to an explicit representation of the content which is the phenotype. The choice of the representation, represents a central issue for the application of search-based procedural content generation. From one side, the genotype representation should be reach enough to represent a wide range of interesting content. From the the other side, the genotype should be small enough to allow the search algorithm to work effectively.

3.4.4. Fitness Function

The fitness function is a function which gives a score to a piece of content. The definition of a fitness function depends on the goal of the content generation process. For example the goal can be to generate a map with a uniform distribution of the resources; a map where all the points are reachable; a platform level that is challenging to complete; a platform level that is fun to play; a racing track which provides many different types of turns.

Depending on the goal of the fitness, there are usually more than a way for computing the fitness function. On this basis we can divide the fitness functions into 3 classes: *direct*, *simulation-based*, *interactive*.

Direct Fitness function

This kind of fitness function is a function that is based on features directly computed from the phenotype. Examples of extracted features can be: the number of rooms in a map, the distribution of the resources, the number of gaps in a platform level, the average length of the stretches in a racing tracks, etc.

These features are the mapped into the fitness function. This can be *theory driven* or *data driven*. A theory driven fitness comes from a study of the problem: e.g. if the goal is to generate challenging levels for a platform game, a theoretical study can suggest a fitness which reward level with many gaps and with a big average size of the gaps.

A data driven fitness comes from a user study. Experimental sessions involving human players are used to collect data. Using the players' reported preferences and the their gameplay statistics it is possible to build a model which can predict users' fun, challenge or other emotional state. Later this model can be used to build a fitness function that can be related for example to the users' fun or challenge.

Simulation-Based Fitness function

A simulation-based fitness is a fitness which requires a simulation in the game. For this purpose, an artificial agent is used to play the generated content and the gameplay statistics are collected. Examples of this statistics can be: the number of jumps necessary for completing a particular level; the number of enemies killed in a map; the lap time and highest speed achieved in a racing track; the amount of resources collected before winning the game. The statistics collected from the artificial player can be used for example to compute a fitness which maximize the challenge.

Interactive Fitness function

When using an interacting fitness function the score of a piece of content comes directly from the user. The content evaluation can happens in two way: *out-of-game* or *in-game*. In the first case, the user does not actually play a piece of content, but he gives a score to a visual representation of the content: e.g., the shape of racing tracks, the top view of a map, the picture of a weapon. In the second case, the user play the game and interact directly with the content to evaluate. At the end of game, the user can give an explicit score of the content or an implicitly evaluation can be inferred from gameplay statistic (e.g., the number of times a weapon was used) or biomedical signals.

These methods are typically applied to problems for which a computable fitness function is difficult or even impossible to define [85]. However, interaction with humans brings new challenges such as users' fatigue. For instance, Takagi [85] presents a review of the research efforts for abating fatigue in interactive evolutionary systems; Llorà et al. [41] introduced an additional synthetic fitness function to reduce the number of users' evaluations; Gong et al. [24] apply clustering to reduce the initial population size.

Interactive evolution has been applied to a wide range of domains including the generation of HTML styles [51]; fashion design [35]; ergonomic design [3]; the generation of terrains and landscapes [107], synthetic images [75], music [112], and art in general. To the best of our knowledge, Galactic Arm Race (GAR) [28] is the only application of interactive evolution to the generation of game content. Galactic Arms Race is a multiplayer online video game driven by methods of automatic content generation. It features a unique weapon systems that automatically evolves based on players' behavior through a specialized version of the NEAT evolutionary algorithm called cgNEAT (content-generating NeuroEvolution of Augmenting Topologies) [28].

3.4.5. Related Works

In this section we review the most important works in the field of Search-Based Procedural Content Generation.

Hastings et al. [27] introduced NEAT Particles, a modified version of NEAT [82] which could be applied to interactively evolve complex and interesting graphical effects to be embedded in computed games so as to enrich their content. The work was extended in [29] where the authors introduced the game *Galactic Arms Race* and provided the first demonstration of evolutionary content generation in an on-line multi-player game. In particular, the authors applied another extension of NEAT, called context-generating NEAT, to evolve new weapons on-line based on what players used more often.

Marks and Hom [49] were the first to evolve a set of game rules to obtain a balanced board game which could be equally hard to win from either side and rarely ended with a draw.

Togelius et al. [94] combined procedural content generation principles with an evolutionary algorithm to evolve racing tracks for a simple 2D game which could fit a target player profile . Later, Togelius and Schmidhuber [96] evolved complete rule sets for games. The game engine was capable of representing simple Pacman-style games, and the rule sets described what objects were in the game, how they moved, interacted (with each other or with the agent), scoring and timing. The fitness function measured how fast another evolutionary algorithm could learn to play the game. The idea behind this fitness function is that fun is dependent on learning, and that a good game is not winnable without learning it, but should be learned quickly.

Frade et al. [21] applied Genetic Programming [36] to the evolution of terrains for games which would attain the aesthetic feelings and desired features of the designer. The approach is currently employed in the game *Chapas*[13].

El-Nasr et al. [19] presented an adaptive lighting design system, called Adaptive Lighting for Visual Attention (ALVA), that dynamically adjusts the lighting color and brightness to enhance visual attention within game environments using features identified by neuroscience, psychophysics, and visual design literature.

Zheng and Kudenko [116] presented an approach to generate commentary to football games in *Championship Manager 2008 (CM2008)*[14] by learning to map game states to high-level commentary concepts.

In [60], Pedersen and colleagues present novel methods for player modeling and suggest that the same methods might be very useful for the automatic generation of game content

More recently, Togelius et al. [101] applied multi-objective evolutionary computation to evolve maps for *StarCraft* using a set of fitness functions evaluating the player's entertainment. Sorenson and Pasquier [78] presented a generative approach for level creation following a top-down approach and validated it using *Super Mario Bros.* and a 2D adventure

[13]https://forja.unex.es/projects/chapas
[14]http://www.championshipmanager.co.uk

game similar to the *Legend of Zelda*[15].

3.5. Summary

In this chapter, we presented an overview of the two main applications of evolutionary computation to games. In the first part, we described the application of EC to evolve non-player characters and in particular we analyzed two key aspects: the representation of the behavior and the fitness function. For each of these two aspects, we presented the different approaches available in the literature. Then, we introduced on-line learning and the additional challenge that it poses.

In the second part, we defined game content and we presented procedural content generation together with some examples in the field of commercial games. Then, we presented search-based procedural content generation that usually applies evolutionary computation to search for the best game content. Finally, we discussed the different types of fitness functions and the most relevant works available in the literature.

[15]Miyamoto, S., Nakago, T., Tezuka, T.: The Legend of Zelda. Nintendo (1986)

Part II.

Racing Games

4. Computational Intelligence and Racing Games

In this chapter, we introduce the genre of racing games and we discuss why they are an interesting domain for applying computational intelligence. Then, we overview the most relevant works and the academic competitions organized in this field. Finally, we present The Open Racing Car Simulator, TORCS, which is the racing game used for the experiments of this thesis.

4.1. Racing Games

In racing games, the goal is to drive a vehicle towards a goal in an efficient manner. In its simplest form, the challenge for a player comes from controlling the dynamics of the vehicle while planning a good path through the course. In more complicated forms of racing, additional challenge comes from e.g. avoiding collisions with obstacles, shifting gears, adapting to variable weather conditions, surfaces and visibilities, following traffic rules and last but not least outwitting competing drivers, whether this means overtaking them, avoiding them or forcing them off the track.

Racing games can be classified in several ways. A first classification can be based on the accuracy of the physics simulation. On the basis of this classification we have two categories: simulators and arcade games. In a simulator, the goal is to offer a realistic driving experience by simulating the physics of a racing environment with the highest level of detail possible (e.g., rFactor[1]). Simulators are generally appreciated by expert users that aims to master any aspect of a racing car while beginners can find these games frustrating. Instead, in arcade racing games, the physics simulation is designed in order to be easy to play and to provide a fun experience rather than a realist driving experience (e.g. Need for

[1]http://www.rfactor.net/

Speed[2]). Some racing games are both simulators and arcade, since they allow to modify the type of physics used for the simulation.

A second classification can be based on the game context. Some racing games reproduce an actual event (for instance, EA F1-2010[3] reproduces the 2010 Formula 1 championship) and therefore involve predefined game content such as specific car models (e.g., the Ferrari F2010), drivers (e.g., Fernando Alonso) and tracks (e.g., Monza). Others do not reproduce a specific event but take place in a fictional game universe and thus focus on the game-play and on the driving experience itself (e.g., Gran Turismo[4]).

Racing games can be classified also on the basis of the type of vehicle involved. Besides the most common vehicles, i.e. cars and motorbikes, there are racing games based on speedboats, spaceships, bikes, airplanes, etc. In this thesis, we will focus on car racing games that is the most popular category as it is possible to see from the big number of titles published.

4.2. Motivations

Racing games resulted to be an excellent test bed for computational intelligence techniques as it is possible to see from the many papers involving this topic [81, 43, 111, 100, 99, 98] and from the several academic competition organized in this field [43, 42]. The interest of researchers for this field is due to the variety of problems that it is possible to investigate using a racing simulator.

The more common problem investigated in the literature is the development of driving behaviors. In its simplest form consists in finding the best policy which drives alone in a track achieving the shortest laptime. This represents a well defined problems which can be used as benchmark to compare different techniques of computational intelligence.

The driving problem becomes much more complex when more than one driver is involved: the driving policy has to cope with new tasks like overtaking and obstacle avoidance. When considering more than one driver, no longer exists one single optimal policy but the best policy depends on the opponents' behavior.

Racing games, often, offer an environment that can dynamically changes during a single race and can require the driver to adapt to the new conditions: e.g., fuel consumption, car damage, weather conditions, friction

[2]http://www.needforspeed.com/
[3]http://www.easports.com/
[4]http://www.gran-turismo.com/

of the roadbed, etc. This aspect can be used to test the robustness of a driving policy to small changes or for implement on-line learning techniques which adapt the driving style to the new conditions.

Some aspects of racing games can be also tackled from a strategic point of view using techniques which involves reasoning and planning. The classical scenario is a race involving many laps and many opponents: in this scenario, the planning of the pit stops strategy is crucial, and should be updated when some particular event happens (e.g. the car is too damaged).

Besides all the challenges for the game AI, racing games are also an excellent context for applying techniques of content generation. This is particularly true for arcade racing games which are not bound to any real world event. In this category of games, the principal type of content is represented from the racing tracks: the playability, the design and the number of the available tracks, can have a key role in the success of a racing game.

4.3. Related Works

Some of the early works on racing games were performed by Togelius and Lucas using a java-based 2D car racing simulator. In [95, 98], Togelius et al. focused on the driving task and investigated several schema of sensory information (e.g., first person based, third person based, etc.).

The overtaking aspect was considered in [55] using a modular fuzzy architecture and in [45] using Reinforcement Learning. In [13] evolutionary computation is applied to find the optimal racing line between the shortest path and the maximum curvature path of the track. Several works, like [52] [11] [105], applied imitation learning both to learn from human data and to generate believable drivers, i.e., drivers with a human driving-style. In [111] and in [12], genetic algorithms were applied to optimize the car setup using respectively the games Formula One Challenge and TORCS. Some of the most recent works [67, 6, 54] on racing games originate from the submission to the Simulated Car Racing Championship [42], an international competition organized by Loiacono et al. where the goal is to develop the best driver for TORCS able to race alone and against the other competitors.

Finally, the computational intelligence techniques have been also applied to some commercial racing games. In *Colin McRae Rally 2.0*[5] a neural network is used to drive a rally car, thus avoiding the need to

[5]www.codemasters.com

handcraft a large and complex set of rules [26]: a feedforward multilayer neural network has been trained to follow the ideal trajectory, while the other behaviors ranging from the opponent overtaking to the crash recovery are programmed. In *Forza Motorsport*[6] the player, can train his own *drivatars*, i.e., a controller that learns the player's driving style and that can take his place in the races.

4.4. Competitions

The interest of researchers for this field is proved by the recent competitions organized on racing games in order to compare controllers evolved using different techniques. The first competition was held at CEC-2007 (the IEEE Congress on Evolutionary Computation) and used a java 2D simulated car racing. The same competition was repeated at CIG-2007 (the IEEE Conference on Computational Intelligence and Games). The spiritual successor of this competition used the TORCS game and was held at WCCI-2008 (IEEE World Congress on Computational Intelligence) and at CIG-2008. In this new competition, the drivers are first scored when driving alone (qualifying) and then the drivers are scored in a complete race where they drive all together. In 2009, this competition became a championship: a joint event involving 3 major conferences (CEC-2009, GECCO-2009, CIG-2009). The championship pushes the competitors to take part to more than one leg and improve the final results. In 2010, the championship was held again and involved the conferences GECCO-2010, WCCI-2010 and CIG-2010. The 2010 championship brought several additional challenges: in particular the noisy sensors and a warm-up phase to allow techniques for on-line learning on the unknown racing tracks. In 2011, the championship was organized at Evo*-2011, GECCO-2011, CIG-2011. This championship used the same setup of the previous year since some challenges were still open: e.g., opponent management with noisy sensors.

During these five years, the Simulated Car Racing competition, had a great success receiving many submissions with a good variety of approaches: fuzzy logic, neuroevolution, potential fields, genetic algorithms, etc. Several submissions to the competition, resulted into one or more scientific publications [67, 6, 54].

[6]www.microsoft.com/games/

4.5. TORCS

TORCS [20], The Open Racing Car Simulator, is a open-source 3D racing game based on OpenGL technologies. Figure 4.1 shows a screenshot of the game. TORCS project was born in 1997 thanks to the work of two french programmers: Eric Espié and Christophe Guionneau. Written in C++, TORCS has been developed mainly with the idea of allowing programmers' challenges in robot development. The software simulates a fully three-dimensional environment and implements a very sophisticated physics engine, that takes into account many aspects of a real racing car, like the damage of the car, the fuel consumption, the traction, the aerodynamics, etc. TORCS provides several tracks, several models of cars, and various game modes (e.g., practice, quick race, championship, etc.). The game distribution includes many programmed bots which can be easily customized or extended to build new bots. TORCS has a wide community of users that constantly contributes to the development of the game and to the design of additional game content.

All the experiments reported in this thesis have been carried out with version 1.3.1 of TORCS.

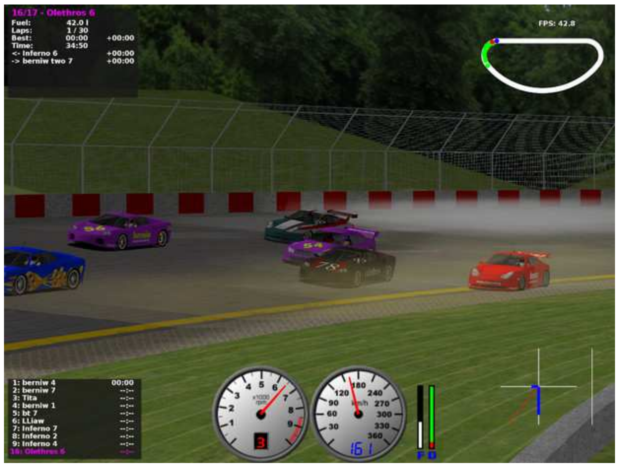


Figure 4.1.: A screenshot from TORCS

4.6. Summary

In this chapter, we introduced the genre of racing games and in particular we presented the motivations that makes this field suitable for research purposes. Racing games are an interesting domain since the offer a wide number of problems with increasing level of complexity.

Then, we presented the most relevant works in the literature and the simulated car racing competitions organized in order to compare different techniques from computational intelligence area. Finally, we presented the game TORCS, that is the open source simulator used for the experiments of this thesis.

5. Off-line Evolution of Non-player Characters

In this chapter, we apply off-line learning using NeuroEvolution of Augmenting Topologies (NEAT) to evolve a competitive racing driver. We compare the performance of our driver with several controllers submitted to an international competition on simulated car racing.

Initially, we present our approach, the competition API and the drivers used for the comparison. Our approach, for evolving a competitive driver, consists in a two steps process. In the first step, we evolve the driving and the overtaking behaviors separately. For each behavior we present the design of the controller and the experimental setup used for the learning process. In the final step, we combine the two behaviors and we evaluate the resulting controller in a complex racing task.

5.1. Evolving a Competitive Driver

In this chapter, we aim to investigate off-line learning to evolve a competitive driver for the racing game TORCS. As already introduced in chapter 3, off-line learning consists in applying a learning algorithm to find the best policy for a given task. This policy is then evaluated in a separated phase, in which the policy is applied as it is, i.e., with no possibilities of modifying its behavior. Off-line learning can be very effective when the evaluation session is very short. In fact, in that conditions, the risks of applying on-line learning can be higher than the possible advantages.

In this chapter, we evaluate our driver in the Simulated Car Racing competition which provide a very competitive contest. In particular, the drivers submitted to the competition can be used both to compare the driving performance and to evaluate the overtaking skills in a complete race.

For a competitive driver we need at least two important skills: driving and overtaking. Our approach consists in a modular driver in which these two skills are implemented in two different modules. A decision layer will activate the most suitable module depending on the race state. Thus, we evolve two separate controllers: one for driving and one for overtaking. Each controller is represent by a neural network, which is evolved using NEAT. Finally, we combine the two controllers into a modular driver.

5.2. Simulated Car Racing Competition

The Simulated Car Racing Competition is a scientific event where the goal is to develop the best driver for a racing game. As introduced in Chapter 4, many editions of this competition have been organized. In this work we focus on the Simulated Car Racing Competition held at WCCI-2008 and at CIG-2008. This edition was based on TORCS 1.3.0.

In the first part of this section we provide an overview of the API provided with the competition to control the car in the game. Then, we briefly provide a short description of the controllers submitted to the competition.

5.2.1. Competition API

In the Simulated Car Racing Competition, the competitors have been provided with a specific software interface developed on a client/server basis. The controllers run as external programs and communicate with a customized version of TORCS through UDP connections. Each controller perceives the racing environment through a number of sensor readings which would reflect both the surrounding environment (the track and the opponents) and the current game state and they could invoke basic driving commands to control the car. The complete list of sensors is reported in Table 5.1 and includes rangefinders to perceive the distance of nearby opponents, the current speed, the current gear, the fuel level, etc (we refer the interested reader to the software manual of the competition [47] for additional details). Table 5.2 reports all the driving commands: besides the rather typical driving commands (i.e., the steering wheel, the gas pedal, the brake pedal, and the gear change) a meta-command is available to reset the state of the race from the client-side.

5.2.2. Submitted Controllers

In this section we briefly introduce the controllers submitted to the past editions of the Simulated Car Racing Competition, that have been used

Name	Description
angle	Angle between the car direction and the direction of the track axis.
curLapTime	Time elapsed during current lap.
damage	Current damage of the car (the higher is the value the higher is the damage).
distFromStartLine	Distance of the car from the start line along the track line.
distRaced	Distance covered by the car from the beginning of the race
fuel	Current fuel level.
gear	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
lastLapTime	Time to complete the last lap
opponents	Vector of 18 sensors that detects the opponent distance in meters (range is [0,100]) within a specific 10 degrees sector: each sensor covers 10 degrees, from $-\pi/2$ to $+\pi/2$ in front of the car.
racePos	Position in the race with respect to other cars.
rpm	Number of rotation per minute of the car engine.
speedX	Speed of the car along the longitudinal axis of the car.
speedY	Speed of the car along the transverse axis of the car.
track	Vector of 19 range finder sensors: each sensors represents the distance between the track edge and the car. Sensors are oriented every 10 degrees from $-\pi/2$ and $+\pi/2$ in front of the car. Distances are in meters and sensors are limited to 100 meters. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), these values are not reliable!
trackPos	Distance between the car and the track axis. The value is normalized w.r.t the track width: it is 0 when car is on the axis, -1 when the car is on the left edge of the track and +1 when it is on the right edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.
wheelSpinVel	Vector of 4 sensors representing the rotation speed of the wheels.

Table 5.1.: Description of available sensors in the competition API.

Name	Description
accel	Virtual gas pedal (0 is no gas, 1 is full gas).
brake	Virtual brake pedal (0 is no brake, 1 is full brake).
gear	Gear value defined in {-1,0,1,2,3,4,5,6} where -1 is reverse and 0 is neutral.
steering	Steering value: -1 and +1 means respectively full left and right, that corresponds to an angle of 0.785398 rad.
meta	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Table 5.2.: Description of available effectors.

in the experimental analysis reported in this work (we refer the interested reader to [43] for more details about the competition and the entries submitted). The sources of all the controllers described below are available on the homepage of the Simulated Car Racing Competition [46], where are also available two sample controllers programmed in C++ and Java. In the remainder of this chapter we refer to each controller with the name of the first author, except for the sample controllers, dubbed as *C++ example* and *Java example*.

Chiu. This controller was submitted to the Simulated Car Racing Competition held in conjunction with CIG-2008. It is a human programmed controller built upon the C++ example controller provided with the competition API. The gas and the brake pedals are controlled with a policy similar to the one used in the C++ sample controller, but without any speed limit. The steering behavior basically consists of driving along the direction corresponding to the furthest distance to the track edges (according to *track* sensor).

Kinnaird-Heether et al. This controller resulted to be the second best entries to the Simulated Car Racing competition held in conjunction with WCCI-2008 [43] with an overall performance very close to the winner one. It exploits a Cultural Algorithm [69] to optimize the parameters of a programmed controller. First a programmed controller is developed decomposing it into four behaviors: (i) *acceleration*, that deals with the gas and the brake pedals to achieve the target speed; (ii) *steering*, that controls the steering wheel of the car; (iii) *shifting*, that implements the gear shifting policy; (iv) *error correction*, that is basically used to recover from critical situations. Then, a Cultural Algorithm is applied to optimize the target speed during turns used in the *acceleration* behavior.

The fitness of individuals is computed simply as the distance raced in a fixed amount of time and the best solution found was submitted as a controller for the competition.

Lucas. This controller was submitted to the Simulated Car Racing Competition held in conjunction with WCCI-2008 [43]. It is a human programmed controller that improves the Java example controller provided with the competition API. It basically increased the speed limit of the Java example controller and extended the steering and the braking policies to deal with the higher speed.

Perez et al. This controller was submitted to the Simulated Car Racing Competition held in conjunction with WCCI-2008 [43] and an improved version of the same controller was submitted to Simulated Car Racing Competition held in conjunction with CIG-2008. In this controller the knowledge is represented as a set of rules that are evolved with an evolutionary algorithm. Each rule consists of a *condition* on the car sensors (i.e., when to apply the rule) and an *action* on the car effectors (i.e., how to apply the rule). Among the available sensors, only the following one have been considered: (i) the angle w.r.t. the track axis (*angle*); (ii) the distance between the car and the track axis (*trackPos*); (iii) the current speed (*speedX*); (iv) the leftmost, the rightmost and the frontal rangefinders to detect the track edges (a subset of *track* sensors). The effectors are all the ones available in the API: the gas and brake pedals, the steering wheel and the gear shifting. The evolutionary process basically consists of selecting two rules in the knowledge base, recombining them, applying a mutation and then adding it to the knowledge base replacing the rule most similar to it. New rules are kept in the knowledge base only if they lead to a better controller, i.e., a controller with an highest fitness, computed on the basis of the lap-time and of the damage suffered by the car.

Simmerson. This controller resulted the winner of the Simulated Car Racing competition held in conjunction with WCCI-2008 [43]. It basically consists of a neural controller evolved using NEAT4j [77], a Java implementation of NEAT. The network has three outputs that control respectively the gas and brake pedal, the steering wheel and the gear shifting. The inputs of the network are chosen in a subset of the ones available in the competition API (see Table 5.1): (i) the angle w.r.t. track axis (*angle*); (ii) the current speed (*speedX*); (iii) the 19 track rangefinders (*track*); (iv) the current gear (*gear*); (v) the spin speed of wheels (*wheelSpinVel*); (vi) the distance between the car and the track axis (*trackPos*); (vii) the current RPM of the engine (*rpm*). The fitness used in the evolutionary process is computed as the distance raced within a fixed amount of game tics, penalizing it for the amount of damage received and the time spent out of the track.

Tan et al. This controller was submitted to the Simulated Car Racing Competition held in conjunction with WCCI-2008 [43]. It was developed with a three-step process. First, the sensory information was aggregated and preprocessed; second, a parametrized controller based on simple rules was designed; finally, the parameters of controller were optimized using evolution strategies. The resulting controller drives in the direction where the rangefinder sensors indicate the largest free distance, with a speed dependent on that distance.

5.3. Evolving the Driving Skill

First we focused on evolving a controller that drives as fast as possible when racing alone on the track. In the following of this section we describe the design of the evolved controller, i.e., we define the inputs and the outputs of the neural network evolved. Then, we describe the experimental setup used to evolve the neural controller, including the fitness function used. Finally we report the experimental results.

5.3.1. Controller Design

In order to apply successfully NEAT to evolve a controller for TORCS, the choice of the proper inputs and outputs of the network plays a key role. As inputs of the neural network we focused on a subset of the sensory information provided with the competition API: (i) six rangefinder sensors to perceives the track edges (provided by the *track* sensor) along the directions $\{-90°, -60°, -30°, +30°, +60°, +90°\}$; (ii) an improved frontal sensor computed as the biggest value among the ones returned from the three rangefinders along the directions $\{-10°, 0°, +10°\}$; (iii) the current speed of the car (*speedX*). Our results show that such a small subset of the sensory inputs available is enough to evolve a fast controller with NEAT. In addition, our empirical analysis suggests that replacing the frontal rangefinder, i.e., the one parallel to the car axis, with the improved frontal sensor leads to a better controller. In fact, the frontal sensor is exploited from the controllers to detect either when a turn is approaching or when it is over: the improved frontal sensor detect more reliably the begin and the end of turns, especially when the car is not perfectly aligned to the axis of the track (see the example in Figure 5.1).

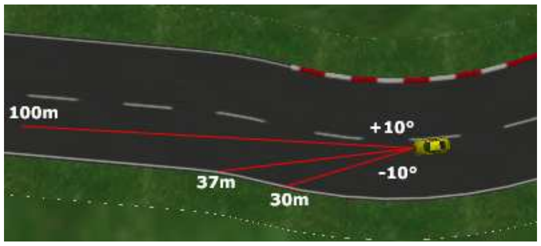


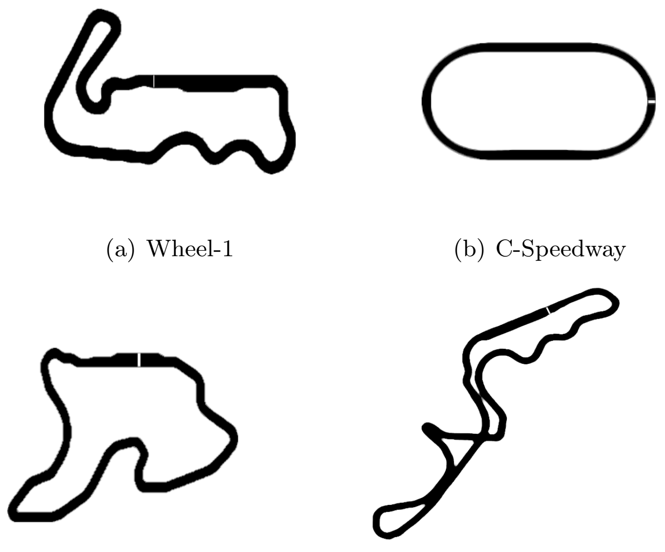
Figure 5.1.: An example of how the improved frontal sensor works. In this example, the rangefinders parallel to the car axis returns 37m even if the turn is almost over, preventing the controller from applying a full acceleration. The improved frontal instead returns the biggest among the three rangefinders reading, i.e., 100m.

Concerning the outputs of the neural controller, we used two continuous outputs in the range [-1,1]. The first one is used to control the steering wheel, according to the *steering* effector described in Table 5.2. The second one is used to control the gas and brake pedals as follows. If the output is less than zero it is assigned, as a positive value, to the *brake* effector, resulting in a braking command. Otherwise it is assigned to the *accel* effector, resulting in an acceleration command. In addition, when the car is in a straight segment of the track, i.e., when the frontal sensor does not perceive the track edge within 100 meters, the gas pedal is set to 1 and the brake pedal is set to 0. Such a design choice forces the controller to drive fast since the early generations and prevents the evolutionary search from wasting time with safe but slow controllers.

Finally, to deal with the gear shifting, we used a programmed policy: while it is quite complex to develop a good policy that controls the speed and the trajectory of the car, an effective gear shifting policy can be quite easily programmed. The controller is also provided with a scripted recovery policy to be used when the car goes outside of the track.

5.3.2. Controller Training

To train the driving behavior we evolved a population of 100 networks for 150 generations with the standard C++ implementation of NEAT [83]. The evolved networks are evaluated on the basis of their performance when racing alone on the `Wheel 1` track depicted in Figure 5.2(a). This track was chosen because it is a fast track but, at the same time, it involves several complex turns, being in our opinion a good mix of the



(a) Wheel-1 (b) C-Speedway

(c) E-track-6 (d) Wheel-2

Figure 5.2.: Tracks used for the experiments reported in this chapter.

tracks available in TORCS. The evaluation process consists of two laps on the `Wheel 1` track. First, in the *warm-up* lap, the network to be evaluated is loaded into the bot that starts to race. Then, in the *evaluation* lap, the performance achieved is recorded and used to compute the fitness of the network as follows:

$$F = C_1 - T_{out} + C_2 \cdot \bar{s} + d,$$

where T_{out} is the number of game tics the car is outside the track; $\bar{s}$ is the average speed (meters for game tic) during the evaluation; d is the distance (meters) raced by the car during the evaluation; C_1 and C_1 are two constants introduced respectively to make sure that the fitness is positive and to scale the average speed term (both C_1 and C_2 have been empirically set to 1000 in all the experiment reported here).

5.3.3. Experimental Results

To test our approach we compared the controller evolved to the ones submitted as entries to the past editions of the Simulated Car Racing Competition. The comparison has been carried out following the same

Controller	`C-Speedway`	`E-Track 6`	`Wheel 2`
NEAT driver	15528.90	**7566.40**	8534.31
Kinnaird-Heether et al.	14573.80	5375.44	6804.25
Simmerson	12629.50	6386.44	2792.97
Tan et al.	12590.00	5136.41	6823.36
Perez et al.	5540.09	4428.98	4612.84
Chiu	**16721.50**	6820.00	**8823.05**
Lucas	12240.30	5022.28	3943.58
C++ example	7265.36	4930.86	4664.39
Java example	5711.37	2932.35	2355.26

Table 5.3.: Distance raced by each controller within 10000 game tics on the `C-Speedway`, `E-Track 6`, and `Wheel 2` tracks. Statistics reported are the medians computed over 10 runs.

approach used for the competition: each controller is scored when racing alone on three different tracks. The performance of the controller is measured as the distance raced by each controller in 10000 game tics, corresponding to 200 seconds of simulated game. In the following experimental analysis we used the same three tracks used for the Simulated Car Racing Competition held at CIG-2008: `C-Speedway`, `E-Track 6`, and `Wheel 2` (depicted respectively in Figure 5.2(b), in Figure 5.2(c), and in Figure 5.2(d)). Table 5.3 compares the performance of the controllers described in Section 5.2 to our controller, dubbed as *NEAT driver*. The first five controllers reported in Table 5.3 have been developed applying an evolutionary algorithm, while the last four controllers are entirely programmed. The results show that the controller evolved with our approach has the highest performance among the evolved controllers, while the programmed controller developed by Chiu appears to be slightly faster in two tracks out of three. Therefore, NEAT is able to evolve a neural controller almost as fast as the best programmed controllers also on tracks different from the one on which it was trained. In addition, the analysis of behavior of the evolved controller reveals that although it does not always follows the optimal trajectory, it controls the car very reliably avoiding to race outside the edges of the track.

To make a fair comparison of the controllers reported in Table 5.3, we should underline that they are evolved using two different approaches. The first approach, followed by Simmerson and by Perez et al. consists of evolving the controller from scratch without using any prior knowledge on the structure of the controller. The second approach, followed by

Kinnaird-Heether et al. and by Tan et al. exploits an evolutionary algorithm to optimize the parameters of a designed driving behavior. Instead, our approach falls somewhere between the former and the latter: the driving skill is evolved from scratch but then is combined with a programmed gear shifting policy and a programmed recovery policy to develop the final controller. Accordingly, the proposed approach does not require any strong assumption on the structure of the searched controller, but at the same time does not involve to deal with a too complex and expensive optimization problem. The results suggest that the proposed approach is effective in practice and leads to a better performance than the one achieved following different approaches.

5.4. Evolving the Overtaking Skill

In the previous section we showed that NEAT can be applied to evolve a controller with good driving skills. Unfortunately this is not enough to develop a competitive car controller for a racing game, where the capabilities of overtaking the opponents and avoiding the collisions are very important. Nevertheless, most of the controllers submitted to the Simulated Car Racing Competition fails to deal with this issue. Accordingly, in the past editions of the Simulated Car Racing Competition the winner was not the fastest controller submitted but the one with the best tradeoff between driving and overtaking capabilities. Therefore, in this section we apply NEAT to evolve a controller with overtaking skills. The section is organized as the previous one. First, we define the controller architecture, then we describe the experimental setup used to train the controller, and finally we discuss the experimental results.

5.4.1. Controller Design

To define the inputs of the neural controller, we focus again on a subset of the available sensors: we use the same inputs described in the previous section and some additional inputs to perceive the presence of nearby opponents on the track. In particular, we introduced eight additional inputs provided by the *opponents* sensor of the competition API: (i) four beams that cover the frontal area of the car between - 20° and + 20° with respect to the car axis; (ii) two diagonal beams that cover respectively the area between - 40° and - 50° and the area between + 40° and + 50° with respect to the car axis; (iii) finally two lateral beams that cover the area between - 70° and - 80° and + 70° and + 80° with respect to the car axis. Similarly to what found in the previous experiment, few inputs are enough to evolve an effective overtaking behavior: in the most critical area of the car, i.e. the frontal area, we use four inputs to represent the presence of opponents while in the lateral and diagonal we use only two inputs. The outputs of the neural controller are the same of the controller previously evolved: one output is used to control the steering wheel and one output is used to control the gas and brake pedals.

5.4.2. Controller Training

In order to evolve the overtaking skill, we designed a specific evaluation process that involves an *overtaker* bot that races against an *opponent* bot. At the beginning of each evaluation, a programmed controller is used to align the *overtaker* and the *opponent* in a random segment (i.e., it can be a straight strong turn, a chicane, etc.) of the track as shown in Figure 5.3: the bots are placed with an horizontal *offset* with respect to axis of the track that is randomly selected in $\{-3m, 0m, +3m\}$; the *opponent* is placed ahead the *overtaker* at a *distance* randomly chosen between $10m$ and $20m$. As this initial setup is completed, the neural

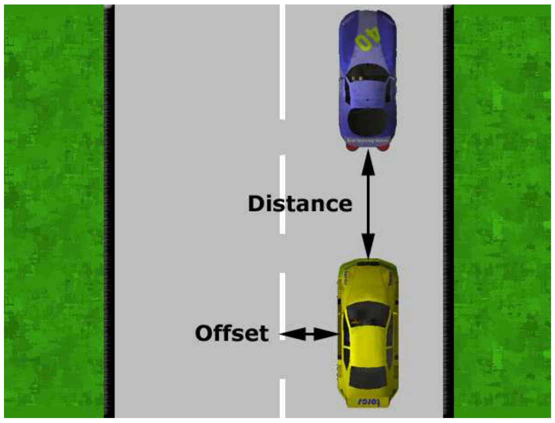


Figure 5.3.: Initial setup used to evaluate the evolved overtaking skill. The blue car (at the top) is the *opponent*, while the yellow car (at the bottom) is the *overtaker*.

controller is loaded in the *overtaker* bot and the neural controller tries to overtake the *opponent*. After 2000 game tics, corresponding to 40s of simulated time, the evaluation is over and the performance of the

Controller	Overtaking Time	$T_{collision}$	T_{out}	Success Rate
NEAT driver	351.77 ± 151.21	109.03 ± 107.99	1.11 ± 11.00	55.2%
NEAT overtaker	**271.49 ± 135.57**	**47.10 ± 97.85**	13.50 ± 39.38	**86.7%**
Chiu	296.65 ± 165.93	152.38 ± 183.14	7.36 ± 22.16	64.1%
Simmerson	397.76 ± 102.62	145.59 ± 150.53	**0.00 ± 0.00**	56.8%

Table 5.4.: Performances of each controller on the overtaking of a slower opponent. Statistics reported are the averages computed over 1000 overtakes.

controller is computed as:

$$P = C - \alpha \cdot T_{out} - \beta \cdot T_{collision} - \gamma \cdot \Delta,$$

where T_{out} is the number of game tics the *overtaker* is outside the track; $T_{collision}$ is the number of game tics a collision with the *opponent* is detected, Δ is the difference between the position of the *opponent* and the position of the *overtaker* (i.e., a negative value means that the overtake succeeded while a positive means that it failed); C is a constant used to make sure that the fitness is positive, while α, β, γ are used to weights the contribution of each term to the fitness (in the experiments reported in this work, we empirically set $C = 8000$, $\alpha = 5$, $\beta = 10$, and $\gamma = 3$). Finally, the fitness of each controller in the population is computed as the average performance achieved over 20 evaluations, in order to assess the quality of the solution in several conditions, i.e., in different track segments.

5.4.3. Experimental Results

In this second set of experiments, we compared the controller with the *overtaking skill* evolved, dubbed *NEAT overtaker*, (i) to the *NEAT driver*, (ii) the winner of the WCCI-2008 edition of the Simulated Car Racing Competition, i.e., the Simmerson's controller and (iii) to the the best programmed controller submitted so far to the Simulated Car Racing Competition, i.e. the Chiu's controller. To compare the four controllers we performed 1000 overtakes following almost the same experimental design used to evolve the *overtaking skill*, except for the initial *offset* and *distance* of the cars that are uniformly chosen respectively in the interval $[-3m, +3m]$ in the interval $[10m, 20m)$, to test the controllers in a broad range of conditions. Table 5.4 compares the performance of the four controllers. The first column, *Overtaking Time*, reports the average number of game tics necessary to overtake the opponent; the second column,

$T_{collision}$, reports the average number of game tics in which a collision with the opponent is detected; the third column, T_{out}, reports the average number of game tics in which the controller is out of the track edges; finally, the last column, *Success Rate*, reports the percentage of successfull overtakes performed by the controller, where an overtake is defined as successfull if, after 500 game tics, corresponding to 10s, the distance between the controlled car and the opponent is at least $10m$. The results show that the evolved overtaking behavior, reported as *NEAT overtaker*, outperforms the other controllers except for T_{out}, the average number of game tics the controller is out of the track.

We applied a *one-way* analysis of variance or ANOVA [23] to test whether the differences in Table 5.4 are statistically significant. We also applied the typical post-hoc procedures (SNK, Tukey, Scheffé, and Bonferroni) to analyze the differences among the four controllers. The analysis of variance shows that there are differences statistically significant at the 99.99% confidence level. In particular, according to the post-hoc procedures: (i) in terms of average overtaking time, the *NEAT overtaker* is significantly better than the others; (ii) in terms of collision avoidance capabilities ($T_{collision}$) the *NEAT overtaker* is significantly better than the others and the *NEAT driver* is significantly better than the Simmerson's and Chiu's controllers; (iii) finally, in terms of capabilities of keeping the track (T_{out}) the *NEAT overtaker* is significantly worse than the other controllers. The results are not surprising as they suggest that the evolved *NEAT overtaker* is able to overtake the opponents faster and more often than the others controller, avoiding the collision as much as possible. However the *NEAT overtaker* also appears to have worse driving capabilities than the other controllers, resulting in a higher average number of game tics spent outside of the track.

5.5. Combining the Skills

In the final experiment we compares the controllers on a complete race against several opponents, to test whether the evolved overtaking skill really gives a competitive advantage in a racing environment. The experiment consists of a complete race against six opponents chosen among the controllers used in the experiments reported in Section 5.3: (i) the Kinnaird-Heether's controller, (ii) the Tan's controller, (iii) the Perez's controller, (iv) the Lucas' controller, (v) the Java example controller and (vi) the C++ example controller. In this experiment we compared the four controllers considered in the previous section to a new controller

Controller	`C-Speedway`	`E-Track 6`	`Wheel 2`	Total
NEAT mixed	9	8	**10**	**27.0**
NEAT overtaker	8	4	9	21.0
NEAT driver	**10**	3.5	5	18.5
Chiu	5	4	4.5	13.5
Simmerson	7	**10**	4	21.0

Table 5.5.: Comparison of the scores achieved by each controller racing against 6 opponents and starting from the last position on the `C-Speedway`, `E-Track 6`, and `Wheel 2` tracks. Statistics reported are the medians computed over 10 races.

that combines the *driving skill* and the *overtaking skill* evolved. This controller, dubbed *NEAT mix*, was developed in a straightforward way: it behaves as the *NEAT overtaker* when it perceive an opponent at a distance equal or smaller than $40m$, otherwise it behaves as the *NEAT driver*. For each controller, we run 10 races on the three tracks of TORCS used also in the previous section: `C-Speedway`, `E-Track 6`, and `Wheel 2`. In each run, the controller to test starts from the last position of the starting grid, while the first six position are populated with a random permutation of the six opponents. At the end of each race, a score is assigned to the controller according to the F1 point system, following the scoring procedure used also in the Simulated Car Racing competition [43]: 10 points to the first, 8 points to the second, 6 points to third, 5 points to the fourth and so on.

Table 5.5 compares the median score of the five controllers for each track and, in the last column, it reports the total score computed as the sum of the median score collected in each track. The results show, that the *overtaking skill* is very important when racing against several opponents; e.g., fast controllers with poor overtaking capabilities, like the Chiu's one and the *NEAT driver*, have a worse score than a slower controller with better overtaking capabilities, like the *NEAT overtaker*. On the other hand, the *driving skill* is very important too; e.g., although the *NEAT overtaker* has the best overtaking capabilities, it does not outperform the other controllers due to its inferior driving capabilities. Accordingly, *NEAT mix* outperforms all the other controllers, as it combines good driving and overtaking capabilities to win the races.

5.6. Summary

In this work we applied off-line neuroevolution to evolve a neural controller for TORCS, that is able to drive fast when racing alone as well to behave reliably in presence of opponents. First, we applied NEAT to evolve a *driving skill*, that is a neural controller specifically devised to race alone as fast as possible on different types of tracks. Then, we extended the same approach to evolve an *overtaking skill*, that is we evolved a neural controller that is able to overtake an opponent and to avoid collisions in a broad range of situations. Finally, we combined the evolved skills in a single controller that is able to drive fast as well to challenge several opponents in a race.

To test our approach, an empirical analysis was performed following the same guidelines used to evaluate the entries submitted to the Simulated Car Racing Competition. In the first experiment we tested the performance of the evolved behaviors alone. Our results show that NEAT is able to evolve a *driving skill* that is competitive with the best human programmed controller available. Similarly, the *overtaking skill* evolved by NEAT outperformed both the best programmed and the best evolved controllers. In the final experiments we compared the best controllers of the Simulated Car Racing Competition to the ones evolved with our approach: a controller with only the *driving skill*, a controller with only the *overtaking skill*, and a controller that combines both the skills. Such a comparison was performed on a challenging task: racing against six opponents starting from the last position of the starting grid. The results suggest that either the *driving skill* or the *overtaking skill* alone does not lead to a very competitive and reliable car controller. Instead, the controllers that exploits both the skills is able to outperform all the other controllers, including the best submitted to the past editions of the Simulated Car Racing Competition.

6. On-line Evolution of Non-player Characters

Off-line learning focuses on the final performance. In contrast, on-line learning focuses both on the performance of learning process and on the final performance. In fact, on-line learning is applied in-game: thus, the learning process should be fast and as transparent as possible to the human player.

In this chapter, we investigate on-line learning for evolving a driving behavior in TORCS. In particular, we consider two cases: (i) learning a driving behavior from scratch, and (ii) adapting an existing driving behavior to a new track. Our experimental analysis takes into account three tracks with increasing difficulty and compares two methods of on-line neuroevolution: NEAT and rtNEAT.

6.1. On-line Neuroevolution

Our work is based on two methods of on-line neuroevolution: (i) the generational approach, introduced by Whiteson and Stone [108], which extends NEAT for the on-line scenario, and (ii) its steady-state version [68], which extends the steady-state version of NEAT (rtNEAT) to an on-line scenario.

6.1.1. On-line Generational Neuroevolution with NEAT

In [108], Whiteson and Stone extended NEAT for on-line learning in stochastic problems. Instead of computing the fitness of each individual as an average over a fixed number of repeated evaluations (as usually done when off-line neuroevolution is applied to stochastic problems), a variable number of evaluations is allocated to each individual based on a mechanism that borrows from the action-selection strategies employed in reinforcement learning. For this purpose, Whiteson and Stone [108] introduce a budget of evaluations that the system can spend on the individuals in the populations. As in off-line neuroevolution, each evaluation involves one problem instance. Opposite to what happens in typical off-line neuroevolution, each individual receives a variable number of evaluations that depends on its performance: the most promising individuals receive an increasingly higher number of evaluations while less promising individuals receive fewer and fewer evaluations [108]. For this purpose, the system needs to balance exploration and exploitation as do reinforcement learning algorithms. Accordingly, on-line neuroevolution applies an *evaluation selection strategy*, borrowed from reinforcement learning, to identify the most promising candidates in the population so as to allocate more resources (more evaluation slots) to them.

The on-line version of NEAT is reported as Algorithm 1. At first, the population is randomly initialized as in the most typical evolutionary algorithm (Algorithm 1, line 2). Then, the individuals in the population are evaluated (Algorithm 1 from line 4 to line 12) according to a certain action-selection strategy (Algorithm 1, line 9). For this purpose, for each individual p in the population P, the system maintains a vector $f(p)$ estimating the individual fitness and a vector $e(p)$ keeping the number of evaluations performed for p; the vector $e(\cdot)$ is initialized to zero while $f(\cdot)$ is initialized with the minimum fitness value which depends on the problem. Every time an individual is selected for the evaluation, its counter is increased (Algorithm 1, line 10) and its estimated fitness is updated (line 11) based on the result of the current evaluation and its previous value. The selection of an individual is performed by the function call SELECT(p) at line 9 in Algorithm 1 that implements one of the evaluation selection strategies proposed in the literature [108, 8] (see Section 6.1.3). When the criterion to stop the evaluation is met (line 12), the evolutionary process continues as usual with selection, recombination and mutation. These steps are repeated until a target number of generations have been completed.

6.1.2. On-line Steady-State Neuroevolution with rtNEAT

rtNEAT [80] is the steady-state version of NEAT in which the usual population-based selection, recombination and mutation operators are replaced with the following steps. First, the worst individual (according to its shared fitness) in the population is removed. Then, two parents are selected, recombined and mutated so as to generate a new individual which is added to the population. rtNEAT was extended for an on-line scenario in [53] following what was already done by Whiteson and Stone for NEAT [108]. The on-line version of rtNEAT works basically the same as on-line NEAT, the only difference with respect to the generational

Algorithm 1 Generational On-line Evolution.

```
1: procedure EVOLUTION($P$)
                        ▷ $P$ is the population, $p$ is an individual of $P$
                        ▷ $e(p)$ is the number of evaluations of $p$
                        ▷ $f(p)$ is the fitness of $p$
2:     Init($P$);                                  ▷ Randomly initialize $P$
3:     repeat
                        ▷ Init fitness and evaluation count in $P$
4:         for $p \in P$ do
5:             $e(p) = 0$
6:             $f(p) = f_{min}$                    ▷ Set to min fitness
7:         end for
8:         repeat
                        ▷ Select the individual to evaluate
9:             $p \leftarrow$ SELECT($P$);
                        ▷ Update the evaluation count
10:            $e(p) \leftarrow e(p) + 1$;
                        ▷ Update the fitness of based on
                        ▷ its current evaluation EVAL($p$)
11:            $f(p) \leftarrow f(p) + \frac{1}{e(p)}(\text{EVAL}(p) - f(p))$;
12:        until Evaluation Termination Criteria Met;
                        ▷ Usual NEAT operators
13:        $P_s \leftarrow$ Selection($P$);
14:        $P_r \leftarrow$ Recombination($P_s$);
15:        $P_m \leftarrow$ Mutation($P_r$);
16:        $P \leftarrow P_m$;
17:    until Maximum Number of Generations Reached
18: end procedure
```

version (Algorithm 1) are the lines from 12 to 16 that in rtNEAT are replaced with steady-state evolution.

6.1.3. Evaluation Selection Strategies

In our study, we considered all the four evaluation selection strategies introduced in the literature: three of them (ε-Greedy, Softmax and Interval-based) are the ones considered in [108]; ε-Greedy-Improved is an extension of ε-Greedy we introduced in [7, 9]. Each strategy has been used to implement the `SELECT` procedure for generational or steady-state on-line neuroevolution (see Algorithm 1).

ε-Greedy is probably the simplest action-selection strategy used in reinforcement learning [84] and its application to on-line evolution is rather straightforward. At each iteration, it selects with probability ε a random individual from the population while with probability $1-\varepsilon$ it selects the individual with the highest fitness (see [8] for an algorithmic description). The evaluation process ends (Algorithm 1 line 12) after k evaluations are completed. The parameter k must be of the same order of the number of new individuals added to the population through recombination and mutation. Accordingly, for on-line NEAT, a generational algorithm, k has to be comparable to (possibly larger than) the population size [108], whereas for on-line rtNEAT, a steady-state algorithm, k will be small since only one new individual in the population is added during each generation. For instance, in the experiments reported in this work k is 300 for on-line NEAT while it is set to 5 for on-line rtNEAT. Note that this termination condition does not guarantee that all the individuals in the population will be evaluated at least once [108, 8].

ε-Greedy-Improved is an extension of ε-Greedy which we devised to avoid some of the limitations of the previous strategy. Like ε-Greedy, this strategy selects the best individual with probability $(1-\varepsilon)$ while with probability ε it selects a random individual *among the ones that have not been evaluated yet*. However, in this case, the best individual is eligible for selection only if its fitness is higher than a threshold θ_{best}. The individuals evaluation process ends when (i) *all the individuals have been evaluated at least once* and (ii) the best individual of the current population has been evaluated for at least θ_{eval} or it is not good enough (i.e., its fitness is below a threshold θ_{best}). These termination criteria guarantee that all the individuals of the population have been evaluated and that the fitness of the champion is very accurate. The parameter θ_{best} controls the exploration/exploitation trade-off: the higher the value of θ_{best} is, the more the time spent exploring the search space for better networks will be. In practice, θ_{best} should be set to the lowest fitness value that a *good* network (i.e., a network with a reasonably high performance) should have. The parameter θ_{eval} is used to adjust the accuracy of the evaluation process: the noisier the single evaluation is, the higher the value of θ_{eval} should be.

The ε-Greedy-Improved strategy is reported as Algorithm 2. At first, line 3, the fitness of the best individual in the population is compared to the target threshold. If its fitness is too low, an individual from the population is randomly selected (note that, such an individual exists since otherwise, the termination condition would have been met at the

Algorithm 2 ε-Greedy-Improved selection strategy.

```
1: procedure SELECT($P$)    ▷ $P$ is the population, $p$ is an individual of $P$
                                        ▷ $e(p)$ is the number of evaluations of $p$
                                                            ▷ $f(p)$ is the fitness of $p$
2:     $best = \text{argmax}_{p \in P} f(p)$;
                                        ▷ If the best individual is not good enough
                                                    ▷ return an individual never
                                                            ▷ evaluated (explore)
3:     if $(f(best) \leq \theta_{best})$ then
4:         return $p|e(p) = 0$;
5:     end if
                                        ▷ If all the individuals have been evaluated
                                                ▷ return the best individual (exploit)
6:     if $(\nexists p \in P|e(p) = 0) \wedge (e(best) < \theta_{eval})$ then
7:         return $best$;
8:     end if
                                        ▷ Otherwise apply the $\varepsilon$-greedy strategy
9:     if $(rand() < \varepsilon)$ then
10:        return $p|e(p) = 0$;
                                ▷ Return an individual never evaluated (explore)
11:    else
12:        return $best$;
                                        ▷ Return the best individual (exploit)
13:    end if
14: end procedure
```

previous iteration so that the evaluation process would have been terminated). Then, if all the individuals have been evaluated but the best has not been evaluated enough, the best is reevaluated (the process continues in its exploitation of the current result). Otherwise, if the best individual is eligible (its fitness is high enough) and other candidates have not been evaluated, the usual ε-Greedy selection is applied (line 9).

Softmax is a probabilistic action-selection strategy widely used in reinforcement learning. When applied to on-line evolution [108], it computes the selection probability of an individual p by using the Boltzmann distribution [84, 108] as follows,

$$\frac{e^{\frac{f(p)}{\tau}}}{\sum_{p' \in P} e^{\frac{f(p')}{\tau}}}$$

Algorithm 3 Interval-Based selection strategy.

```
1: procedure SELECT($P$)
                    ▷ $P$ is the population, $p$ is an individual of $P$
                    ▷ $e(p)$ is the number of evaluations of $p$
                    ▷ $f(p)$ is the fitness of $p$
                    ▷ Initially, all the individuals are evaluated
2:     if $\exists p \in P | e(p) = 0$ then
3:         return $p$;
4:     else                    ▷ Adjust the fitness according to
                               ▷ an $\alpha$ confidence interval
5:         return $\text{argmax}_{p \in P} \left[ f(p) + z_{\left(\frac{1+\alpha}{2}\right)} \cdot \frac{\sigma(p)}{\sqrt{e(p)}} \right]$;
6:     end if
7: end procedure
```

where $f(p)$ is the fitness of p, P is the population and τ is a parameter to control the balance between exploration and exploitation which is set empirically [108]. A low value of τ favors exploitation, since small differences in the fitness of individuals will correspond to large differences in their selection probabilities; a high value of τ favors exploration, since even large differences in the individual fitnesses will correspond to small differences in their selection probabilities. The evaluation process terminates after k evaluations are completed [108]. The parameter k is set using the same criterion used for ε-Greedy.

Interval-based. None of the previous strategies takes into account the uncertainty affecting the current estimation of the fitness of the individuals. Interval-based selection addresses this issue by introducing confidence intervals (see Algorithm 3). At first, all the newly created individuals are evaluated, then for the remaining iterations the most promising individual (the one with the highest upper bound of its confidence interval) is returned (Algorithm 3, line 5). As for ε-Greedy and Softmax the evaluation process stops after k iterations. Also in this case, the parameter k is set using the same criterion used for ε-Greedy.

6.2. Learning to Drive using On-Line Neuroevolution

The application of off-line neuroevolution to TORCS is straightforward [10]: the population consists of candidate drivers (i.e., networks) whose fitness is computed as their performance on one problem instance which corresponds to driving for one or more laps; most important, all the evaluations are independent and they can be potentially carried out in parallel.

In contrast, the use of on-line neuroevolution in TORCS poses several challenges since there is just one car racing on a track and a population of networks competing to gain control of that one car. This scenario demands for reasonable performances in a limited amount of time (i.e., game ticks) and makes the evaluation of each network using several laps infeasible. In fact, it would be unacceptable to allow networks with poor driving capabilities to control the only car for too long. As a consequence, the evaluations of candidate drivers have to be carried out using rather short time slots. This approach however opens up several issues since (i) each candidate network is tested on rather brief sections of the track, (ii) the evaluation of an individual intrinsically depends on the results of the evaluation of the previous individuals; (iii) there might be several abrupt and unsafe changes in the car driving behavior when a controller replaces the previous one (in fact, different controllers might have totally different ways to deal with the same situation, leading to chopped driving behaviors); finally, (iv) extremely poor controllers might stop the entire evaluation process (for instance, if the car gets stuck somewhere, all the subsequent controllers will perform poorly and possibly lead to a premature convergence).

Since evaluations cover only brief portions of the track (because of the short time slots), they might easily lead to either underestimate or overestimate the performance of candidate drivers. For instance, a network might have a high fitness, just because it has been evaluated only on favorable parts of the track (e.g., straight stretches). Most important, in the on-line scenario, the evaluation of a candidate driver begins where the evaluation of the previous one ended. Thus, an evaluation intrinsically depends on the results of the previous ones; for instance, a candidate network might be evaluated starting in an unfavorable position of the track just because the previous drivers performed poorly. The on-line neuroevolution approaches applied in this work (see Section 6.1) aim at reducing the noise and the uncertainty due to the evaluation process (i) by reevaluating the most promising candidates more than once and (ii) by averaging the results of more evaluations into a unique fitness value.

Finally, an evaluation of a poor controller might jeopardize the entire learning process. For instance, the car might get stuck somewhere off-track so that the subsequent candidate drivers would receive low fitnesses possibly leading to a premature convergence. To avoid this issue, we introduced a recovery procedure that is activated when the car is outside the track (where rangefinder inputs are unavailable [43]). The recovery policy was implemented as a separate programmed module inspired to the one available in the drivers distributed with TORCS (similarly to what was done in [54]).

6.3. Design of Experiments

All the experiments presented in this work have been carried out with TORCS 1.3.1, using the setup of the 2009 Simulated Car Racing Championship [44], and the version 1.0 of the NEAT/rtNEAT package distributed by the NEAT user group[1], modified to compile with gcc-4.1.2.

Sensors and Actuators. We used the sensors provided by the competition software to build a sensor model consisting of (i) six rangefinder sensors to perceive the track edges along several directions (i.e., -90°, -60°, -30°, +30°, +60°, +90°); (ii) an aggregate sensor to perceive the track edges *in front of the car*, which combines the readings of the three rangefinders along the direction - 10°, 0° and + 10°; and (iii) the current speed of the car. The effectors provided in the competition software were mapped into two real-valued effectors that control the steering wheel and the gas/brake pedals. In addition, when the car frontal sensor does not perceive the track edge within 100 meters (i.e., when the car is probably on a straight stretch), the gas pedal is set to the maximum value by default. Therefore, drivers need to control the gas/brake pedals only when facing a turn. This design forces controllers to drive as fast as possible from the very early generations and prevents the evolutionary search from wasting resources on reliable but slow controllers. Note that, drivers do not control the gear shift which, in this work, is controlled by a programmed policy taken from one of the drivers distributed with TORCS.

Experimental Setup. An experiment consists of ten runs. In each run, a driver is evolved using either the on-line version of NEAT or the on-line version of rtNEAT and one of the four evaluation-selection policies available (i.e., ε-Greedy, ε-Greedy-Improved, Softmax, and Interval-based).

Each run lasts for 2000 laps (i.e., the car must complete 2000 laps before the evolutionary process stops) and involves a population of 100

[1]`http://www.cs.ucf.edu/~kstanley/neat.html#group`

controllers. An evaluation slot consists of 1000 game ticks. Thus, a network is evaluated for approximately 20 seconds of actual play time, then the control is passed to the next network to be evaluated. Note that, our timeslot is very short, actually shorter than the time needed to complete one lap on the simplest track (A-Speedway). In fact, the fastest driver we evolved in A-Speedway requires around 30 seconds (Section 8.3.2) to complete a lap so that our evaluation slot is only the 66% of the fastest lap. In a more difficult track like Ruudskogen this difference is more dramatic where, in the initial learning stage, the evaluation timeslot roughly corresponds to the 5% of the average lap time (Table 6.1a).

The fitness of a controller during an evaluation slot is computed as,

$$eval = \eta - T_{out} + \beta \cdot \bar{s} + d,$$

where T_{out} is the number of game ticks the car spent outside the track; $\bar{s}$ is the average speed during the evaluation, computed as meters for game tick; d is the distance raced by the car during the evaluation, computed as meters run; η and β are two constants introduced to guarantee that the fitness is positive and as a scaling factor for the average speed term (both η and β have been empirically set to 1000 in all the experiments reported here). All the NEAT/rtNEAT parameters were set to their default values, taken from the configuration files distributed with the software package, except for a few ones.[2] For ε-Greedy, Softmax, and Interval-based, k is set to 300 for on-line NEAT and to 5 for on-line rtNEAT following the indications given in [108]. The other parameters were set to values which we empirically determined. In particular, for ε-Greedy and ε-Greedy-Improved, ε is set to 0.25; for ε-Greedy-Improved, θ_{eval} is 5 (the best individual must be evaluated at least five times) and θ_{best} is 2100 (i.e., only individuals with a fitness higher than 2100 can be considered champions); for Softmax, τ is 50; for Interval-based, the confidence level α has been set to 0.8. All the statistics reported in this work are averages over ten runs.

Statistical Analysis. To analyze the results produced during each experiment, we performed a two-way analysis of variance (ANOVA) followed by the typical post-hoc procedures (SNK, Tukey, Scheff´e, and Bonferroni) [23]. The two-way ANOVA has been applied to check whether there are significant differences in the results with respect to (i) the

[2]With respect to the default settings we modified the following parameters: `weigh_mut_power`=0.1; `recur_prob`=0.05 (i.e., we allowed recurrent connections); `mutdiff_coeff`=1.0; `compat_thresh`=1.0; `mutate_add_node_prob`=0.005; `mutate_add_link_prob`=0.005; and `interspecies_mate_rate`=0.01.

type of neuroevolution applied (NEAT or rtNEAT), and to (ii) the evaluation strategy used (ε-Greedy, ε-Greedy-Improved, Softmax, or Interval-based). The post-hoc procedures have been applied to identify groups of configurations with similar performance.

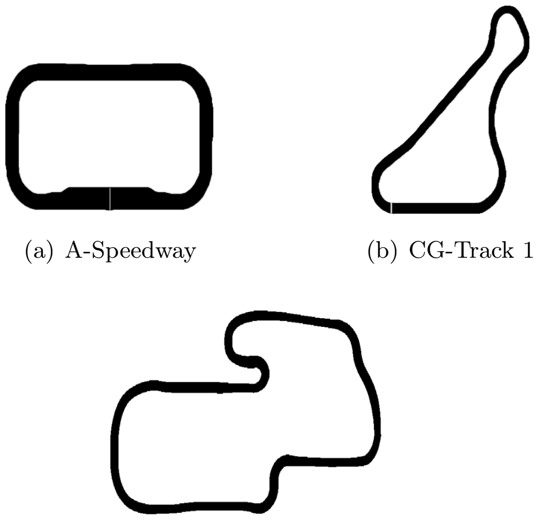
6.4. Experimental Results

We applied on-line neuroevolution using NEAT and rtNEAT to evolve a driver for each one of the three tracks depicted in Figure 7.3. The tracks have increasing difficulty and they are all available in the TORCS distribution. A-Speedway is a simple oval with four *identical* left turns; a driver for this track, to be competitive, just needs to learn how to approach that one turn. CG-Track 1 mainly contains high-speed turns; it is not very difficult, but it requires complex trajectories to achieve low lap times. Ruudskogen is a rather complex and narrow track which contains both fast and slow turns, both on the left and on the right side; moreover the height of the track changes significantly making the control of the car more difficult when approaching some of the turns; accordingly, in third track, drivers must learn sophisticated behaviors for each part of the track to be competitive.

6.4.1. Learning from Scratch.

In the first set of experiments, we applied the on-line versions of NEAT and rtNEAT to evolve a driver starting from a population initialized as usually done in NEAT and rtNEAT, i.e., using the simplest network topology possible. Table 6.1 compares the performance of on-line NEAT/rtNEAT using one of the four selection strategies on the three tracks; statistics are averages over ten runs; best performances are highlighted in bold. Table 6.1a reports the average lap time over *the first 100 laps*; Table 6.1b reports the average lap time over *the first 500 laps*; Table 6.1c reports the average lap time over *the last 100 laps*; and Table 6.1d reports the average lap time of *the best individuals evolved* for each configuration during each run.

Statistical Analysis. For each track, we applied a *two-way analysis of variance* (a two-way ANOVA [23]) to test whether the differences in Table 6.1 are statistically significant *with respect to two factors*, the type of neuroevolution approach used (NEAT or rtNEAT) and the type of evaluation selection used (ε-Greedy, ε-Greedy-Improved, Softmax, and Interval-based). When the tests returned a statistical significance we



(a) A-Speedway (b) CG-Track 1

(c) Ruudskogen

Figure 6.1.: The three tracks used for the experiments reported in this work.

also applied the typical post-hoc procedures (SNK, Tukey, Scheff´e, and Bonferroni) to analyze the differences among the four selection strategies.

Initial 100 Laps. As can be noted from Table 6.1a, in the initial 100 laps, rtNEAT appears to learn faster than NEAT: in fact, the average lap times obtained by rtNEAT are usually lower than the corresponding values obtained by NEAT (apart from a very few exceptions). However, the two-way analysis of variance [23] of the data in Table 6.1a, for the averages over the first 100 laps, shows that the reported differences are not statistically significant. The same analysis shows that, with respect to the selection algorithms, there is no statistically significant difference between ε-Greedy-Improved, Softmax, and Interval-based. Finally, the post-hoc procedures make evident a clear distinction in A-Speedway between ε-Greedy (which is the worst one) and all the other three policies (ε-Greedy-Improved, Softmax, and Interval-based).

Initial 500 Laps. As the learning proceeds, NEAT improves much more than rtNEAT so that the difference between their performances becomes blurred. The statistical analysis of the average lap times over the first 500 laps (Table 6.1b) shows that the difference between NEAT and rt-NEAT is now statistically significant *only* in the most difficult track, Ruudskogen, with a confidence level of the 95%. This result might appear counterintuitive since, in A-Speedway and CG-Track 1, rtNEAT still performs better than NEAT for three out of the four selection algorithms. Note however that, in the same tracks, when ε-Greedy is applied, rtNEAT performs much worse than NEAT so that, on the average, the differences between NEAT and rtNEAT are not statistically significant. With respect to the selection strategies, the data in Table 6.1b confirm and extend what was previously found: there is a significant difference between ε-Greedy and the other three strategies in all the three tracks.

Last 100 Laps. Table 6.1c reports the average lap times over the last 100 (learning) laps. The reported results show that, at the end, the difference between NEAT and rtNEAT is blurred: in 8 out of the 12 cases, the average lap time obtained using NEAT is slightly higher than rtNEAT's one, but this difference is not statistically significant. In the remaining cases, NEAT performs significantly better than rtNEAT. With respect to the selection strategies, the statistical analysis confirms the previous findings showing a statistically significant difference between ε-Greedy and the other three strategies in *all the tracks*.

Testing. During learning, the whole population controls the only car available. Therefore, several networks selected from the population may be used within the same lap so that there is no guarantee that a complete driver has been evolved. Accordingly, at the end of the learning, we tested the best individuals, the champions, evolved during each run for each configuration to assess their final performance on the whole track. For this purpose, each champion drove two laps and we measured its performance as the time taken to complete the second lap. Table 6.1d reports the average lap times obtained by the best ten individuals, the ten champions, evolved for each configuration (one for each one of the ten runs); an "$*$" indicates that one of the champions could not complete the two test laps, i.e., the champion was not able to drive on the entire track but only on a part of it. The results show that, although the drivers are evaluated only on small parts of the track, the vast majority of the best individuals can drive on the whole track reliably. This is not surprising as the best drivers are actually reevaluated several times and thus, overall, are very likely to be tested on the entire track.

The results in Table 6.1d are similar to the ones found for the final 100 laps of the learning phase (Table 6.1c) where the differences between NEAT and rtNEAT are blurred. At the end, when coupled with on-line rtNEAT, ε-Greedy still performs significantly worse than the other three evaluation strategies. However, when on-line NEAT is used, Interval-based performs worse in the most difficult tracks (although this difference is not statistically significant).

6.4.2. Seeded Evolution

We repeated the same set of experiments starting from populations seeded with the best individuals previously evolved for each track. The goal was to test whether, in TORCS, on-line neuroevolution could exploit previous existing knowledge and possibly improve it so as to evolve a better driver. Note that, Taylor et al. [89] studied how to transfer previously evolved knowledge to more complex tasks involving different sensors/actuators models. In our case, the task is the same, to drive competitively, as well as the sensors/actuators model; accordingly, we applied a much simpler approach than the one in [89].

For each configuration, we applied on-line neuroevolution starting from a seeded population and let the car drive for 2000 laps. All the experiments performed demonstrate a behavior similar to the one illustrated by Figure 6.2 where we compare the performance of NEAT (Figure 6.2a) and rtNEAT (Figure 6.2a), in CG-Track 1 using Softmax, when the evolutionary process (i) starts from scratch, (ii) is seeded with a champion of A-Speedway and (iii) is seeded with a champion of Ruudskogen. As the curves show, seeding the population with a champion (significantly) boosts the learning at the beginning. At the end, after 2000 laps, the performance achieved from a seeded population is *at least* as good as the performance achieved from scratch.

Table 6.2 summarizes all the results reporting the average lap times of *the best individuals* evolved in each run with on-line NEAT and rtNEAT; statistics are averages over ten runs; best performances are highlighted in bold. In this case, the two-way ANOVA reports no statistically significant difference between the performance of NEAT and rtNEAT nor among the four evaluation strategies. It is worth noting however that the best performance is obtained when a driver evolved on a complex track is used as a seed to evolve a driver for a simpler track. For instance, when the A-Speedway champion is used to seed the evolution of a driver for CG-Track 1, the final performances are worse than those obtained when using the Ruudskogen champion as a seed.

Note that, our analysis does not take into account the time required to evolve the champions (and thus follows the *target task time scenario* of [87]). However, the goal of this set of experiments was rather limited and way behind the wider scope of transfer learning. In fact, our goal was just to investigate the possibility of reusing previously evolved drivers on

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	142.74 ± 31.66	281.22 ± 38.19	448.81 ± 56.23
	ε-Greedy-Improved	110.14 ± 19.78	273.87 ± 31.26	460.66 ± 44.38
	Softmax	115.66 ± 14.52	248.22 ± 37.11	432.70 ± 46.08
	Interval-based	101.32 ± 24.55	255.81 ± 42.70	380.11 ± 49.89
rtNeat	ε-Greedy	134.37 ± 55.73	293.94 ± 69.44	507.59 ± 104.83
	ε-Greedy-Improved	**82.65 ± 13.16**	244.29 ± 37.27	445.44 ± 84.33
	Softmax	106.11 ± 25.28	224.77 ± 39.73	356.01 ± 65.22
	Interval-based	106.81 ± 13.17	**224.13 ± 42.90**	**349.76 ± 71.88**

(a)

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	76.41 ± 9.60	154.95 ± 13.64	264.02 ± 20.98
	ε-Greedy-Improved	63.01 ± 5.16	134.38 ± 7.98	239.36 ± 10.16
	Softmax	65.07 ± 5.88	135.01 ± 10.61	241.52 ± 15.86
	Interval-based	67.18 ± 8.31	140.64 ± 15.14	222.04 ± 15.49
rtNeat	ε-Greedy	104.93 ± 49.58	233.23 ± 85.26	428.14 ± 121.07
	ε-Greedy-Improved	**51.90 ± 3.76**	**118.41 ± 15.71**	251.54 ± 42.58
	Softmax	60.75 ± 8.52	121.79 ± 10.23	**200.50 ± 21.77**
	Interval-based	60.86 ± 9.44	133.38 ± 26.62	207.95 ± 36.60

(b)

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	39.12 ± 3.03	87.78 ± 11.04	155.83 ± 27.33
	ε-Greedy-Improved	37.36 ± 2.32	73.39 ± 15.28	**125.67 ± 11.54**
	Softmax	37.96 ± 4.79	76.59 ± 6.80	140.75 ± 10.02
	Interval-based	41.83 ± 4.64	83.60 ± 12.93	144.00 ± 9.32
rtNeat	ε-Greedy	81.94 ± 29.02	160.39 ± 44.87	314.40 ± 59.16
	ε-Greedy-Improved	37.09 ± 2.33	**68.40 ± 3.41**	139.19 ± 8.16
	Softmax	36.24 ± 1.29	75.11 ± 6.89	139.64 ± 6.82
	Interval-based	**35.79 ± 0.94**	72.15 ± 5.99	135.61 ± 3.76

(c)

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	31.86 ± 0.99	73.09 ± 13.40	151.34 ± 19.87
	ε-Greedy-Improved	31.33 ± 1.01	60.34 ± 22.63	**110.50 ± 19.57**
	Softmax	31.04 ± 0.46	55.38 ± 9.08	132.57 ± 27.56
	Interval-based	31.20 ± 0.67	92.03 ± 39.83	155.73 ± 29.57
rtNeat	ε-Greedy	(*) 59.08 ± 29.27	(*) 114.95 ± 44.68	(*) 297.63 ± 67.61
	ε-Greedy-Improved	31.17 ± 0.97	**52.05 ± 1.76**	134.23 ± 20.28
	Softmax	**30.63 ± 0.56**	59.86 ± 7.71	136.18 ± 20.18
	Interval-based	30.92 ± 0.91	(*) 73.20 ± 30.13	136.06 ± 19.38

(d)

Table 6.1.: On-line NEAT and rtNEAT applied to TORCS: (a) average lap time after 100 and (b) after 500 laps; (c) final performance computed as the average lap time during the last 100 laps; (d) average performance of the best individuals in the final populations; statistics are averages over ten runs.

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy		69.63 ± 20.52	131.08 ± 17.67
	ε-Greedy-Improved		**55.64 ± 10.32**	121.76 ± 26.32
	Softmax		60.20 ± 14.25	**112.61 ± 10.58**
	Interval-based		87.06 ± 39.79	155.02 ± 46.57
rtNeat	ε-Greedy		83.61 ± 17.72	164.62 ± 51.64
	ε-Greedy-Improved		59.98 ± 21.48	153.51 ± 21.27
	Softmax		(*) 57.96 ± 15.28	137.38 ± 23.76
	Interval-based		(*) 86.98 ± 24.78	154.40 ± 15.86

(Source Track: A-Speedway)

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	30.49 ± 0.68		104.18 ± 21.95
	ε-Greedy-Improved	30.37 ± 0.66		(*) 89.02 ± 3.93
	Softmax	**29.73 ± 0.18**		**85.33 ± 3.63**
	Interval-based	29.80 ± 0.21		102.69 ± 31.34
rtNeat	ε-Greedy	31.43 ± 0.77		86.99 ± 1.81
	ε-Greedy-Improved	29.96 ± 0.37		90.42 ± 18.35
	Softmax	29.85 ± 0.38		93.26 ± 20.83
	Interval-based	30.11 ± 0.41		94.44 ± 20.54

(Source Track: CG-Track-1)

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
Neat	ε-Greedy	30.94 ± 0.60	61.97 ± 11.52	
	ε-Greedy-Improved	30.90 ± 0.63	62.83 ± 15.39	
	Softmax	30.12 ± 0.14	(*) 57.48 ± 11.67	
	Interval-based	30.31 ± 0.38	61.50 ± 9.43	
rtNeat	ε-Greedy	32.12 ± 0.27	70.06 ± 9.03	
	ε-Greedy-Improved	**30.07 ± 0.97**	**48.32 ± 1.41**	
	Softmax	30.24 ± 0.25	64.45 ± 16.99	
	Interval-based	30.50 ± 0.50	(*) 70.97 ± 22.01	

(Source Track: Ruudskogen)

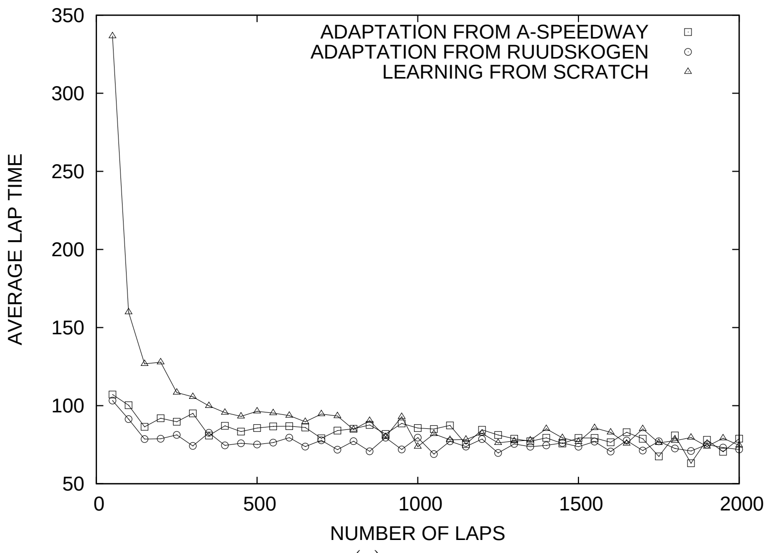
Table 6.2.: On-line NEAT and rtNEAT applied from a seeded population: average performance of the best individuals.

new unseen track as typically happened during many recently organized scientific car racing competitions.

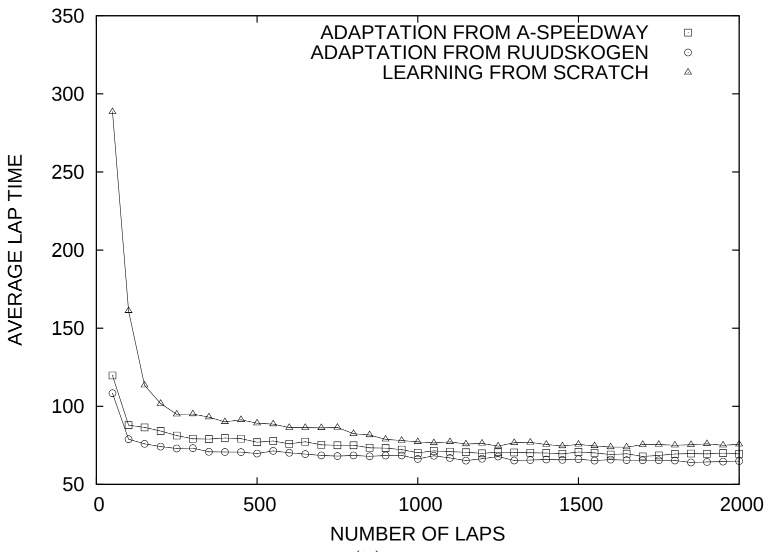
6.4.3. Generalization

At the end, we tested the champions evolved for each one of the three tracks (i.e., one champion for each one of the ten learning runs executed for each configuration on each track) on a large set of the most difficult tracks available in the TORCS distribution. For each track, we let each champion drive two laps and recorded its performance as the time to complete the second lap.

Our results (see Tables 6.4, 6.5, 6.6) show that NEAT evolves drivers which, on the new unseen tracks, are generally faster than the ones evolved by rtNEAT. The usual two-way analysis of variance (ANOVA) shows that such difference is always statistically significant on all the three tracks with a confidence level of 99.9%. With respect to the evaluation strategies, the statistical analysis confirms what was previously found showing that ε-Greedy performs significantly worse than the



(a)



(b)

Figure 6.2.: On-line (a) NEAT and (b) rtNEAT applied to the CG-Track 1 track with Softmax starting from scratch (triangular dots), a population seeded with a champion from A-Speedway (squared dots) and a champion from Ruudskogen (circled dots). Curves are averages over 10 runs.

other three strategies. In addition, the data collected for the champions evolved in Ruudskogen show a statistically significant difference *among all the four selection strategies* suggesting that, in this case, the best driver is the one evolved using ε-Greedy-Improved, followed by Softmax, Interval-based, and ε-Greedy.

We also analyzed the data regarding the number of champions that were able to complete the test process (reported between parentheses in the Tables 6.4, 6.5, 6.6) using a two-way ANOVA. The analysis shows that the number of champions evolved using NEAT, that are capable of completing the test, is significantly higher (with a confidence level of 99%) than the number of champions evolved using rtNEAT, i.e., NEAT evolves more champions which can race on new unseen tracks. With respect to the evaluation strategies, the two-way ANOVA again shows that ε-Greedy performs worse than the other three strategies. In addition, all the four post-hoc tests cluster the four evaluation strategies in three groups: one consisting of ε-Greedy (the worse one), one consisting of Interval-based and Softmax (the middle ones), and one containing ε-Greedy-Improved (the best one).

Interestingly, our analysis also shows that the champions evolved in A-Speedway can generalize significantly less than the drivers evolved in CG-Track 1 and Ruudskogen. In fact, the number of A-Speedway champions that can complete two laps on an unknown track is much smaller than that of CG-Track 1 and Ruudskogen champions and this difference is statistically significant with a confidence level of 99%. This result confirms what was previously found in the experiments with seeded population: it is convenient to use drivers evolved on the most difficult tracks.

6.4.4. Discussion

Our experimental results show that, on-line rtNEAT learns faster at the beginning but, somewhat surprisingly, the difference soon becomes blurred so that at the end there are no statistically significant differences. The two approaches perform similarly also when applied to adapt an old driver to a new track. However, NEAT appears to be more reliable when it comes to generalizing in that it evolves drivers that can complete significantly more laps on unknown tracks. Among the four selection strategies, the analysis suggests that ε-Greedy-Improved and Softmax may be the best ones. Thus, *overall*, our results suggest that on-line NEAT may be the best approach to evolve a driver for TORCS and that NEAT should be coupled with ε-Greedy-Improved or Softmax.

Indeed, when learning from scratch, rtNEAT *at the beginning* (Table 6.1a) performs significantly better than NEAT: since it uses steady-state evolution, rtNEAT explores the solution space less than NEAT (which is generational); accordingly, rtNEAT initially reaches a better performance. However, in the long run, NEAT catches up and at the end NEAT and rtNEAT perform similarly.

The experiments with seeded populations (Section 6.4.2) show that is possible to transfer previous knowledge (previous evolved drivers) to seed the evolution of drivers for different tracks. More interestingly, they show that it is more convenient to transfer the knowledge evolved from difficult tracks to simpler ones (e.g., to use a driver evolved for Ruudskogen to seed a population for CG-Track 1) rather than the other way around. Overall, the experiments with seeded populations show no statistically significant difference between the performance of NEAT and rtNEAT or the four selection strategies. Moreover, at the end, both methods reach a comparable performance (in fact, there is no statistically significant difference at the end).

The final testing performed on many unknown tracks suggests that NEAT can generalize better than rtNEAT in that it generates significantly more controllers capable of driving on difficult unknown tracks.

With respect to the evaluation strategies, all the experiments seem to agree. ε-Greedy is the worst evaluation strategy often leading to a significantly lower performance while the other three evaluation strategies usually perform similarly. Only in some cases, the statistical analysis reported that the performance achieved with Interval-based is significantly worse than the one achieved with ε-Greedy-Improved and Softmax.

6.5. Length of the Evaluation Slot

We repeated the experiments of Section 6.4.1 using short evaluation slots of 500 ticks, medium evaluation slots of 2000 ticks (twice the length used in Section 6.4.1), and long evaluation slots of enough ticks to complete two laps in each track (i.e., 3000 ticks for A-Speedway, 6000 ticks for CG-Track 1, and 12000 ticks for Ruudskogen).

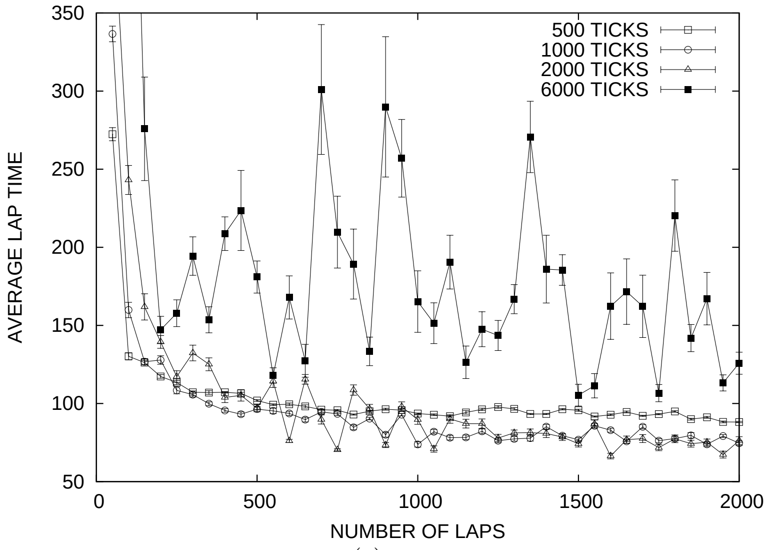
Overall, all the experiments show the same behavior exemplified in Figure 6.3 where we report the learning curves for (a) on-line NEAT and (b) on-line rtNEAT applied to CG-Track 1 using Softmax with different evaluation slot sizes. As can be noted, very short evaluation slots result in faster learning at the beginning but, lead to a final performance worse than that achieved with longer evaluation slots. However, with very long evaluation slots, the learning is initially very slow and, after 2000 laps, the performance is much worse than that achieved with shorter slots. Evaluation slots of 1000 or 2000 ticks (approximately 20 and 40 seconds of actual game time) result in the best performances and reasonable learning speeds at the beginning. In particular, our results show that in general there is no significant difference between the performance using 1000 or 2000 game ticks. Thus, an evaluation slot of 1000 ticks seems to be the best trade-off since it allows for the best learning speed (shorter slots correspond to faster learning) without a statistically significant decrease in the final performance.

6.6. Comparison with Off-Line Neuroevolution

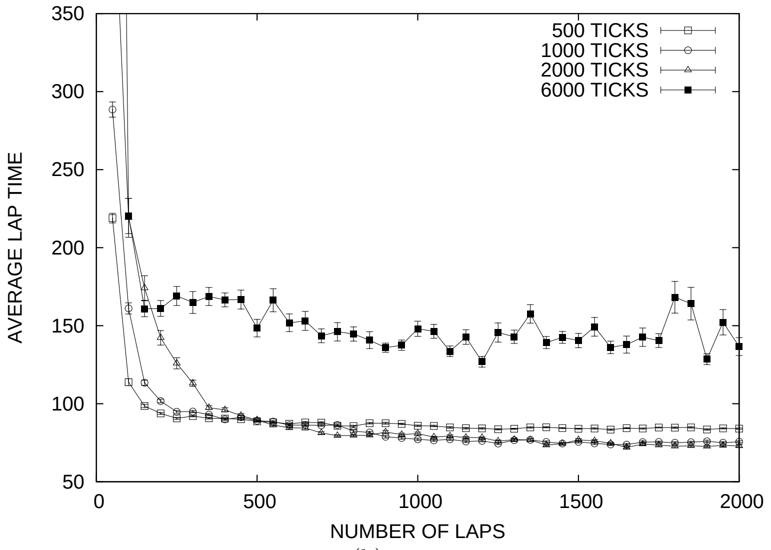
In the last set of experiments, we compared the on-line approaches considered in this work with the typical off-line neuroevolution [48, 4, 59, 33], previously applied to TORCS in [10]. In this case, the evaluations of individuals are performed (i) *independently*, in that, they potentially may be carried out in parallel, and (ii) more accurately, in that, each evaluation involves a complete problem instance. We applied (off-line) NEAT and rtNEAT to evolve drivers for each one of the tracks previously considered. Each individual was evaluated by applying it to drive two laps on the target track. All the evaluations started with the car placed in the same position and ended when either the car completed two laps or a large amount of game ticks had elapsed (to avoid spending to much time when a driver gets stuck). The fitness is the same used for on-line neuroevolution (see Section 6.2 and [10]). Note that, it is not possible to compare on-line and off-line neuroevolution based on the number of laps completed (as we did for the on-line case) since off-line evolution involves multiple evaluations and not just one car racing on the track. Accordingly, to provide a fair comparison, we computed the average number of game ticks used for each setting tested in Section 6.4.1 and stopped off-line neuroevolution when a comparable number of actual game seconds had elapsed.[3]

Table 6.3 compares the average lap times of the best individuals evolved using (i) on-line neuroevolution (taken from Table 6.1d) and (ii) off-line neuroevolution; the statistics are averages over 10 runs. In the simplest track, A-Speedway, off-line neuroevolution performs significantly bet-

[3]In A-Speedway, off-line neuroevolution was stopped after 110000 actual game seconds were elapsed; in CG-Track 1 after 220000 game seconds; in Ruudskogen after 405000 game seconds.



(a)



(b)

Figure 6.3.: On-line neuroevolution applied to the CG-Track 1 track using Softmax and different sizes of the evaluation slot: (a) NEAT; (b) rtNEAT. Curves are averages over 10 runs; bars report the standard error.

NeuroEvolution	Selection Strategy	A-Speedway	CG-Track 1	Ruudskogen
On-line Neat	ε-Greedy	31.86 ± 0.99	73.09 ± 13.40	151.34 ± 19.87
	ε-Greedy-Improved	31.33 ± 1.01	60.34 ± 22.63	**110.50 ± 19.57**
	Softmax	31.04 ± 0.46	55.38 ± 9.08	132.57 ± 27.56
	Interval-based	31.20 ± 0.67	92.03 ± 39.83	155.73 ± 29.57
On-line rtNeat	ε-Greedy	(*) 59.08 ± 29.27	(*) 114.95 ± 44.68	(*) 297.63 ± 67.61
	ε-Greedy-Improved	31.17 ± 0.97	**52.05 ± 1.76**	134.23 ± 20.28
	Softmax	30.63 ± 0.56	59.86 ± 7.71	136.18 ± 20.18
	Interval-based	30.92 ± 0.91	(*) 73.20 ± 30.13	136.06 ± 19.38
Off-line Neat		30.21 ± 0.42	55.10 ± 5.51	(*) 145.45 ± 24.08
Off-line rtNeat		**29.73 ± 0.22**	59.24 ± 8.01	(*) 144.09 ± 31.59

Table 6.3.: Off-line and On-line neuroevolution applied to TORCS: average performance of the best individuals in the final populations; statistics are averages over ten runs.

ter than on-line NEAT and off-line rtNEAT at a 99% confidence level. As the track difficulty increases, in CG-Track 1, the performance of on-line approaches increases and the difference becomes more blurred: off-line neuroevolution only outperforms on-line approaches using ε-Greedy which proved to be the worst performing on-line approach. In the most difficult track, Ruudskogen, on-line NEAT using ε-Greedy-Improved performs significantly better than off-line neuroevolution (with a confidence of the 99%) while on-line rtNEAT using ε-Greedy (the worst combination possible for on-line neuroevolution) performs significantly worse than off-line NEAT and rtNEAT.

6.7. Summary

We applied on-line neuroevolution to evolve drivers for TORCS. We focused on the two major on-line neuroevolution approaches (on-line NEAT [108] and on-line rtNEAT [68]) and on the four evaluation selection strategies introduced in the literature (ε-Greedy, Softmax, Interval-based [108] and ε-Greedy-Improved [9]). We considered eight methods (each one combining one neuroevolution approach with an evaluation strategy). We performed three sets of experiments to compare the methods in terms of learning capabilities, adaptation, and generalization. We also performed an experiment to study the influence of the length of the evaluation timeslot on the overall performance. At the end, we compared the on-line approaches considered with the most typical off-line neuroevolution.

In the first set of experiments, we applied the eight methods to evolve a driver for three different tracks of increasing difficulty and compared their performance at the beginning of the learning phase and at the end of it. When learning to drive from scratch, rtNEAT initially performs significantly better than NEAT; as the learning proceeds, NEAT catches up and, at the end, both methods perform similarly with no statistically significant difference reported.

The second set of experiments focused on knowledge transfer. The champions evolved for each track were used to seed the evolution of drivers for different tracks. The results suggest that is convenient to exploit previous knowledge: the new seeds speed up the learning process at the beginning and result in an overall improvement of the final performance. More interestingly, our results suggest that knowledge should be transferred from the drivers evolved on more difficult tracks.

The third set of experiments was aimed at testing the generalization capabilities of the eight approaches. The results suggest that NEAT can generalize more, producing (significantly more) drivers capable of completing two laps on difficult unknown tracks.

With respect to the evaluation strategy, all the experiments performed basically agree: ε-Greedy is significantly worse than ε-Greedy-Improved, Softmax, and Interval-based, which perform similarly. This results confirm the findings of Whiteson and Stone [108], who also noted that, on the typical reinforcement learning benchmarks, ε-Greedy performed worse. In addition, as reported in [108], also our analysis shows no statistically significant difference between Softmax and Interval-based. In particular, with respect to the generalization capabilities, the results also show a difference in the performance of ε-Greedy-Improved, Softmax and Interval-based, with a cluster containing what appear to be the *best technique* (ε-Greedy-Improved) and a cluster containing (Softmax and Interval-based).

We performed another set of experiments to study how the length of the evaluation slot could affect the final performance. Our results show that, short evaluation slots can boost the learning at the beginning, but might lead to the convergence to a suboptimal driver. Long evaluation slots (e.g., similar to the one which would be used for off-line neuroevolution) result in an extremely slow learning at the beginning and very poor drivers at the end. Evaluation slots of 1000 or 2000 ticks appears to be the best choice in that they provide a reasonable learning speed and lead to a similar performance. In particular, we believe that 1000 ticks might be the best compromise providing the best learning speed initially, with no statistically significant decrease in the final performance.

In the last set of experiments, we compared the on-line approaches considered here with typical off-line neuroevolution. Our results suggest that on-line neuroevolution can be competitive, notwithstanding the several challenges that on-line neuroevolution has to face. In fact, off-line neuroevolution performs slightly better than on-line only in the easiest track (A-Speedway) while, as the track difficulty increases, the differences become blurred and, on the most difficult track, on-line approaches may outperform off-line neuroevolution. Overall, our results showed that, although a general solution to tackle the several challenges posed by realistic game platforms to on-line learning is still missing, it is possible to adapt existing approaches to obtain interesting performances.

Test Track	Selection Strategy	NEAT ($\mu \pm \sigma$)	rtNEAT ($\mu \pm \sigma$)
alpine-1	ε-Greedy	260.54 ± 118.15 (5)	— (0)
	ε-Greedy-Improved	236.00 ± 51.13 (7)	276.81 ± 157.98 (7)
	Softmax	220.13 ± 95.06 (3)	163.58 ± 10.72 (5)
	Interval-based	206.31 ± 51.08 (5)	310.19 ± 111.59 (2)
alpine-2	ε-Greedy	168.65 ± 18.27 (3)	284.36 ± 0.00 (1)
	ε-Greedy-Improved	231.12 ± 112.84 (5)	141.50 ± 19.59 (5)
	Softmax	168.76 ± 71.42 (3)	163.19 ± 19.99 (2)
	Interval-based	162.18 ± 50.40 (5)	— (0)
e-track-1	ε-Greedy	251.97 ± 69.38 (9)	338.74 ± 39.44 (3)
	ε-Greedy-Improved	255.03 ± 66.41 (7)	214.63 ± 51.70 (7)
	Softmax	238.78 ± 42.87 (5)	256.87 ± 47.86 (7)
	Interval-based	224.05 ± 37.71 (8)	249.21 ± 44.19 (5)
e-track-2	ε-Greedy	280.19 ± 87.57 (2)	339.87 ± 0.00 (1)
	ε-Greedy-Improved	328.99 ± 0.00 (1)	271.21 ± 50.51 (4)
	Softmax	— (0)	279.32 ± 0.00 (1)
	Interval-based	— (0)	364.15 ± 0.00 (1)
e-track-3	ε-Greedy	287.92 ± 99.13 (7)	309.47 ± 68.56 (4)
	ε-Greedy-Improved	216.21 ± 36.78 (6)	201.03 ± 40.90 (7)
	Softmax	250.83 ± 31.63 (7)	263.42 ± 76.56 (6)
	Interval-based	176.81 ± 17.74 (6)	276.81 ± 115.09 (6)
e-track-4	ε-Greedy	403.87 ± 123.13 (7)	569.66 ± 142.25 (4)
	ε-Greedy-Improved	375.14 ± 93.63 (9)	292.69 ± 47.83 (7)
	Softmax	376.91 ± 101.42 (5)	382.36 ± 88.56 (6)
	Interval-based	322.25 ± 46.29 (8)	408.96 ± 107.28 (3)
e-track-6	ε-Greedy	375.74 ± 82.58 (2)	— (0)
	ε-Greedy-Improved	401.83 ± 0.00 (1)	345.25 ± 45.95 (3)
	Softmax	— (0)	— (0)
	Interval-based	296.12 ± 0.00 (1)	277.53 ± 0.00 (1)
forza	ε-Greedy	262.76 ± 89.89 (4)	458.56 ± 103.31 (7)
	ε-Greedy-Improved	274.26 ± 30.45 (7)	277.38 ± 48.07 (7)
	Softmax	328.81 ± 119.77 (7)	375.33 ± 98.19 (7)
	Interval-based	327.16 ± 141.17 (8)	306.18 ± 63.84 (4)
g-track-1	ε-Greedy	120.12 ± 52.80 (9)	200.69 ± 37.06 (3)
	ε-Greedy-Improved	120.27 ± 44.21 (10)	106.14 ± 22.54 (8)
	Softmax	135.37 ± 61.48 (10)	113.00 ± 49.52 (8)
	Interval-based	120.25 ± 31.30 (10)	117.00 ± 29.57 (6)
g-track-2	ε-Greedy	172.11 ± 61.49 (7)	259.56 ± 49.01 (5)
	ε-Greedy-Improved	169.58 ± 42.42 (8)	131.47 ± 20.17 (7)
	Softmax	150.08 ± 43.13 (7)	154.28 ± 44.60 (6)
	Interval-based	143.35 ± 39.10 (8)	191.50 ± 65.99 (6)
g-track-3	ε-Greedy	223.03 ± 70.42 (7)	273.48 ± 14.85 (2)
	ε-Greedy-Improved	225.26 ± 70.90 (6)	205.16 ± 55.28 (5)
	Softmax	173.09 ± 17.75 (5)	208.70 ± 49.98 (6)
	Interval-based	203.26 ± 69.48 (5)	230.83 ± 15.99 (2)
ole-road-1	ε-Greedy	360.64 ± 91.97 (7)	624.40 ± 44.80 (2)
	ε-Greedy-Improved	346.33 ± 74.46 (8)	363.50 ± 59.95 (8)
	Softmax	377.35 ± 127.89 (8)	350.89 ± 156.47 (6)
	Interval-based	357.63 ± 111.94 (6)	391.85 ± 129.64 (3)
ruudskogen	ε-Greedy	259.34 ± 79.69 (5)	149.69 ± 0.00 (1)
	ε-Greedy-Improved	267.07 ± 67.02 (7)	230.70 ± 81.21 (7)
	Softmax	277.62 ± 69.64 (7)	229.16 ± 29.18 (5)
	Interval-based	241.82 ± 61.56 (6)	275.92 ± 23.06 (3)
wheel-1	ε-Greedy	279.95 ± 111.43 (9)	330.08 ± 88.50 (5)
	ε-Greedy-Improved	218.53 ± 41.69 (9)	205.67 ± 25.76 (8)
	Softmax	219.24 ± 38.84 (10)	259.63 ± 101.15 (9)
	Interval-based	193.85 ± 37.54 (8)	229.29 ± 51.75 (6)
wheel-2	ε-Greedy	359.86 ± 72.04 (5)	559.51 ± 39.50 (3)
	ε-Greedy-Improved	408.79 ± 116.08 (7)	390.42 ± 89.45 (8)
	Softmax	425.71 ± 39.08 (3)	455.42 ± 80.84 (6)
	Interval-based	365.01 ± 69.73 (7)	526.87 ± 7.97 (2)

Table 6.4.: Testing the champions of track A-Speedway on other unseen tracks. The table reports the average lap time for each champion on each track with the standard deviation and, between parentheses, the number of evolved champions that were able to complete the two laps.

Test Track	Selection Strategy	NEAT ($\mu \pm \sigma$)	rtNEAT ($\mu \pm \sigma$)
alpine-1	ε-Greedy	299.20 ± 198.56 (7)	303.85 ± 55.28 (3)
	ε-Greedy-Improved	213.94 ± 36.55 (7)	219.50 ± 28.80 (9)
	Softmax	204.66 ± 20.04 (8)	204.10 ± 39.06 (7)
	Interval-based	229.33 ± 71.03 (7)	214.07 ± 10.31 (6)
alpine-2	ε-Greedy	148.83 ± 31.60 (7)	185.56 ± 74.33 (5)
	ε-Greedy-Improved	160.16 ± 43.37 (6)	174.31 ± 65.65 (8)
	Softmax	141.81 ± 13.43 (6)	151.51 ± 27.64 (6)
	Interval-based	186.05 ± 75.69 (7)	150.90 ± 12.88 (6)
e-track-1	ε-Greedy	168.17 ± 53.13 (9)	253.36 ± 79.85 (7)
	ε-Greedy-Improved	152.79 ± 25.49 (7)	169.75 ± 32.12 (8)
	Softmax	157.51 ± 21.27 (9)	193.63 ± 36.36 (7)
	Interval-based	206.51 ± 48.93 (6)	214.42 ± 50.17 (8)
e-track-2	ε-Greedy	217.22 ± 40.16 (6)	329.91 ± 47.39 (4)
	ε-Greedy-Improved	240.59 ± 60.98 (6)	197.62 ± 44.28 (8)
	Softmax	194.37 ± 33.05 (8)	235.38 ± 33.36 (7)
	Interval-based	265.67 ± 62.00 (4)	259.89 ± 57.26 (5)
e-track-3	ε-Greedy	197.03 ± 69.46 (9)	314.80 ± 111.38 (8)
	ε-Greedy-Improved	177.41 ± 23.94 (9)	169.85 ± 21.66 (9)
	Softmax	163.05 ± 22.30 (9)	183.22 ± 41.27 (10)
	Interval-based	254.12 ± 99.43 (10)	215.03 ± 71.87 (7)
e-track-4	ε-Greedy	287.51 ± 61.01 (9)	309.64 ± 80.42 (7)
	ε-Greedy-Improved	217.97 ± 60.05 (10)	198.69 ± 52.11 (10)
	Softmax	203.24 ± 58.27 (10)	207.36 ± 52.76 (10)
	Interval-based	229.67 ± 74.04 (10)	305.51 ± 111.50 (9)
e-track-6	ε-Greedy	212.10 ± 66.39 (3)	320.39 ± 75.96 (6)
	ε-Greedy-Improved	191.75 ± 58.01 (6)	225.96 ± 71.06 (7)
	Softmax	212.47 ± 45.40 (6)	220.04 ± 63.93 (8)
	Interval-based	227.87 ± 46.03 (7)	279.84 ± 120.50 (5)
forza	ε-Greedy	193.74 ± 11.48 (8)	311.57 ± 109.16 (7)
	ε-Greedy-Improved	192.76 ± 37.71 (8)	238.41 ± 67.02 (9)
	Softmax	229.11 ± 76.68 (7)	214.38 ± 36.23 (8)
	Interval-based	261.42 ± 55.61 (8)	293.07 ± 86.04 (6)
g-track-1	ε-Greedy	73.09 ± 13.40 (10)	114.95 ± 44.68 (9)
	ε-Greedy-Improved	60.34 ± 22.63 (10)	52.05 ± 1.76 (10)
	Softmax	55.38 ± 9.08 (10)	59.86 ± 7.71 (10)
	Interval-based	92.03 ± 39.83 (10)	73.20 ± 30.13 (8)
g-track-2	ε-Greedy	123.44 ± 44.69 (10)	168.56 ± 39.69 (7)
	ε-Greedy-Improved	100.45 ± 21.72 (9)	105.63 ± 26.52 (10)
	Softmax	105.96 ± 19.88 (10)	121.74 ± 28.48 (10)
	Interval-based	115.52 ± 36.10 (8)	130.68 ± 39.76 (9)
g-track-3	ε-Greedy	176.42 ± 32.21 (7)	224.03 ± 71.71 (7)
	ε-Greedy-Improved	170.74 ± 68.66 (8)	163.33 ± 40.07 (10)
	Softmax	163.97 ± 53.88 (8)	177.67 ± 63.11 (9)
	Interval-based	168.58 ± 48.68 (8)	174.49 ± 16.07 (6)
ole-road-1	ε-Greedy	309.83 ± 109.62 (9)	374.49 ± 138.31 (7)
	ε-Greedy-Improved	272.69 ± 76.28 (8)	268.03 ± 59.51 (9)
	Softmax	272.28 ± 68.90 (8)	351.67 ± 143.24 (8)
	Interval-based	331.53 ± 117.33 (9)	341.41 ± 118.52 (6)
ruudskogen	ε-Greedy	175.18 ± 41.47 (10)	196.41 ± 32.48 (6)
	ε-Greedy-Improved	147.00 ± 21.28 (9)	170.31 ± 42.19 (10)
	Softmax	149.19 ± 36.14 (9)	167.65 ± 26.04 (9)
	Interval-based	192.28 ± 37.03 (10)	209.33 ± 69.02 (8)
wheel-1	ε-Greedy	174.39 ± 39.01 (10)	251.10 ± 94.08 (9)
	ε-Greedy-Improved	157.88 ± 25.21 (9)	146.84 ± 23.14 (10)
	Softmax	149.70 ± 26.88 (10)	162.52 ± 20.59 (10)
	Interval-based	177.48 ± 42.24 (10)	188.89 ± 62.54 (7)
wheel-2	ε-Greedy	284.33 ± 65.01 (10)	439.37 ± 87.77 (5)
	ε-Greedy-Improved	269.42 ± 100.93 (9)	247.26 ± 44.12 (10)
	Softmax	296.79 ± 91.99 (10)	277.66 ± 57.11 (9)
	Interval-based	324.04 ± 98.39 (10)	336.74 ± 122.25 (8)

Table 6.5.: Testing the champions of track CG-Track-1 on other unseen tracks. The table reports the average lap time for each champion on each track with the standard deviation and, between parentheses, the number of evolved champions that were able to complete the two laps.

Test Track	Selection Strategy	NEAT ($\mu \pm \sigma$)	rtNEAT ($\mu \pm \sigma$)
alpine-1	ε-Greedy	184.73 ± 21.46 (7)	343.88 ± 135.86 (2)
	ε-Greedy-Improved	215.95 ± 14.57 (9)	180.64 ± 17.27 (9)
	Softmax	211.63 ± 35.64 (10)	211.29 ± 30.63 (8)
	Interval-based	248.12 ± 139.08 (7)	235.89 ± 82.29 (10)
alpine-2	ε-Greedy	135.29 ± 23.08 (7)	203.28 ± 94.59 (4)
	ε-Greedy-Improved	143.11 ± 18.03 (9)	124.16 ± 13.04 (9)
	Softmax	128.66 ± 9.83 (9)	156.24 ± 61.37 (8)
	Interval-based	136.22 ± 25.46 (9)	157.26 ± 63.23 (9)
e-track-1	ε-Greedy	202.77 ± 79.32 (8)	331.61 ± 10.18 (2)
	ε-Greedy-Improved	140.71 ± 30.07 (9)	162.59 ± 35.52 (8)
	Softmax	165.40 ± 35.97 (10)	183.93 ± 19.14 (8)
	Interval-based	160.22 ± 12.69 (5)	181.56 ± 20.40 (10)
e-track-2	ε-Greedy	261.40 ± 31.31 (4)	— (0)
	ε-Greedy-Improved	195.93 ± 62.66 (9)	205.93 ± 39.49 (6)
	Softmax	190.26 ± 51.08 (9)	213.82 ± 27.09 (5)
	Interval-based	209.85 ± 14.32 (5)	192.68 ± 33.42 (5)
e-track-3	ε-Greedy	180.81 ± 26.96 (10)	354.14 ± 124.12 (4)
	ε-Greedy-Improved	148.21 ± 33.67 (9)	178.45 ± 32.53 (7)
	Softmax	177.38 ± 34.19 (9)	186.09 ± 24.51 (10)
	Interval-based	192.29 ± 74.77 (9)	175.19 ± 23.79 (10)
e-track-4	ε-Greedy	285.93 ± 48.84 (8)	357.71 ± 116.45 (4)
	ε-Greedy-Improved	158.92 ± 21.91 (9)	237.09 ± 71.89 (9)
	Softmax	181.99 ± 37.77 (9)	233.52 ± 62.42 (8)
	Interval-based	245.00 ± 53.77 (8)	244.45 ± 63.17 (10)
e-track-6	ε-Greedy	192.88 ± 23.67 (2)	183.94 ± 0.00 (1)
	ε-Greedy-Improved	155.02 ± 27.01 (9)	201.02 ± 32.32 (6)
	Softmax	206.81 ± 65.15 (7)	207.68 ± 23.79 (6)
	Interval-based	225.98 ± 46.61 (4)	213.93 ± 33.15 (7)
forza	ε-Greedy	216.49 ± 50.41 (10)	357.84 ± 99.18 (8)
	ε-Greedy-Improved	186.87 ± 37.34 (10)	193.37 ± 28.93 (10)
	Softmax	194.83 ± 29.21 (10)	204.00 ± 28.69 (10)
	Interval-based	239.13 ± 69.77 (10)	221.12 ± 61.09 (10)
g-track-1	ε-Greedy	85.14 ± 18.00 (10)	137.41 ± 38.64 (6)
	ε-Greedy-Improved	61.98 ± 14.23 (10)	66.80 ± 7.91 (10)
	Softmax	66.61 ± 13.16 (10)	83.66 ± 26.65 (10)
	Interval-based	91.83 ± 19.01 (10)	75.73 ± 19.86 (10)
g-track-2	ε-Greedy	133.29 ± 37.71 (10)	220.30 ± 79.96 (6)
	ε-Greedy-Improved	92.47 ± 20.78 (10)	103.57 ± 18.85 (10)
	Softmax	111.70 ± 33.27 (9)	115.28 ± 34.49 (9)
	Interval-based	124.63 ± 21.16 (8)	111.60 ± 40.88 (8)
g-track-3	ε-Greedy	166.30 ± 22.56 (10)	230.16 ± 0.00 (1)
	ε-Greedy-Improved	125.70 ± 29.63 (10)	147.18 ± 35.02 (8)
	Softmax	125.06 ± 24.52 (10)	175.01 ± 54.81 (10)
	Interval-based	133.05 ± 28.40 (8)	157.10 ± 50.09 (10)
ole-road-1	ε-Greedy	279.90 ± 74.83 (10)	485.27 ± 120.26 (4)
	ε-Greedy-Improved	222.22 ± 34.62 (10)	228.36 ± 28.94 (9)
	Softmax	211.02 ± 28.98 (10)	269.58 ± 75.37 (10)
	Interval-based	280.09 ± 61.33 (10)	246.49 ± 33.37 (10)
ruudskogen	ε-Greedy	151.34 ± 19.87 (10)	297.63 ± 67.61 (7)
	ε-Greedy-Improved	110.50 ± 19.57 (10)	134.23 ± 20.28 (10)
	Softmax	132.57 ± 27.56 (10)	136.18 ± 20.18 (10)
	Interval-based	155.73 ± 29.57 (10)	136.06 ± 19.38 (10)
wheel-1	ε-Greedy	170.16 ± 26.34 (9)	305.89 ± 125.85 (5)
	ε-Greedy-Improved	136.46 ± 16.38 (10)	144.26 ± 26.06 (10)
	Softmax	156.71 ± 45.43 (10)	152.00 ± 27.22 (10)
	Interval-based	156.15 ± 31.98 (10)	150.31 ± 19.77 (10)
wheel-2	ε-Greedy	289.18 ± 82.89 (10)	363.35 ± 105.20 (3)
	ε-Greedy-Improved	224.90 ± 70.25 (9)	240.01 ± 43.11 (10)
	Softmax	239.37 ± 35.06 (10)	300.44 ± 127.77 (10)
	Interval-based	294.20 ± 47.31 (9)	271.15 ± 89.24 (10)

Table 6.6.: Testing the champions of track Ruudskogen on other unseen tracks. The table reports the average lap time for each champion on each track with the standard deviation and, between parentheses, the number of evolved champions that were able to complete the two laps.

7. Transfer Learning of Driving Behaviors

In this chapter, we investigate how to transfer driving behaviors from TORCS to VDrift, two well known open-source racing games. We apply a neuroevolutionary learning approach and we compare three different methods of transfer learning: (i) the direct transfer of the learned behaviors; (ii) the transfer of the learning process; (iii) the transfer of both the behaviors and the process.

7.1. Approach

The increasing complexity of computer games has quickly incremented the time necessary for the game development. To boost the development process it is possible to use third-party packages like graphics engines and physics engines. In this context, it became important to have some techniques to speed-up also the development of the Artificial Intelligence of the game, i.e., the behaviors of the non-player characters . An interesting approach is to re-use an existing AI already developed in a similar game. This type of problem is part of the area known as Transfer Learning. In this work, we show how transfer learning can be applied to exploit the knowledge gathered in one *source domain* for accelerating the learning in a *target domain*. We take into account two racing games with many differences related to the physics engine and the game dynamics: TORCS and VDrift. In particular we want to exploit the driving behaviors evolved for TORCS (see chapter 5) to accelerate the learning in the racing game VDrift.

In general, transfer learning, involves the transfer of policies or functions that represent a particular behavior. In this work, we use the term transfer learning in a broader sense since we study also how it is possible transfer the fitness function and the evaluation mechanism across different domains. In our experimental analysis we compare three different approaches for transfer learning: (i) transfer of the learned behaviors, by copying the existing behaviors from the source domain to the target domain; (ii) transfer of the learning process, by replicating in the target domain the evaluation mechanism and the fitness designed for the source domain; (iii) transfer of both the behaviors and the process, by replicating the evolutionary process of the source domain and by performing an incremental learning using the source behavior as a seed.

7.2. Target Domain

In this chapter, we will apply transfer learning using the game TORCS as source domain. As target domain, we need to choose a racing game which is quite different from TORCS. However, the target game should be open-source since we need to implement the same car sensor model used in TORCS. In the following of this section we review all the open-source car racing games currently available and we finally motivate why we focus on the game VDrift.

7.2.1. VDrift

VDrift [106] is one of the best open-source car racing game. The developments was started in 2005 by *Joe Venzon.* Its main main features are represented by the high quality 3D visualization and by a good playability. For the physics simulation it uses Bullet[1], an open source physics engine featuring 3D collision detection, soft body dynamics, and rigid body dynamics (also used in commercial games like for example *Grand Theft Auto IV*[2]). Thus, the physics simulation is very accurate. Particular attention is devoted to the simulation of the loss of traction that result in the *drift.* The only aspect missing is a model for the damage suffered by the cars. The community around VDrift is smaller that the TORCS one and it is hard to find additional resources.

7.2.2. Others

Among the other open-source car racing games we cite *Speed Dreams*, *RARS*, and *Vamos.*

Speed Dreams [2] is a fork of TORCS, which aims to implement exciting new features to make a more enjoyable game for the player, as well as constantly improving visual and physics realism. The main claim of

[1]http://bulletphysics.org/
[2]http://www.rockstargames.com/IV/

Figure 7.1.: A screenshot from VDrift.

the Speed Dreams project is a faster release process (1 or 2 releases per year) thanks to a more active community of users.

RARS (Robot Auto Racing Simulator) is an old 3D racing simulator designed to enabled pre-programmed AI drivers to race against one another. RARS was used as the base for TORCS.

Vamos is an automotive simulation framework with an emphasis on thorough physical modeling. It was used as a base for developing VDrift.

7.2.3. Discussion

The goal of this work is to compare the results already available for TORCS with a similar game type but with enough differences in the game dynamics and in the physics simulation. We discarded RARS since its code was used as a base for TORCS. We discarded Speed Dreams as it is a fork of TORCS and share the same physics engine (even if there can be some upgrades). The only remaining choices are Vamos and VDrift. At this point, VDrift was a quite obvious choice since it is a more sophisticated version of Vamos.

TORCS and VDrift have completely different physics engines: the first one has a physics engine specifically designed for that racing game; the latter use a general purpose 3D physics engine (Bullet). Also the simulation granularity is different: TORCS updates the game state 50 times every game second while VDrift updates the game state 100 times

Table 7.1.: Comparison between the VDrift-AI and one of the best bot of TORCS.

Track	VDrift-AI(s)	Torcs-AI (s)	Overhead
Dirt-3	100.87	65.5	35%
Ruudskogen	92.67	65.9	29%
Suzuka-2005	170.15	114.6	33%

per seconds. These differences lead also to different game dynamics in terms of drivability, steering sensitivity and achievable speeds. Other differences are introduced by the fact that the car models available for VDrift are different from the ones available for TORCS.

To have an idea of the differences between TORCS and VDrift we compared the performance of the bots available in each game driving in the same tracks. Some tracks are available for both games since they use the same file format (a common standard for 3D objects called AC3D [3]). Table 7.1 reports the lap times achieved by the bot Inferno in TORCS and the bot VDrift-AI in VDrift. It is possible to see that the lap times achieved in VDrift are higher than in TORCS with an average overhead of 32%. Part of this overhead is due to the different programmed policy used by the bots. The different driving policy is not enough to justify such a big difference. From a deeper analysis it seems that the overhead is mainly due to the different accelerations achievable and so to the maximum speed and the braking capabilities.

7.3. Experimental Results

In this section, we present three methodologies for transferring the knowledge acquired in one domain (TORCS) for learning a driving policy in another domain (VDrift). For each methodology, we present the experimental results and finally we discuss which is the approach with the best trade-off.

In this work, we focus in the case in which both the source and the target domain have the same sensor and action space. Thus, in VDrift, we implemented the same car sensors presented in Chapter 5 for TORCS. In figure 7.2, it is possibile to see a graphical rendering of the rangefinder used in VDrift. Regarding the action space, no modification was needed

[3]http://www.inivis.com/ac3d/man/ac3dfileformat.html

since both TORCS and VDrift use the typical effectors of a car: throttle, brakes, steering wheel and gear shift.

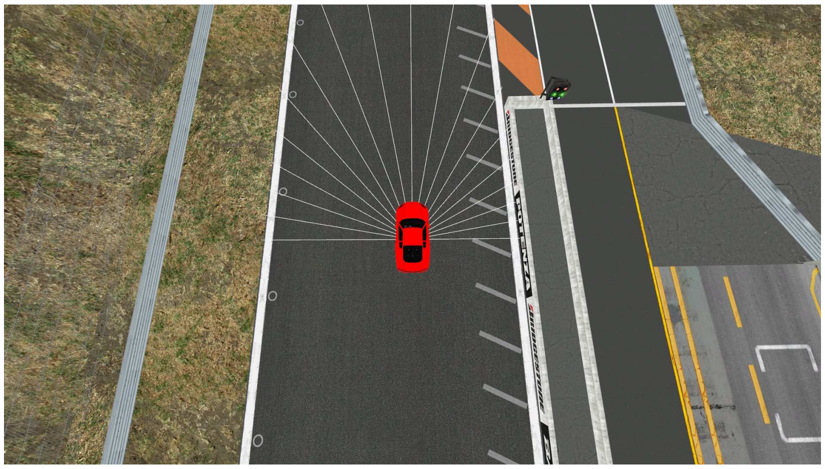
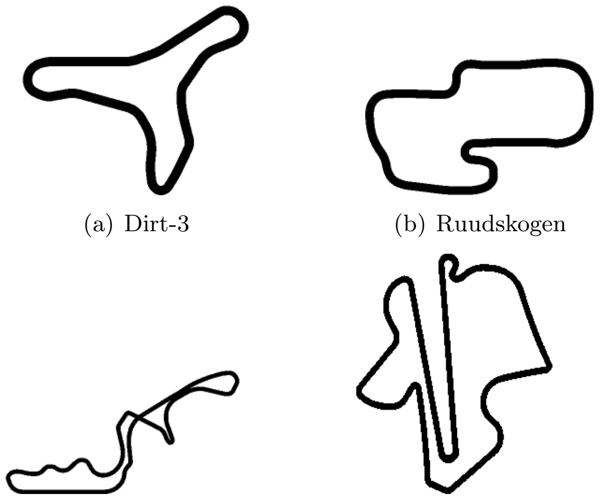


Figure 7.2.: A screenshot of the rangefinder sensor.

7.3.1. Using an existing Driver

The simplest way for transferring the knowledge from TORCS to VDrift is to copy the given behavior as it is, from the source domain to the target domain. In our case the driving behavior is represented through a neural network that can be easily loaded and simulated in another game. Among the drivers evolved for TORCS in chapter 5 we selected the neural network with the best driving performance (i.e., with the highest fitness). This neural network is the core of the driver that resulted to be one of the best entries of the Simulated Car Racing Championship 2009 [42] and the winner of the 2008 IEEE CIG Simulated Car Racing competition. Thus we will refer to this particular neural network with the name *Champion2009*. To allow this neural network to work properly, we implemented in VDrift the same driving aids presented in chapter 5 for TORCS: a recovery policy, a simple gear shifting policy, and a policy which forces the car to accelerate at full speed when the frontal sensor detects a straight.

To test the performance of Champion2009 when driving in VDrift we selected 4 tracks with increasing difficulty and the car *Ferrari 360 Modena*. The 4 tracks used for the experiments of this work are reported in Figure 7.3. The tracks Dirt3, Ruudskogen, Suzuka are available both for TORCS and VDrift while the track Sepang is available only for VDrift.



(a) Dirt-3 (b) Ruudskogen

(c) Suzuka (d) Sepang

Figure 7.3.: VDrift tracks used for the experiments reported in this work.

Table 7.2 reports the performance of the Champion2009 when drives in VDrift. The performance are measured as the lap time in seconds achieved during the second lap. As reference, Table 7.2 reports also the lap time achieved by the best AI available for VDrift. In the track Dirt-1, the Champion2009 drives well and its lap time is similar to the one achieved by VDrift-AI. In Ruudskogen, the performance is worse but the Champion2009 is able to drive well for the entire track. In the most difficult tracks, Suzuka and Sepang, the car goes outside the track in some critical points. The recovery policy is used to take the car on track. The lap times are much higher since each time the recovery policy is activated, it takes around 20 seconds to bring the car on track. If we consider that there are many differences between the dynamics of the two games (as pointed out in Section 7.2), it is surprising how well the driver behaves in VDrift: it drives well on two medium difficulty tracks and for a large part of the two most difficult tracks.

7.3.2. Evolution From Scratch

In general, applying neuroevolution to learn a target behavior in a given game is not an easy task. It requires a good knowledge of the problem in order to design an effective evaluation process of the individuals. This usually involves: (i) the definition of a simulation in the game to extract some measures of the target task; (ii) the definition of the fitness which

Table 7.2.: Lap times of Champion2009 versus VDrift-AI in VDrift.

Track	*Champion2009* (s)	*VDrift-AI* (s)
Dirt-3	106.55	100.87
Ruudskogen	117.50	92.67
Suzuka-2005	227.47	170.15
Sepang	271.47	159.37

is a combination of the extracted features. Finally, it may be necessary to tune the parameters of evolutionary algorithm, that in some cases can affect drastically the final performances. This activity, is usually a trial and error process that is quite time consuming. In this second approach we want to exploit the knowledge acquired for TORCS to speed-up the design of the evolutionary process for learning a driver in VDrift. Thus, the goal is to understand if the neuroevolutionary approach proposed for TORCS is general enough and can be applied to another game without any tuning. In this context, the term Transfer Learning is overloaded since it is usually associated to transfer a policy rather than to the design of the learning experiment and the fitness function.

In this experiment we replicated the same experimental setup and the same fitness used in chapter 5 without any tuning to the coefficient of the formula or to the parameters of the evolutionary algorithm. To train the driving behavior we evolved a population of 100 networks for 150 generations with the standard C++ implementation of NEAT [83]. The performance of each neural network is evaluated using a simulation in VDrift. During the simulation, the neural network is used to control the car to drive on a given track. The evaluation ends when the a lap is completed or when the simulation time is greater than a certain threshold. When the simulation is over, the fitness of the neural network is computed as follows:

$$F = C_1 - T_{out} + C_2 \cdot \bar{s} + d,$$

where T_{out} is the number of game tics the car is outside the track; $\bar{s}$ is the average speed (meters for game tic) during the evaluation; d is the distance (meters) raced by the car during the evaluation; C_1 and C_2 are two constants introduced respectively to make sure that the fitness is positive and to scale the average speed term. C_1 was set 1000 like in chapter 5. Instead C_2 was set to 6000, the double of the value used in chapter 5, just because TORCS engine simulates 50 tics per second while

Table 7.3.: Evolution from scratch in VDrift.

Track	*Laptime (s)*	*Fitness*	*Outside (s)*	*Raced*
Dirt-3	111.96	4412.10	0.09	100%
Ruudskogen	110.60	5835.08	0.00	100%
Suzuka-2005	256.92	8167.85	1.04	100%
Sepang	403.68	7342.87	0.22	100%

VDrift simulate 100 tics per second.

Table 7.3 reports the results of the evolution performed in four different tracks. For each experiment we report the performance of the best driver evolved. In particular we report the fitness, the lap time, the seconds spent outside the track and percentage of track covered by the driver. From the results, it is possible to see that in every track was possible to evolve a driver that can cover the entire track without going outside. The lap times are very similar to the previous case. The only exception is track *Sepang* where to avoid going off-road, the evolution generated a very safe and slow driver. Thus, the experimental design presented for TORCS worked without any modification for VDrift and allowed to avoid a trial and error activity to setup an effective learning experiment.

7.3.3. Adapting an Existing Driver

The third approach that we investigated is somehow a combination of the previous approaches. We take the neural network evolved in TORCS and we apply another run of neuroevolution to adapt that behavior to the new dynamics of VDrift. To do so, we follow the same approach outlined in the evolution from scratch with the same experimental setup, the same evaluation mechanism and the same fitness. While in the evolution from scratch the initial population is random, in this case the initial population is seeded with slightly muted copies of the neural network of the Champion2009.

The results are reported in Table 7.4 where we report the fitness of the Champion2009 (the seed), the fitness of best driver evolved and the percentage of the improvement. The evolution is always able to improve the performance of the Champion2009. While in the medium difficulty track the improvement is moderate in the most difficult track the improvement is very high.

Table 7.4.: Seeded Evolution in VDrift.

Track	*Champion2009*	*Champion Adaptation*	*Improvement*
Dirt-3	3711.16	4387.49	18%
Ruudskogen	5269.72	6005.55	14%
Suzuka-2005	4172.36	8657.21	107%
Sepang	2379.44	7742.95	225%

7.3.4. Discussion

In this section we want to compare the performance of the three approaches presented so far. Table 7.5 reports the lap time, the average speed and the time spent outside the track for all the drivers developed with the considered methods. It is possible to see that the behavior of the Champion2009 is very aggressive: it drives very fast but often can go off track. This is particularly true in the last track where its aggressive policy drives very fast but it is not able to brake effectively before the two narrow turns after the two main straights (see Figure 7.3(c)). This can happens because of the difference in the simulation engine between TORCS and VDrift. On the other side the best drivers evolved from scratch present a very cautious driving policy that never goes off road. While in the first three tracks the drivers achieve competitive lap times, in the last track a too cautious driving policy results in a very high lap time: to avoid going off-road in the two strongest turns of Sepang the driver learns a sub-optimal policy which drives very slow during all the straight. The drivers evolved using adaptation show the best trade-off between the aggressive driving policy of the Champion2009 and the cautious one of the drivers evolved from scratch. So, the aggressive and fast policy of the Champion2009 is somehow adapted to the new dynamics generated from the VDrift engine. The best drivers evolved with adaptation achieve the best lap time in 3 tracks over 4 and only in one case a driver goes off-road for more than one second.

It is also interesting to consider the evolution time for finding a good driver. The evolution time is very important since, each run can last from 10 to even more than 100 hours [4] depending on the length of the track and the performance of the individuals. This happens because each fitness evaluation requires to simulate a race of one lap which takes

[4]The computer used for the experiments has a Core i5 and 4 GB of RAM

Table 7.5.: Comparison between the Champion2009 and the best drivers evolved From Scratch and with Adaptation.

Track	Statistics	Champion2009	From Scratch	Adaptation
Dirt-3	outside (s)	7.39	0.09	**0.00**
	lap time (s)	**106.55**	111.96	112.64
	avg speed (Km/h)	**74.7**	71.1	70.7
Ruudskogen	outside (s)	4.91	**0.00**	0.17
	lap time (s)	117.50	110.60	**100.91**
	avg speed (Km/h)	97.8	104.0	**114.0**
Suzuka-2005	outside (s)	28.38	1.04	**0.00**
	lap time (s)	250.48	256.92	**189.00**
	avg speed (Km/h)	84.1	82.0	**111.5**
Sepang	outside (s)	76.38	**0.22**	2.27
	lap time (s)	271.47	403.68	**268.86**
	avg speed (Km/h)	73.6	49.5	**74.3**

few seconds, but in some cases can be longer than 10-20 seconds [5]. As an example we report the performance on the track Dirt-3. Figure 7.4 shows the fitness trend of the evolutionary process From Scratch and with Adaptation in the track Dirt-3. For comparison it is also reported the fitness of the Champion2009. The process with Adaptation takes only 4 generations to find a competitive driver with a fitness of 4192.47. The process from scratch takes 24 generations to find a driver with fitness 4176.88, and 37 generations for a champion of 4191.4.

From our analysis it is possible to see that the drivers evolved with adaptation represent the best trade-off in terms of driving performance and learning speed (it can save several hours of computation). Thus, adapting for few generations a pre-existing model seems to be a promising way for transferring the knowledge across similar domains.

7.4. Summary

In this work we investigated how to transfer the knowledge between two complex domains. In particular we took into account two types of knowledge: (i) the behavior of an agent represented by a neural network; (ii) the fitness and the experimental design used to evolve the agent for a particular task. We considered two racing games with enough differences

[5]The simulation is performed using a customized version of VDrift where we disabled the graphical visualization. In this way, it was possible to get a simulation around 10 times faster.

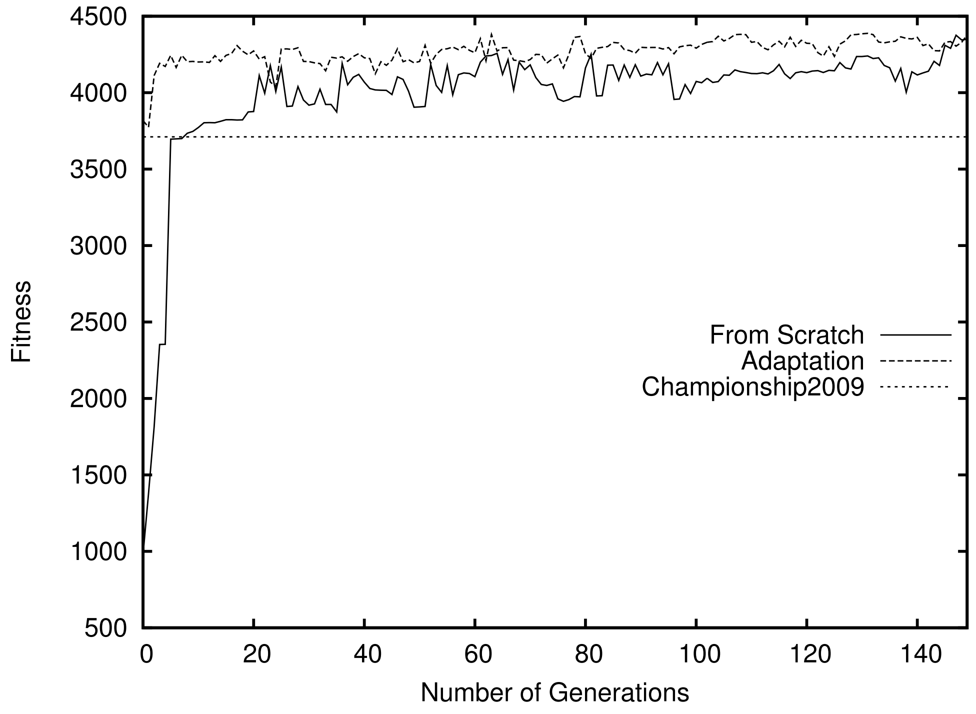


Figure 7.4.: Fitness trend of the evolutionary processes From Scratch and with Adaptation in track Dirt-3. For comparison it is also reported the fitness of the Champion2009.

regarding the simulation engine: TORCS and VDrift.

We investigated three different methodologies for transferring knowledge. The first one consists in copying directly the behavior evolved in TORCS to VDrift. The second one consists in replicating in VDrift the same fitness and the same experimental setup used to evolve a driver for TORCS. In this case we do not transfer a behavior, but the process for evolving the desired behavior. The third one is a combination of the previous methods: the behavior evolved in TORCS, is adapted in VDrift applying another run of neuroevolution. In this way the original behavior is adapted to the different dynamics of the target game.

For the experiments we used 4 tracks: two with medium difficulty and two with very high difficulty. Three of these tracks are available for both games while one is available only for VDrift. The results show that all the three methods work. The behaviors obtained using the three techniques are quite different: (i) the first method yields a very fast and aggressive driver that in the most difficult tracks can go off-road; (ii) the second one evolves very safe drivers that in some tracks can be much slower; (iii) the third one generates drivers that are both fast and safe. The third method shows a good trade-off in terms of performance and evolution time, since only few generations are necessary to obtain a good driver. In conclusion, adapting an existing solution is a promising method for transferring knowledge between two similar domains.

8. Evolving Racing Tracks

In this chapter, we apply Search-Based Procedural Content Generation for evolving racing tracks. In particular, we present two approaches for evaluating a track. The first approach, applies a fitness function based on entropy for evolving tracks with a good amount of variety and challenge. The second approach, is based on the user feedback and applies an interactive genetic algorithm to evolve tracks which can directly maximize the user's preferences.

8.1. Evolving Tracks for Racing Games

Racing games are a popular game genre with a rather rich catalog of titles which ranges from games that reproduce an actual event to games that take place in a fictional game universe. The latter ones are not bound to a specific event and therefore their game-play is not restricted to a specific set of tracks. Accordingly, they base their commercial success on their capability of providing a rich set of high quality tracks (e.g., Trackmania by Nadeo) and represent the ideal application domain for the methods of search-based procedural content generation. In fact, the possibility of generating a virtually infinite number of high-quality tracks may represent an attractive feature for the players, who would get infinite fun, but also for the developers and the publishers, who can inexpensively extend the game's shelf life.

In this chapter, we apply evolutionary computation for evolving racing tracks in TORCS. We design a track representation which extends the *radial encoding* of [94] by including additional degrees of freedom. We implement a mapping procedure between the encoding (the genotype) and the actual track (the phenotype) which deal with the several constraints that a high-end simulator like TORCS introduces.

We investigate two different approaches for evaluating the tracks. In the first approach, we use a theory-driven fitness function which focuses on the evolution of tracks with a large degree of diversity. More precisely, we view *track diversity* as the main source of challenge and interest for a racing games and eventually as one of the several features enabling players' fun. We focus on two metrics to capture diversity in terms of (i) the variety of turns and straights in the track and (ii) the range of driving speed achievable along the track.

In the second approach, we focus on maximizing the users' preferences. Thus, we propose an interactive genetic algorithm where the fitness of each track comes directly from the score given by users. We investigate two aspects: (i) evolution in single-user mode as a tool for supporting game designers; (ii) evolution in collaborative mode which can be used by a community of game users.

8.2. Track Representation

The representation of game content is a central issue in Search-Based Procedural Content Generation [103]. In this section, we briefly describe the TORCS representation of tracks, i.e., the phenotype, the indirect encoding we designed, i.e., the genotype, the genetic operators, and the mapping between genotypes and phenotypes.

8.2.1. Track Representation in TORCS (the phenotype)

In TORCS, a track is represented as an ordered list of segments. Each segment is either a straight or a turn. A straight is defined by just one parameter, its length. A turn is defined by (i) the direction (i.e., left or right); (ii) the arc it covers measured in radians; (iii) its start radius and (iv) its end radius. In addition, the track must be feasible, i.e., it must be *closed*, and therefore the last segment must overlap the first segment.

8.2.2. Track Encoding (the genotype)

The direct encoding of the track representation in TORCS into a genotype is infeasible since it produces a huge search space (thus leading to the curse of dimensionality) in which the feasible solutions (i.e., the closed tracks) are only a tiny proportion. Accordingly, we employed an *indirect encoding* inspired by the work of Togelius et al. [94] on a simple 2D car racing simulator. In [94], a track is represented as a set of *control points* that the track has to cover; the track is generated as a sequence of Bezier curves connecting three control points and, to guarantee smoothness, have the same first and second derivatives at the point they join.

In this work, we encoded a track as a sequence of control points $\mathbf{p} = \{p_1, \ldots p_n\}$, where p_i consists of three parameters r_i, θ_i, and s_i; the parameters r_i and θ_i identify the position of the control point p_i in a polar coordinate system (r_i is the distance from the origin or *radial coordinate*, θ_i is the *angular coordinate*); s_i controls the slope of the track tangent line in p_i. Figure 8.1 shows an example of our encoding: control points are depicted in red; the blue dot represents the origin of the polar coordinate system; the curves represent what generated by the genotype to phenotype mapping process, discussed in the next section.

8.2.3. Genotype to Phenotype Mapping

Algorithm 4 reports the pseudo code of the GenerateTrack procedure that maps our encoding into the track representation used in TORCS. The procedure takes as input the n control points $\mathbf{p}$ and returns a list $\mathbf{t}$ of track segments in TORCS format. Initially, the polar coordinates of the n control points (i.e., the r_i and θ_i values) are used to compute the ranges of feasible slope values for each control point (line 2). If there exist a control point for which there are no feasible slope values (thus it is not possible to join the incoming and the outgoing segments in that point), no track can be derived from $\mathbf{p}$ and therefore no track (a Null track) is returned. Otherwise, given the ranges of feasible slope values, the actual slope values are computed on the basis of the s_i values of the control points $\mathbf{p}$ (line 6). Next, for each pair of control points, p_i and p_{i+1}, the corresponding TORCS segment is generated using both the position and the slope values as constraints. Then, the procedure tries to close the track by generating one or more segments to connect t_{n-1} to t_1 (line 9). If the process succeeded (line 10), the resulting track $\mathbf{t}$ is returned otherwise a Null track is returned.

8.2.4. Closing the Track

In general, it is not always possible to connect the last and first control points with just one single segment and a sequence of straights and turns might be required. Accordingly, the procedure CloseTrack (line 10) considers the parameters defining the end and starting control points p_n and p_1 and try different heuristics to generate a feasible closed track.

Algorithm 4 Generate the track from the genotype

```
1:  procedure GenerateTrack($\mathbf{p}$)
                              ▷ $\mathbf{p}$: array of $n$ control points $\langle r_i, \theta_i, s_i \rangle$
                              ▷ $\mathbf{t}$: array of TORCS segments $\langle t_1, \dots t_n \rangle$
2:      $\{I_i\}$ = GenerateSlopeRanges($\mathbf{p}$)
                              ▷ No feasible slope range
3:      if $\exists i | I_i = \emptyset$ then
4:          return Null
5:      end if
                              ▷ Select slope values
6:      $\boldsymbol{\alpha}$ = SelectSlopes($\mathbf{p}$, $\{I_i\}$)
                              ▷ Generate the first $n-1$ segments
7:      for $i = 1$ to $n-1$ do
                $t_i$ = GenerateSegment($p_i$, $p_{i+1}$, $\alpha_i$)
8:      end for
                              ▷ Close the track
9:      $t_n$ = CloseTrack($t_{n-1}$, $t_1$)
                              ▷ Check whether the procedure failed
10:     if ClosedTrack($\mathbf{t}$) then
11:         return $\mathbf{t}$
12:     else
13:         return Null
14:     end if
15: end procedure
```

```
1:  procedure GenerateSegment($p_i$, $p_j$, $\alpha$)
                              ▷ Compute the slope of $\overline{p_i p_j}$
2:      $\alpha^*$ = Slope($p_i$,$p_j$)
3:      if $|\alpha - \alpha^*| < \epsilon$ then
                return GenerateStraightSegment($p_i$, $p_j$)
4:      else if $\alpha < \alpha^*$ then
                return GenerateLeftTurn($p_i$, $p_j$, $\alpha$)
5:      else if $\alpha > \alpha^*$ then
6:          return GenerateRightTurn($p_i$, $p_j$, $\alpha$)
7:      end if
8:  end procedure
```

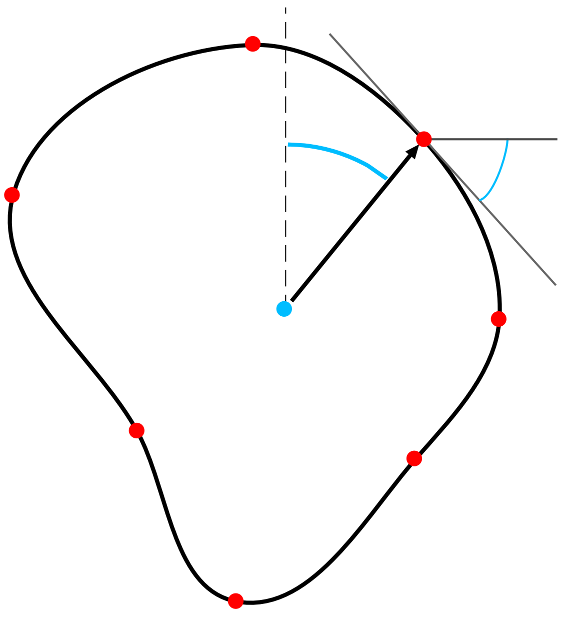


Figure 8.1.: Track encoding used by the evolutionary process: the red dots identify the control points; the blue dot is the origin of the coordinate system; an example of a feasible slope is also shown. The curves are the result of the genotype to phenotype mapping process.

8.3. Tracks Evolution based on Diversity

In this section, we apply evolutionary computation to generate racing tracks which show a good amount of diversity. In the first part, we define diversity and how this concept is translated into a fitness function. In the second part, we present the experimental analysis in which single and multi objective evolution is applied to maximize two different measures of diversity.

8.3.1. Track Evaluation

Our approach focuses on the evolution of racing tracks that can provide both an adequate amount of challenge (as in [94]), and can also provide a large degree of diversity, in a rather broad sense. Thus, we focused on the maximization of *the track diversity*. In particular, we can identify four ways of defining diversity in a racing game. The number, the distribution, and the types of turns in the track, i.e., the *curvature profile*, is one of the most evident source of diversity. The range and the distribution of the achievable driving speeds over the track, i.e., the *speed profile*, is another major source of diversity; in this respect, it is worth noting that, two tracks with the same curvature profile (i.e. containing the same number and types of turns and straights) can have very different shapes and therefore very different achievable speeds. Also the variety and the details of the roadbed (e.g., the presence of bumps, the percentage of gravel and asphalt) might be considered as a source of diversity. Finally, the surrounding scenery (e.g., landscape, trees, etc.) is an additional source of diversity.

In this work, we focused on diversity in terms of shape and in terms of achievable driving speed and present an approach to evolve racing tracks that maximize the diversity in terms of curvature and speed profiles. For this purpose, we evaluate racing tracks based on the entropy of their curvature and speed distributions and apply single-objective and multi-objective real-coded genetic algorithms to evolve tracks which maximize either one of the two criteria or both at the same time.

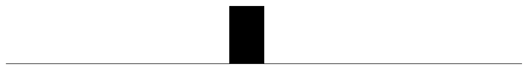
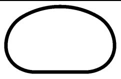
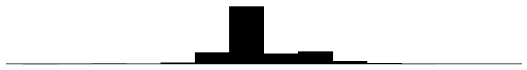
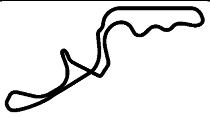
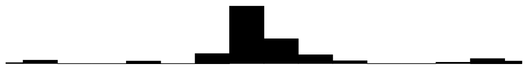
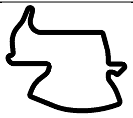
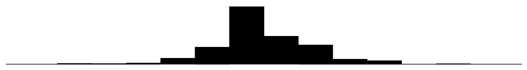
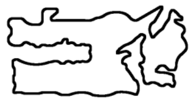
Evaluation Based on Curvature Profiles

As the first measure of track diversity, we focused on the shape and more precisely on the number and types of track segments. Accordingly, we defined the diversity of a track as the entropy of its *curvature profile* C, that is, the distribution of curvature values of all its track segments. For this purpose, we initially estimated the range of feasible curvature values by analyzing all the segments of all the human-designed tracks available on-line (i.e., those available in the TORCS distribution and the ones created by the users). Then, we partitioned the curvature range into sixteen bins $(b_1, \dots, b_{16})$.

Given a new track $\mathbf{t}$, we define its curvature diversity as follows. Firstly, for each track segment t_i we compute its length and its curvature as $1/\hat{r}_i$, where $\hat{r}_i$ is defined as (i) the radius for a right turn, (ii) the opposite of the radius for a left turn and (iii) it is considered infinite for a straight. Next, we generate the track curvature profile $C = \{c_1, \dots, c_{16}\}$, using the sixteen bins computed from the human-designed tracks; c_i is the percentage of the track with a curvature that belongs to bin i. Finally, we compute the entropy of its curvature profile $H(C)$ as,

$$H(C) = -\sum_{i=1}^{16} c_i \log_2 c_i \tag{8.1}$$

Table 8.1.: Curvature profiles in four of the tracks available with TORCS.

Track Name	**Profile**	Shape
D-Speedway		
Wheel-2		
Aalborg		
Spring		

so that $H(C) \geq 0$ and $H(C) \leq \log_2 16$. The entropy $H(C)$ measures the diversity in the distribution of curvature values in the track: it is maximum when all the curvature values occupy the same percentage of the track that is $c_i = 1/16$ for all the i; it is minimum when all the curvature values belong to the same bin, that is, there is an i such that c_i is one. Note that, we determined the number of bins (16) empirically as the best trade-off. In fact, an analysis we performed showed that a small number of bins (e.g., 2 or 4) would tend to produce too simple profiles which could not capture many of the difference between similar tracks. On the other hand, a larger number of bins (e.g., 32 or 64) would tend to produce too fine-grained profiles with several empty.

As an example, Table 8.1 reports the curvature profiles of four tracks available in the TORCS distribution, listed in increasing entropy value. The first track, *D-Speedway*, has an oval shape with a long straight and two long turns with large curvature radius r; accordingly, all the track segments belong to the central bin corresponding to near zero curvatures. In fact, since the curvature of a turn is computed as $1/r$ and the curvature of a straight is zero, both straights and large-radius bends have near zero values. As the number and model of different track segments increases, the track diversity increases too so that the curvature profile tends to include other bins corresponding to values quite far from zero (see for instance, the track *Aalborg*). The last track, *Spring*, provides the greatest diversity with a good variety of bends; as a consequence, the majority of the sixteen bins are covered by some track segments.

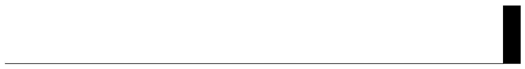
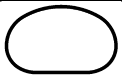
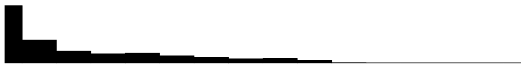
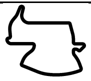
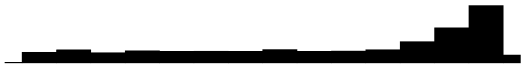
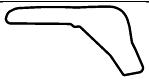
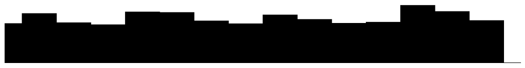
Evaluation Based on Speed Profiles

The second measure of track diversity we considered is based on the distribution of the achievable speeds. The underlying idea is that tracks are different not only because they look different (they have different types and distribution of turns and straights) but also because they allow for a wide variety of speeds. Therefore, we defined the track diversity in terms of achievable of speed as the entropy of its *speed profile* which we computed as follows.

As the very first step, we let several TORCS bots drive in all the available (human-designed) tracks for several laps and collected the car speed during each game tick; this produced the distribution of the feasible speed values over all the human-designed tracks. Then, we partitioned the range of feasible speeds into sixteen bins $(b_1, \ldots, b_{16})$, as we did for curvature profiles.

Given a new track $\mathbf{t}$, we measure its speed diversity as the entropy

Table 8.2.: Speed Profiles in some tracks of the TORCS package

Track Name	**Profile**	Shape
D-Speedway		
Aalborg		
Forza		
Wheel-2		

of its speed profile, which we compute as follows. Firstly, we have a bot completing three laps on the track $\mathbf{t}$ and, during the second lap, we measure the car speed for each game tick. Next, we generate the speed profile $S = \{s_1, \dots, s_{16}\}$, using the previously generated bins, where s_i is the percentage of game ticks, in the second lap, that the car raced with a speed corresponding to the bin i. Finally, we compute the entropy of the track speed profile as,

$$H(S) = -\sum_{i=1}^{16} s_i \log_2 s_i \tag{8.2}$$

As before, $H(S) \geq 0$ and $H(S) \leq \log_2 16$. The entropy $H(S)$ measures the diversity in the distribution of speed values over the track: it is maximum when the car spent the same amount of time in all the speed ranges $s_i = 1/16$ for all the i; it is minimum when the car spent most of the time in the same speed range, that is, there is an i such that s_i is one.

Table 8.2 reports the speed profiles of four tracks available in TORCS, listed in increasing entropy value. The oval track *D-Speedway*, which did not provide much diversity in terms of curvature profile, does not provide much diversity also in terms of speed. Its long bends and long straights allows the driver to keep basically the same high speed range all the time. In fact, all the recorded speed values fall into the last bin. The second track, *Aalborg* has several narrow bends that result in a speed profile skewed toward the lower speeds. In contrast, *Forza* has several fast stretches and few bends of rather different curvature accordingly its speed profile is skewed towards higher speeds; on the other hand, the rather different curvatures of the few bends in *Forza* allows the car to cover almost all the possible speed ranges. The fourth track, *Wheel-2*, is one the track providing the highest degree of diversity among all the one available in TORCS as demonstrated by the well-balanced and almost uniformly distributed speed profile.

Discussion

Wheel-2 and *Aalborg* are good examples of how the two diversity measures we introduced actually estimates two very different ideas of diversity. When the topology is concerned, although *Wheel-2* looks more articulated than *Aalborg*, it turns out that *Wheel-2* provides *less* diversity than *Aalborg* since most of its turns and bends have similar curvatures. In contrast, *Aalborg* has several long stretches but also many turns of very different radius which provide more diversity than *Wheel-2*. On the other hand, when speed is concerned, *Wheel-2* provides *more* diversity than *Aalborg* because its balanced combination of many soft turns and few tight turns, which allows the drivers to cover basically all the possible speed ranges. In contrast, the many different tight turns of *Aalborg* restrict the driver towards the lower scale of the speed range. Even when the car is driving along the stretches, the car speed cannot increase too much because of the incoming very tight turns. In fact, the two measures we introduced are only slightly overlapping since there are very few tracks with qualitatively similar curvature and speed profiles (e.g., *D-Speedway* or other ovals). Accordingly, in this work we investigated both the application of (i) single-objective evolution to maximize the diversity for each one of the two criteria and (ii) multi-objective evolution to search for a good trade-off between the two.

Finally, we wish to point out that, although the entropy of curvature profiles provides a more intuitive idea of track diversity (that it is related to a visual property) and it simpler to compute (since no racing is involved), it actually poses much tighter constraints than the entropy of speed profiles. In fact, the entropy of a curvature profile is maximal when the track contains *the same percentage of turns with all the feasible curvature values*. In contrast, the entropy of a track speed profile is maximal when the driver has to cover all the possible speed ranges for the same amount of racing time. Accordingly, while high-entropy curvature profiles correspond to rather complex (and difficult to evolve) track topologies, high-entropy speed profiles are possible even in simpler (and easier to evolve) topologies.

8.3.2. Experimental Results

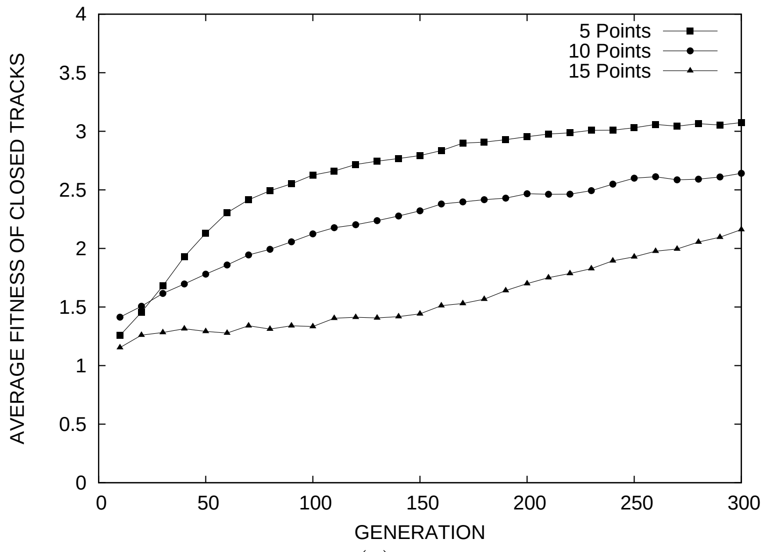
We applied single-objective and multi-objective real-coded genetic algorithms to evolve tracks with a large degree of diversity both in terms of curvature profile (i.e., the number and types of turns and straights) and in terms of speed profile. For this purpose, we performed three sets of experiments respectively focused on the maximization of (i) the entropy of curvature profiles (i.e., of the shape diversity), of (ii) the entropy of speed profiles (i.e., of the speed diversity) and (iii) the search of a good trade-off between these two measures. All the experiments were performed using the implementations of the single and multi-objective genetic algorithms available in Sastry's genetic algorithm toolbox [72] while for the evaluation of tracks was performed using TORCS 1.3.1.

Maximizing the Entropy of Curvature Profiles

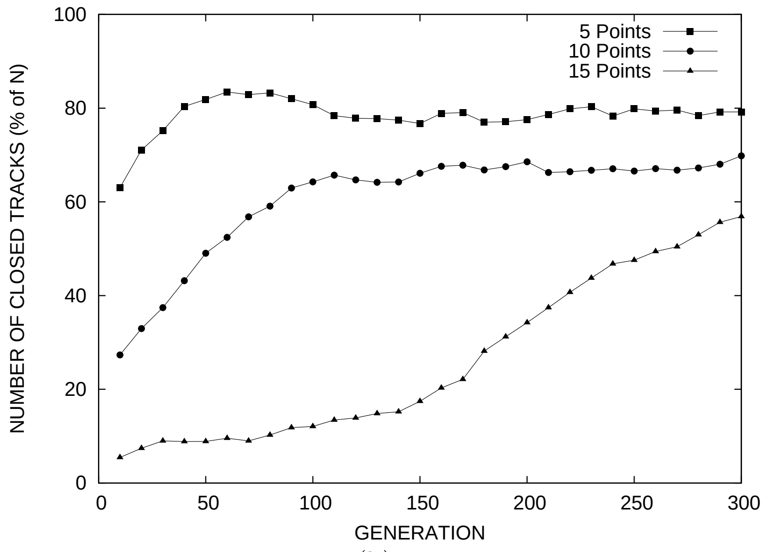
In the first set of experiments, we applied a single-objective real-coded genetic algorithm in which individuals are tracks encoded using 5, 10, or 15 control points (Section 8.2) and the fitness function is computed as the entropy of the track curvature profile (Section 8.3.1); individuals corresponding to infeasible (open) tracks receive a zero fitness; in addition, since the track generator of TORCS cannot deal with track intersections, we penalized the fitness of individuals by 0.5 for each intersection. The parameters of the genetic algorithm were set as follows [72]: the mutation probability was 0.1 while the crossover probability was 0.9; the population size was set to 50, 100, and 150 individuals for tracks represented using 5, 10, and 15 control points, respectively; selection is performed using tournament selection with a tournament size of 2; the process ended as soon as 300 generations were completed.

Figure 8.2 reports the average population fitness (Figure 8.2a) and the percentage of individuals in the population which represent feasible (closed) tracks (Figure 8.2b). As can be noted, the higher the number of control points used to represent the tracks is, the more complex the search space becomes. In particular, as the number of control points increases, the evolution of high entropy individuals becomes more difficult: when tracks are represented using 10 or 15 control points, the average fitness of the population grows more slowly and reaches a smaller entropy value when the process ends (Figure 8.2a). Similarly, as the number of control points increases, it becomes more and more difficult to generate valid tracks and the percentage of valid (closed) tracks in the population grows much slower than fewer points (Figure 8.2b). It is important to stress that the genetic algorithm can always increase the number of closed tracks consistently as the generations proceed. For instance, when individuals encode 15 control points, only the 5% of the initial randomly generated population consists of valid (closed) tracks; however, after 300 generation, slightly more than the 56% of the individuals in the final population represent valid tracks.

The results for the tracks encoded using 15 control points suggest that the problem of generating valid tracks is not trivial and in fact, a random population contains quite a small percentage of valid tracks. On the other hand, the evolutionary approach we propose appears to be rather successful in that it can actually produce a significant increase both in terms of diversity, as the average fitness reaches a near-optimal value (Figure 8.2a), and in terms of number of valid tracks evolved, as their number reaches a tenfold increase (Figure 8.2a). Figure 8.3 reports

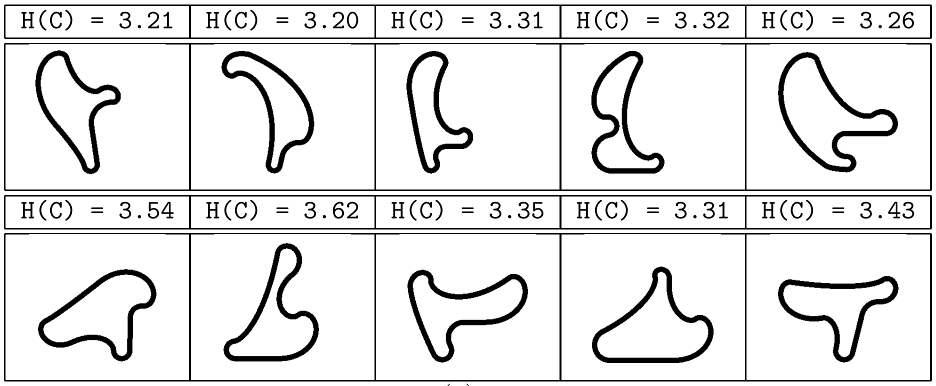


(a)

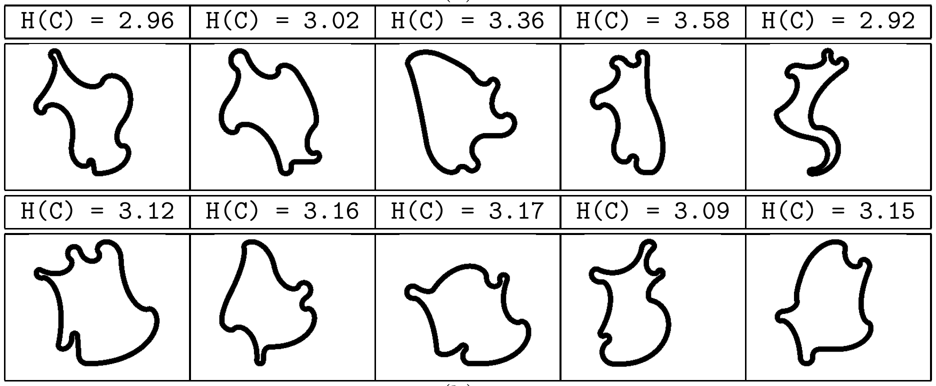


(b)

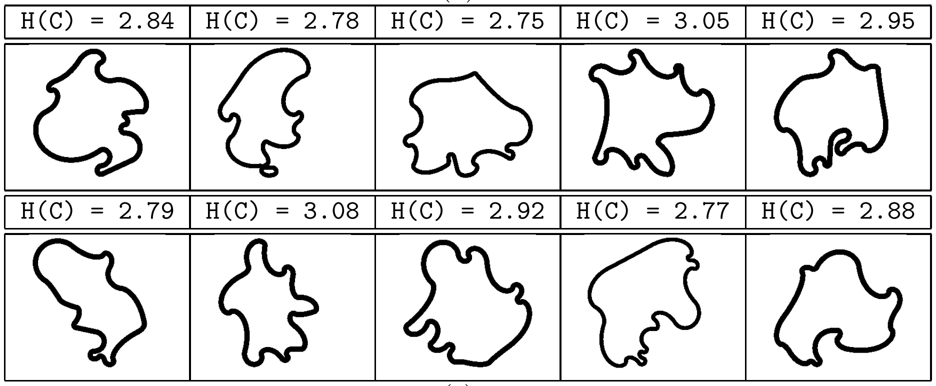
Figure 8.2.: Single-objective genetic algorithm using the entropy of curvature profiles as fitness function and a track encoding involving 5, 10 and 15 control points: (a) average fitness of the closed tracks in the population; (b) percentage of individuals that represent a feasible (closed) track. Curves are averages over ten runs.



(a)



(b)



(c)

Figure 8.3.: The best tracks evolved for each run using the entropy of curvature profiles when the tracks are encoded using (a) 5 control points, (b) 10 control points, and (c) 15 control points; for each track is also reported `H(C)`, the entropy of the curvature profiles.

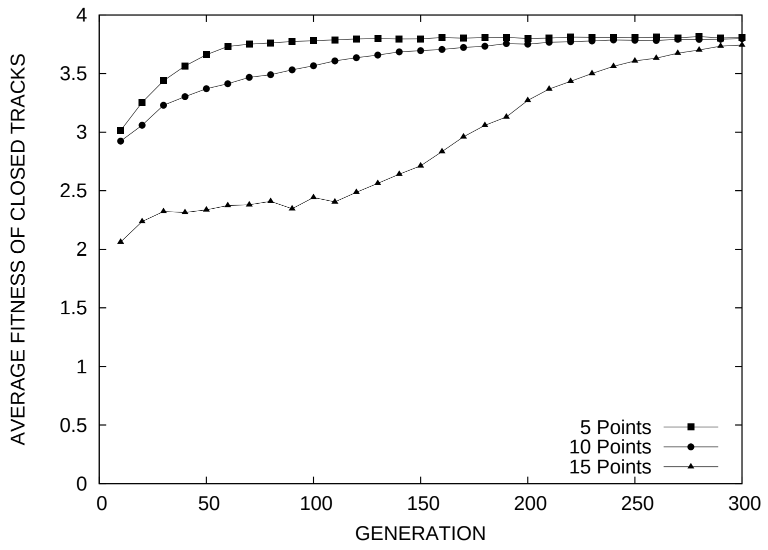
the best track evolved for each of the ten runs of the genetic algorithm when 5 control points (Figure 8.3a), 10 control points (Figure 8.3b), and 15 control points (Figure 8.3c) are used. As should be expected, as the number of control points increases, also the complexity of the evolved tracks increases, making the maximization of the entropy of curvature profiles more difficult. In fact, the best tracks evolved using 5 control points (Figure 8.3a) have a slightly higher fitness than the best tracks evolved using more control points. From the one hand, this result suggests that 5 control points are enough to evolve tracks which (i) include a quite large range of different turns (in terms of radius) and (ii) are also well-balanced, in that they provide a good mix of different types of turns and straights. On the other hand, with more than 5 control points, it is still possible to evolve tracks featuring a wide range of different turns and straights, but it might be difficult to obtain well-balanced tracks.

Finally, it is worth noticing that not all the tracks include straights. This is not surprising and depends on the way we compute the curvature profile for a track. In fact, according to our definition of curvature profile, both straights and turns with very large radius of curvature are basically identical in terms of profile since they belong to the same discretization bin. As a consequence, evolution will tend to evolve tracks featuring turns with with small curvature turns (or with very large radius of curvature) instead of simple straights.

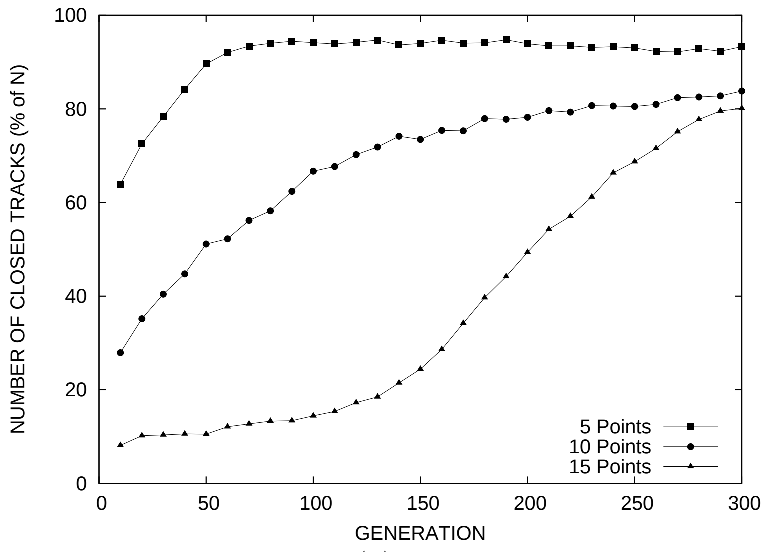
Maximizing the Entropy of Speed Profiles

We repeated the same set of experiments, using the same parameter settings, while computing the fitness of a track as the entropy of its speed profile. Also in this case, we penalized the fitness of individuals by 0.5 for each intersection. Figure 8.4 reports the average population fitness (Figure 8.4a) and the percentage of feasible (closed) tracks in the population (Figure 8.4b) as a function of the number of generations. As in the previous set of experiments involving curvature profiles, a higher number of control points results in a slower convergence to high entropy individuals (Figure 8.4a). In contrast to what happened with the curvature-based fitness, in this case, all the average population fitnesses converge almost to the same near-optimal value. This result can be easily explained by considering that, as we previously noted in Section 8.3.1, it is generally easier to evolve tracks with high entropy speed profiles than it is for curvature profiles because of the tighter constraints they pose.

The plot for the percentage of closed tracks in the population (Figure 8.4b) also confirms our previous findings: as the number of control



(a)



(b)

Figure 8.4.: Single-objective genetic algorithm using the entropy of speed profiles as fitness function and a track encoding involving 5, 10 and 15 control points: (a) average fitness of the closed tracks in the population; (b) percentage of individuals that represent a feasible (closed) track. Curves are averages over ten runs.

points used to represent the track increases, the evolution of valid closed tracks becomes more difficult. In fact, a higher number of control points corresponds to an initial random population mainly made of infeasible (open) tracks: with 10 control points only around the 30% of the random tracks are closed, with 15 control points only around the 5% of the random tracks are closed. Also in this case however, the evolutionary process can successfully get rid of infeasible individuals so as to converge to a population containing mainly valid tracks with high entropy values. In particular, it is worth noticing that the percentage of valid tracks in the final population evolved using speed profiles (Figure 8.4b) is much higher than the one obtained using curvature profiles (Figure 8.2b).

Figure 8.5 reports the best tracks evolved during each one of the ten runs involving 5 control points (Figure 8.5a), 10 control points (Figure 8.5b), and 15 control points (Figure 8.5c). Figure 8.5 confirms our previous findings in that 5 control points seem enough to evolve tracks with a large degree of diversity which, in this case, means tracks involving a wide range of turns and straights and uniformly distributed variety of speed values over the lap. In this case however, in contrast to what happens for curvature profiles, the tracks evolved using 10 and 15 control points achieve almost the same high entropy values as the tracks evolved using only 5 control points.

Searching for a Trade-off Using Multi-Objective GAs

The results for the single-objective genetic algorithm show that the fitness based on curvature profiles produces tracks with a rather broad range of turns, from hairpin to very large and fast bends, while the fitness based on speed profiles produces tracks with a good balance of fast and slow stretches, leading to a rich and challenging driving experience. Early on, in Section 8.3.1, we argued that these two fitness definitions capture rather different ideas of diversity. We tested our hypothesis empirically by analyzing the correlation between the entropy of curvature profiles and the entropy of speed profiles using a set of 1000 tracks evolved using the same single-objective genetic algorithms used in the previous experiments. Our analysis returned a correlation coefficient of 0.56 suggesting that the two fitness definitions *are not highly correlated* and that a track with a high entropy curvature profile does not necessarily have a high entropy speed profile. Accordingly, we applied a multi-objective genetic algorithm to evolve tracks, encoded with 5, 10, and 15 control points, which could maximize the two objective at the same time. For this purpose, we used the same parameters settings used

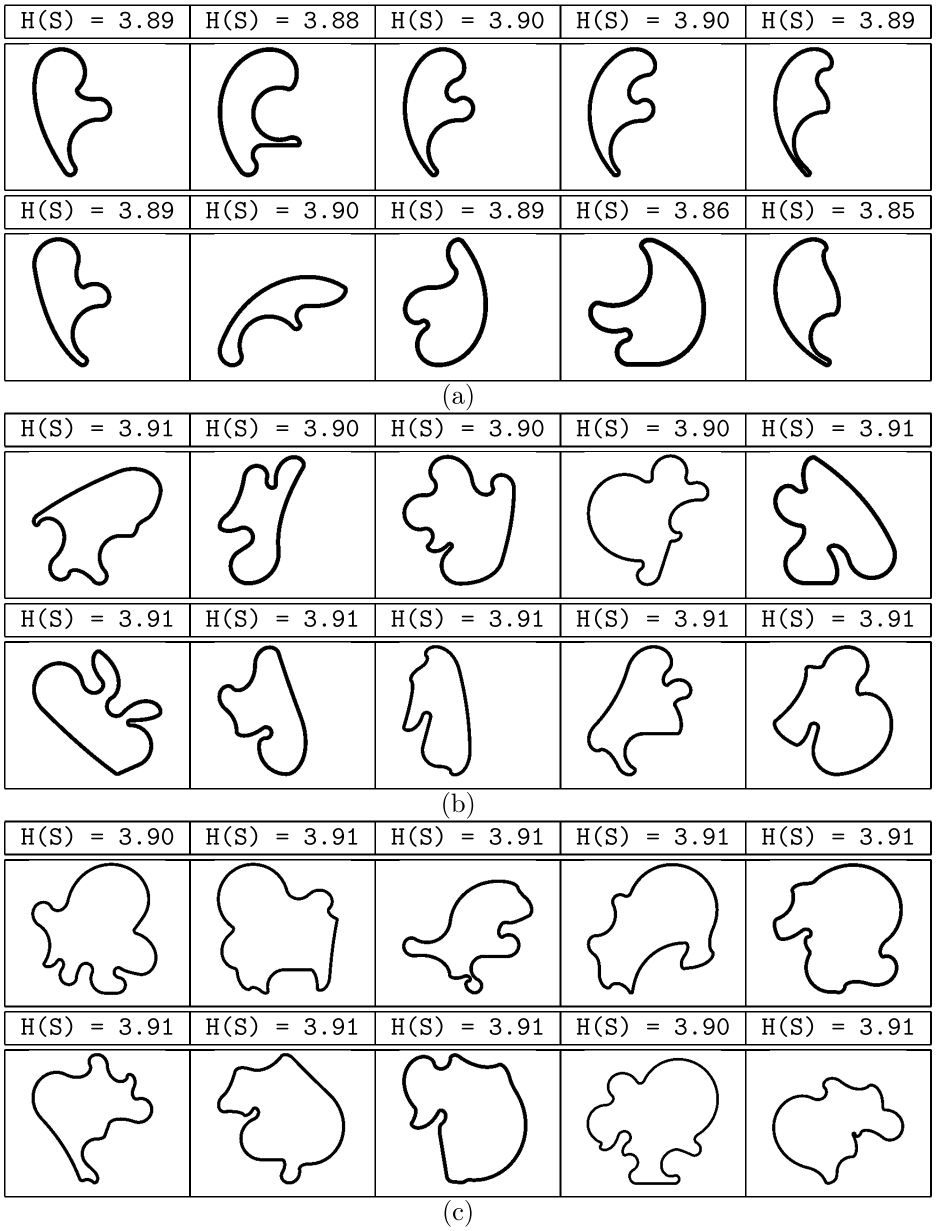
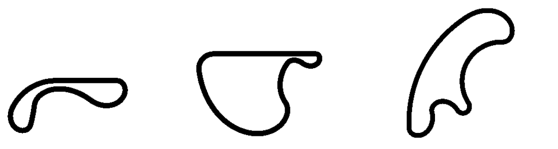
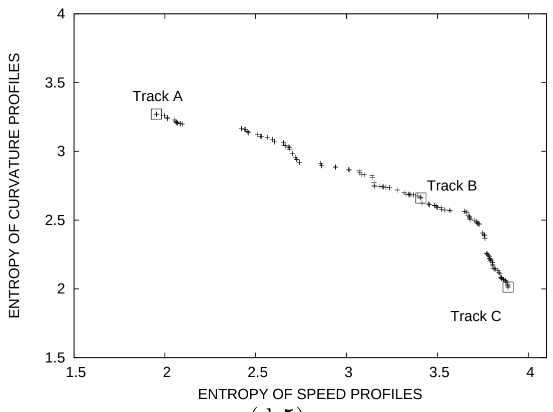


Figure 8.5.: The best tracks evolved for each run using the entropy of speed profiles when the tracks are encoded using (a) 5 control points, (b) 10 control points, and (c) 15 control points; for each track is also reported H(S), the entropy of the speed profiles.



(a-5) Track A (b-5) Track B (c-5) Track C



(d-5)

Figure 8.6.: Multi-objective genetic algorithms applied to evolve tracks encoded using 5 control points;. Track A, Track B, and Track C are examples of tracks evolved in different sections of the pareto; (d-5) is an example of the final pareto front evolved after 300 generations.

in the previous experiments; offspring selection was performed using the Non-Dominated Sorting Genetic Algorithm (NSGA-II) implementation available in Sastry's genetic algorithm toolbox [17, 72].

Figures 8.6, 8.7 and 8.8 report, for each encoding, three representatives of the Pareto front evolved after 300 generations and the overall Pareto front. All the results agree in that the three Pareto fronts clearly show that there is a conflict between the two objectives, i.e., the maximization of diversity in terms of curvature and in terms of speed profiles. Nevertheless, the evolved Pareto fronts contains very different tracks models providing both a very good trade-off between the two competing objectives with a nice balance between the variety of turns and a rich driving experience.

Interestingly, the best solutions evolved by the multi-objective genetic algorithm have very high (sometime near-optimal) entropy values both in terms of curvature and speed profiles suggesting that multi-objective

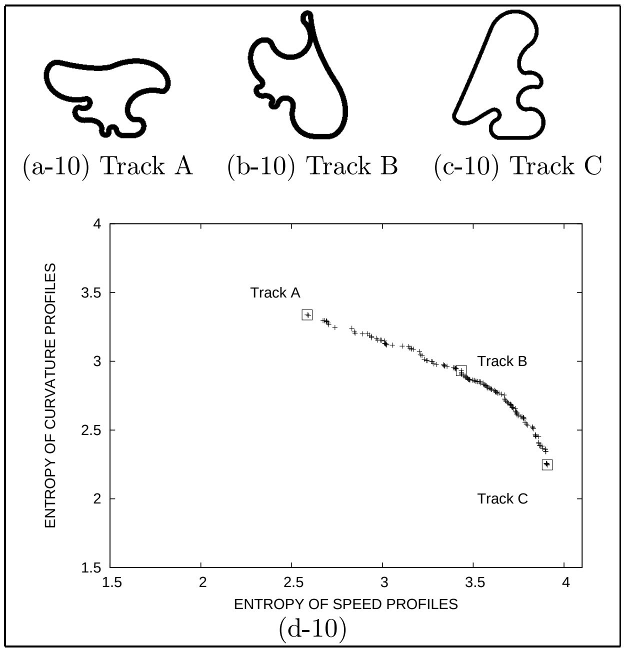


Figure 8.7.: Multi-objective genetic algorithms applied to evolve tracks encoded using 10 control points;. Track A, Track B, and Track C are examples of tracks evolved in different sections of the pareto; (d-10) is an example of the final pareto front evolved after 300 generations.

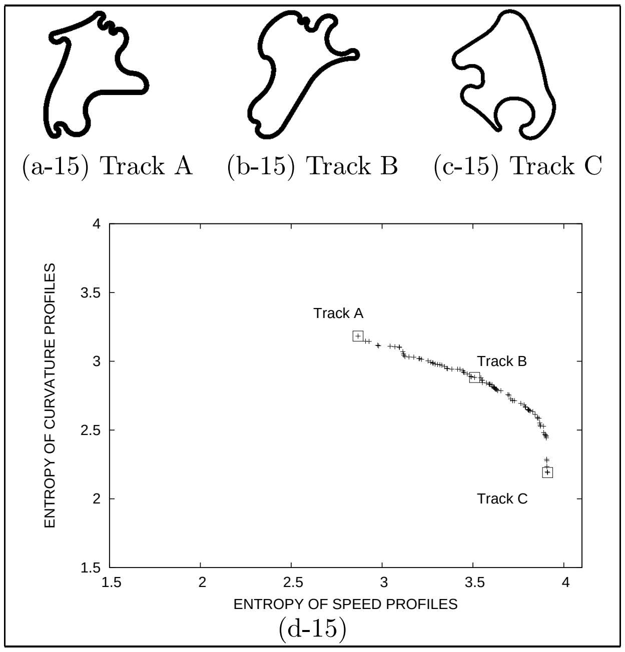


Figure 8.8.: Multi-objective genetic algorithms applied to evolve tracks encoded using 15 control points;. Track A, Track B, and Track C are examples of tracks evolved in different sections of the pareto; (d-15) is an example of the final pareto front evolved after 300 generations.

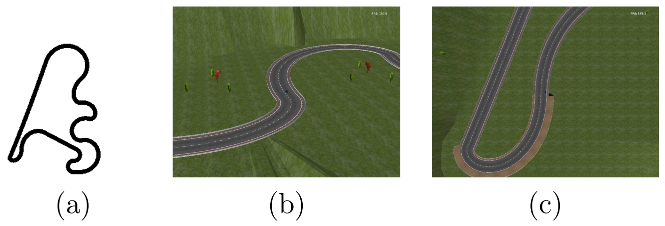


Figure 8.9.: The track `Wild-Speed` used in the second leg of the 2010 Simulated Car Racing Championship held at WCCI-2010, Barcelona, Spain: (a) the shape of the track, (b) and (c) screenshots of the track in the game.

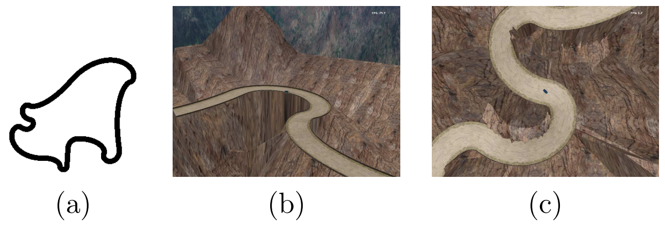


Figure 8.10.: The track `Rocky` used in the third leg of the 2010 Simulated Car Racing Championship held at CIG-2010, Copenhagen, Denmark: (a) the shape of the track, (b) and (c) screenshots of the track in the game.

evolution is probably the best approach to tackle this application.

As a proof of concept, we selected two of the tracks evolved using the multi-objective genetic algorithm for the 2010 Simulated Car Racing Championship. Figure 8.9 and Figure 8.10 show the track `Wild-Speed` and the track `Rocky` which were used during the second leg and the third leg of the championship. The evolved tracks were not modified and their shapes are just the result of the evolutionary process. We added however some scenic elements to make them more attractive for the competition attendees (for instance, we added a terrain, some trees, and changed the roadbed).

8.4. Tracks Evolution using Interactive GA

In this section, we focus on evolving racing tracks which can maximize the preference of the users. Thus, we propose an interactive genetic algorithm where the fitness of each track comes directly from the score given by users. To validate this approach we designed and implemented a web-based system. The proposed system can be used both in single user-mode or in collaborative mode with a common population shared by several users. In the first part of the section, we present the architecture of our system which include two main parts: (i) a web frontend and (ii) an evolutionary backend. In the second part, we validate our system with two sets of experiments involving some human subjects.

8.4.1. Our Framework

The structure of our interactive evolutionary system is depicted in Figure 8.11. The systems consists of two servers. The web frontend (i) manages the interaction with the users, (ii) renders active populations, (ii) collects evaluations of the users, (iii) maintains the database of evolving/evolved populations, and (iv) updates the request queues for the evolutionary backend. The evolutionary backend runs (i) all the evolutionary operators (selection, recombination, and mutation) on the currently active populations, (ii) the generation of actual game content (i.e., the mapping procedures between genotypes and phenotypes), and (iii) any other TORCS-related procedure such as the rendering of the track thumbnail pictures, which requires the invocation of a TORCS executable. The partitioning between a web/database dedicated frontend and an evolutionary dedicated backend was mainly introduced to improve the load-balancing between the generally light-weighted user interface related tasks and the CPU demanding evolutionary and TORCS related tasks. In addition, since TORCS executables require libraries that are not generally available on the most typical webservers, we decoupled the the web-related application from the TORCS-related applications also to make our system portable to more hosting services.

The Frontend User Interface

People can access the framework using a browser by logging in either as registered or anonymous users, and can request either to start/join a single-user session or to join an active multi-user session.

Registered users are fully tracked and thus can (i) save the state of an active evolutionary process, (ii) continue a previously saved process, and (iii) explore all their history (e.g., all the populations they evolved, all their preferences). In contrast, anonymous users are only tracked using browsers' cookies so that the status of their evolutionary process

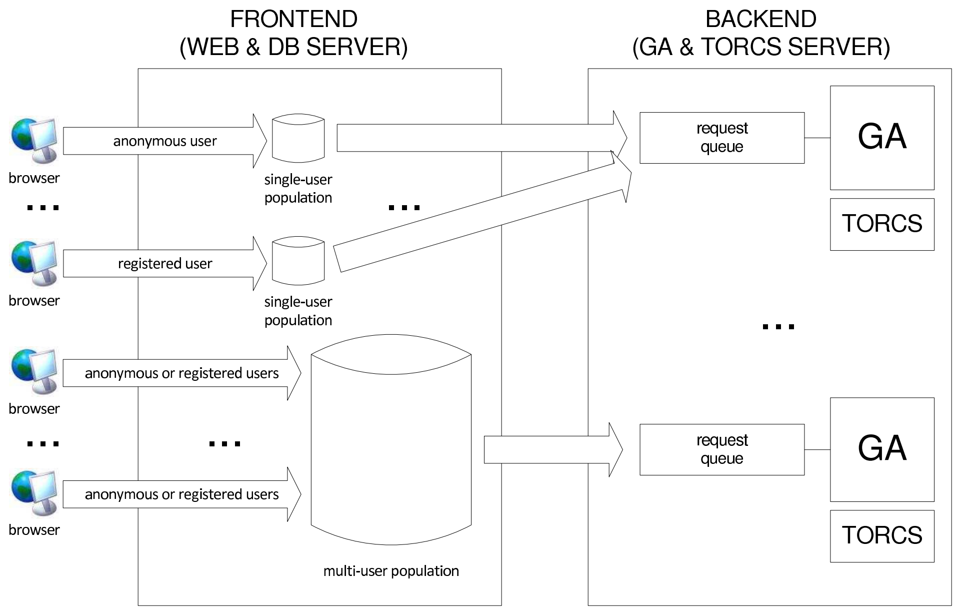


Figure 8.11.: Structure of the interactive evolutionary system.

is lost as soon as the browser cookies are cleared or they connect from a different machine.

In single-user mode, users have the complete control over the evolutionary process in that (i) they can customize the parameter set up by specifying the population size, the selection policy (tournament or truncation), etc. (ii) they can evaluate *all* the individuals in the population, and (iii) they can request the next selection, recombination, mutation cycle. As an example, Figure 8.12 shows the interface for a single-user evolution. The upper section of the interface includes all the commands available to the user: `Evolve` requests the next selection, recombination and mutation step to the evolutionary backend and then updates the current page with the new population; `Reset` restarts the evolution from scratch (all the previous populations are stored and a new random population is generated); `Generations` provides access to the data of all the previous generations.

The lower section of the interface depicts the current population and, for each track, it shows (i) the track thumbnail and (ii) its scoring interface. In particular, the framework provides two scoring interfaces (Figure 8.13): a ranking interface (see Figure 8.12 and Figure 8.13a) which asks users to rank an individual with an integer between 1 and 5; a like/dislike interface (Figure 8.13b), which asks users simply whether they like the track.

In multi-user mode, users have very limited control over the process in that they are *only* allowed (i) to explore and evaluate a small number individuals, randomly selected from the much larger underlying population, and (ii) to access the hall of fame of the best individuals evolved so far. In this mode, users cannot specify the parameters of the genetic algorithm, nor they can request the activation of the evolutionary cycle which is actually triggered by a heuristic based on the number of evaluations received.

The Evolutionary Backend

The backend runs of all the evolutionary and TORCS related tasks. It hosts a number of servers (labeled GA in Figure 8.11) each one listening to a separate request queue. GA servers are mainly responsible of (i) running the selection, recombination, mutation cycle; (ii) rendering the newly created populations, which requires the execution of TORCS applications; and (iii) updating the status of the evolutionary process on the main databases. A selection, recombination, mutation cycle can be requested either (i) explicitly, in single-user mode, by the user through the `Evolve` command on the main interface, (ii) implicitly, in multi-user mode, by the GA server, when a sufficient number of evaluations for the individuals in the population has been received. Note that, for load balancing purposes, all the incoming requests from the web frontend are queued so that each GA server can explicitly manage the number of running processes. When a new generation is required, a GA server first performs the standard selection, recombination, and mutation; then, for each one of the newly created individuals, the server runs a TORCS executable that generates the actual TORCS track (the phenotype), computes several track statistics, and render the track shape; finally the status on the database is updated. Since TORCS executables tend to be CPU and disk intensive, the GA server schedules the number of active TORCS invocations based on the number of cores available to avoid overloading the server.

GA servers implement a rather simple real-coded genetic algorithm that accesses a population of tracks stored on a remote database, using tournament or truncation selection, single-point crossover, and mutation. When an infeasible track is generated, it is discarded and another one is generated. In single-user mode, when very small population are involved, the 10% of a new population is filled with randomly generated tracks so as to avoid premature convergence.

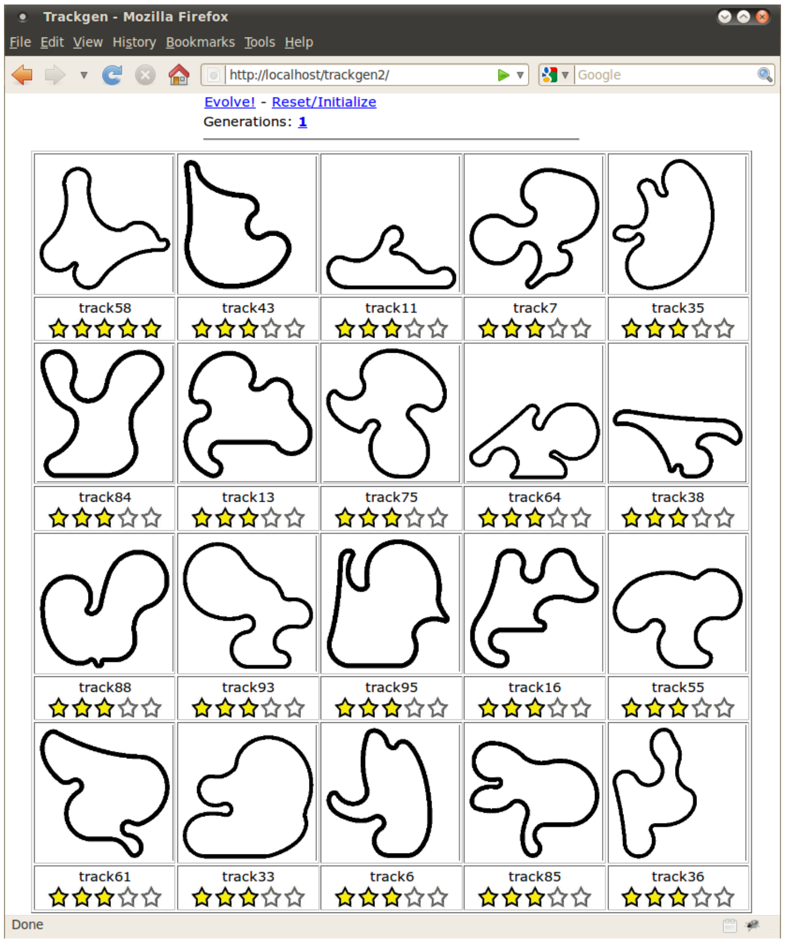


Figure 8.12.: A screenshot of the User Interface.

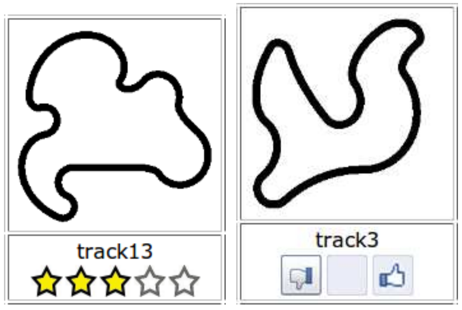


Figure 8.13.: Scoring interfaces: (a) the ranking interface asks users to rank an individual with an integer between 1 and 5; (b) the like/dislike asks users to specify just whether they like the track.

8.4.2. Experimental Validation

We performed two sets of experiments involving five users to provide an initial validation of the proposed framework. In the first set of experiments, we focused on the single-user mode and investigated how (i) the scoring interface (i.e., either a ranking interface or a like/dislike interface) and (ii) the selection policy (i.e., either tournament or truncation selection) affect the quality of the evolutionary process. In the second set of experiments, we studied the system working in multi-user mode.

Single-User Mode

In the first set of experiments, we tested four different configurations: (i) the ranking scoring interface combined with tournament selection; (ii) the like/dislike scoring interface combined with tournament selection; (iii) the ranking scoring interface combined with truncation selection policy; (iv) the like/dislike scoring interface combined with truncation selection policy. For each configuration, we asked five human subjects to complete ten generations of single-user evolution using a population consisting of 20 individuals. During the experiments subjects did not know the type of selection mechanisms in use (tournament or truncation). Since subjects could access the system as registered users, they were allowed to pause and restart the evolutionary process as they wished (which, in principle, should have limited their fatigue). However, our tests show that the evaluation of a population with 20 individuals for 10 generations would typically take around 20-30 minutes of actual time (i.e., with no pause in between evaluations). To measure the progress during the evaluation process, for each generation we computed the average score of the individuals in the population as follows. When the ranking interface is used, a integer value (i.e., a fitness) between 1 and 5 is assigned to each track according to the user preferences. When the like/dislike interface is used, a (fitness) value of 1 is assigned to tracks the user does not like whereas a (fitness) value of 5 is assigned to tracks the user likes.

We first analyzed how the scoring method influences the evolutionary process. For this purpose, we grouped all the collected data based on the scoring method used. Figure 8.14 compares the average population scores for the runs using the ranking interface (empty dots) and the like/dislike interface (solid dots). Both curves show an improvement in the users satisfaction as the number of generation proceeds. The use of the simpler like/dislike interface appears to provide a more evident increase of the user satisfaction, whereas the ranking interface results in a slight improvement over time. This result was later confirmed by the feedback we received by the subjects who stated they perceived the simpler (like/dislike) interface as more effective to express their preferences while they found it difficult to provide meaningful/coherent overall rankings.

We performed a similar analysis to study how selection may influence the evolutionary process. Accordingly, we grouped all the collected data based on the selection method; it is important to stress that, users did not know the selection method used during the experiments. Figure 8.15 compares the average population scores for the runs using tournament (empty dots) and truncation selection (solid dots). Again both curves show an improvement in the users satisfaction as the number of generation proceeds. In this case however, there is no noticeable difference between the trials using truncation or tournament selection, suggesting that the selection mechanism had no influence on the users.

Overall, the feedback we received from the subjects was generally positive. In most cases, users reported that they perceived improvements in the quality of the individuals between subsequent populations. They also found that, at the end, the process produced interesting tracks. As an example, Figure 8.17 and Figure 8.18 show two of the most interesting tracks evolved according to the users. The only criticism we recorded concerned the ranked scheme which appears to cause, sometimes, fatigue and frustration in the users. In addition, users told us they would tend to be annoyed when many very similar individuals appeared in the same population.

Multi-User Mode

At the end, we performed a preliminary experiment to test the multi-user mode. The experiment involved the same five human subjects, participating in the previous experiments, who in this case shared the same population; subjects could not communicate one another in anyway while the experiment was running. Given the limited number of subjects involved, we used a population of only 20 individuals (the same size used in the previous experiments) and thus allowed all the subject to score all the individuals. Tracks were evaluated using the simpler like/dislike interface; track fitness was computed as the average score received from the users; truncation selection was applied.

Figure 8.16 reports the average population score over the number of generations. As in the single-user experiments, the curve shows a clear

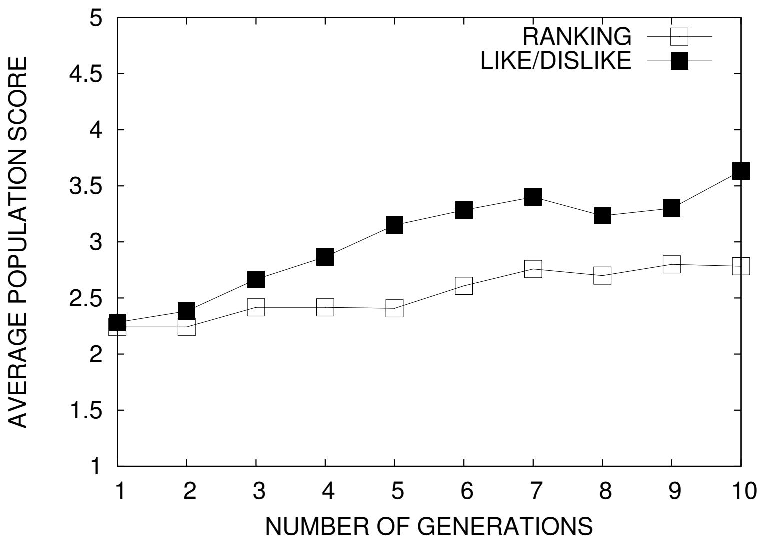


Figure 8.14.: Average population score for the trials using the like/dislike evaluation (solid dots) and the ranked evaluation (empty dots). Curves are averages over 10 trials.

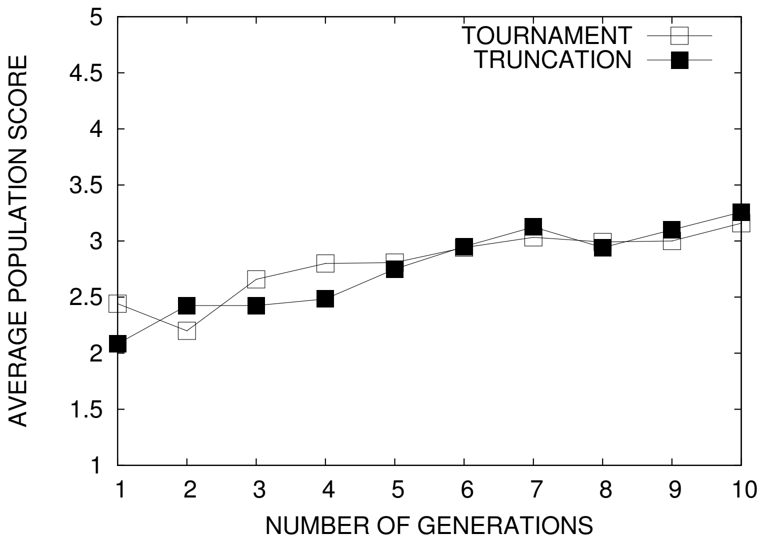


Figure 8.15.: Average population score for the trials using truncation selection (solid dots) and tournament selection (empty dots). Curves are averages over 10 trials.

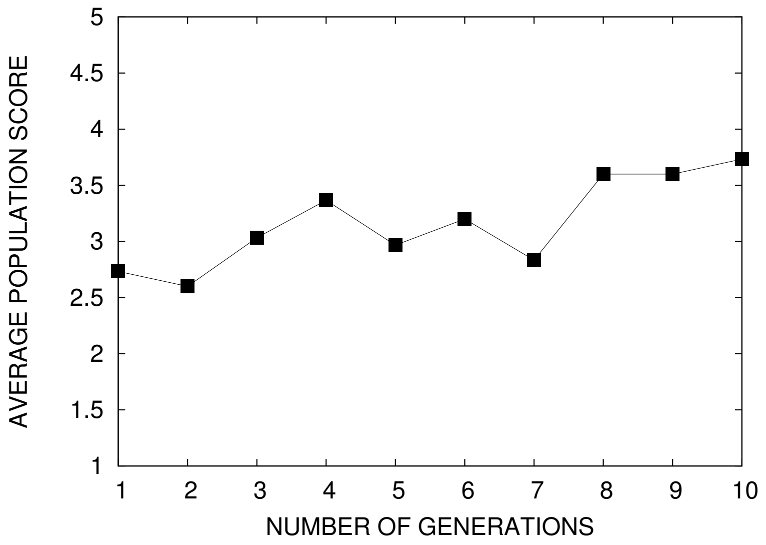


Figure 8.16.: Average population score for one trial using like/dislike scoring and truncation selection.

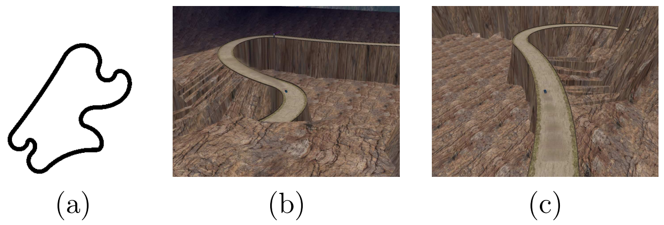


Figure 8.17.: First example of track evolved in single-user mode: (a) shape of the track as used during the evolution; (b), and (c) actual in-game renderings.

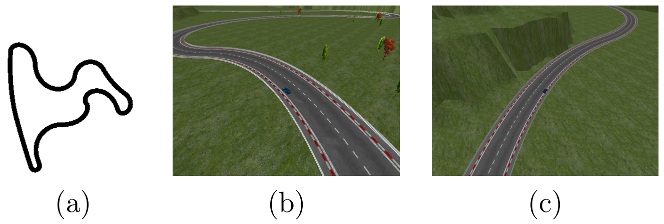


Figure 8.18.: Second example of track evolved in single-user mode: (a) shape of the track as used during the evolution; (b), and (c) actual in-game renderings.

improvement in the users' satisfaction, in fact, the number of likes increases as the evolution proceeds, notwithstanding possible conflicting opinions that the users have expressed.

8.5. Summary

In this chapter, we investigated the automatic generation of tracks for a high-end open-source car racing simulator (TORCS). We introduced a track representation that is an extension of the *radial representation* proposed in [94], and is based on a set of points expressed in a polar coordinate systems. We introduce two different approaches for evolving tracks: (i) the first one focus on the *maximization of the track diversity*; (ii) the second one aims the generate tracks which directly maximize the users' preferences.

The first approach, is based on the maximization of the track diversity both in terms of shape (e.g., the types and number of turns and straights) and in terms of achievable driving speed. In particular, the diversity is measured using the entropy of the curvature profiles and the entropy of the speed profiles. We performed three sets of experiments involving single-objective and multi-objective genetic algorithms to evolve tracks encoded using 5, 10, and 15 control points. Our results show that both single-objective and multi-objective approaches generate a wide variety of feasible TORCS tracks with a high degree of diversity both in terms of shape and in terms of achievable speeds. In particular, multi-objective evolution can successfully evolve rich Pareto fronts containing large numbers of feasible tracks which provide a good trade-off between the two competing objectives (i.e., having a rich structure while also providing an articulated driving experience in terms of achievable speed). The proposed approach can be used as a tool for quickly generating a large quantity of game content without human intervention.

In the second approach, we aim to evolve tracks which maximize the users' preference. For this purpose we proposed a web-based interactive genetic algorithm which can be used both in single-user and in multi-user mode. The implemented system includes two main parts: (i) a web frontend, and (ii) an evolutionary backend. The web frontend manages the interactions with anonymous/registered users who can work on their own population (in single-user mode) or can cooperate on a shared population (in multi-user mode). The evolutionary backend manages both the interactive genetic algorithms and all the tasks strictly connected to the target racing game (e.g., the generation of the actual in-game content, the rendering of the thumbnails needed by the web frontend). We validated the initial prototype of the framework with five human subjects who were asked to perform four single-user trials and one multi-user trial involving all the subjects together. The preliminary feedback we received was generally positive. In most cases, users perceived improvements in the quality of the tracks between subsequent generations. Users also stated that, at the end, the process produced some interesting tracks. The results suggest that the proposed framework can be used either as a tool for supporting game designer or as a system for generating game content based on the preference communities of users.

Part III.

Beyond Racing Games

9. Game Content Generation in First Person Shooters

In this chapter, we investigate the application of search-based Procedural Content Generation for evolving maps in First Person Shooters (FPS). Initially, we introduce first person shooters and the platforms publicly available for research. Then, we present *Cube 2*, the open source FPS that we used for the experiments of this work. Next, we introduce four types of representations for FPS maps and we present the fitness function designed for evolving maps with interesting gameplay. We present an experimental analysis to validate the proposed fitness and to compare the four representations of the maps. Finally, we extend one of the proposed representations to evolve fully 3D maps, i.e., maps with more than one floor.

9.1. First Person Shooters

First Person Shooters (FPS) are videogames where the player views the game world from the perspective of the character he is controlling, and where the gameplay involves both navigating into a complex three-dimensional environment and engaging in combat using weapons of different types. FPS are one of the world's most popular video game genre, with no signs of the popularity abating. *Call of Duty*, *Halo*, *Battlefield* and other similar game series sell millions of copies each year. Still, developers are plagued by the rising costs of developing content for such games, leading to shorter games and anxiety about creatively and thematically diverging from the mainstream. A typical modern FPS has less than ten hours worth of single-player campaign and just a few multiplayer maps, despite costing tens of millions of dollars to develop.

Computational intelligence techniques have been applied to FPS games before, but mostly to create NPC behaviour of various kinds. Among the most popular games used for research purposes we have *Quake* and *Unreal Tournament*. In particular, Unreal Tournament was used for *2k*

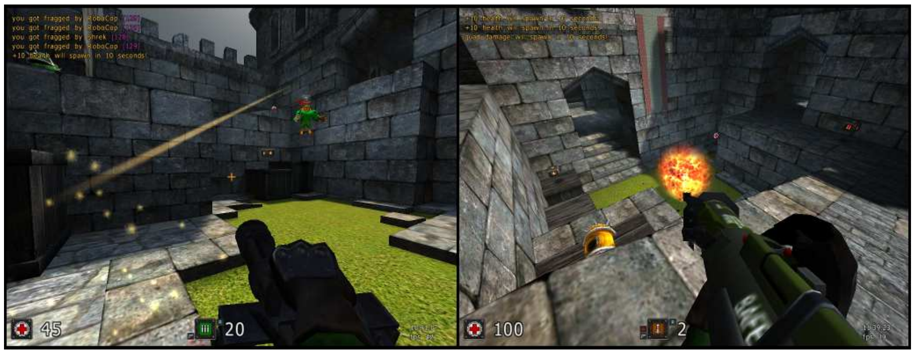


Figure 9.1.: Screenshots from Cube 2: Sauerbraten

BotPrize competition [30] where the goal is to design a bot that can be indistinguishable from an human player. Unfortunately, neither Quake nor Unreal Tournament have an open-source map editor which can be modified in order to automatically generate maps.

9.2. Cube 2

Cube 2: Sauerbraten [104] is a free open-source FPS game that supports both single- and multi-player gameplay. Cube 2 comes with several graphical character models, a set of weapons and a large number of maps. The engine underlying Cube 2 is fast and sophisticated, allowing smooth rendering of high-polygon environments, approximating the visuals found in commercial games (see Figure 9.1). Technically, the game engine is based on a 6 directional height-field in octree world structure which also supports lightmap-based lighting with accurate shadows, dynamic shaders, particles and volumetric explosions. Cube 2 also supports a simple but complete configuration/scripting language which allows customization of all menus and most other aspects of the game, and which makes it easy to build "mods" of the game.

One of the standout features of the engine is the capability for in-game geometry editing and also multi-player cooperative editing. The integrated map editor is very powerful: it allows the user to build and edit new shapes, to apply textures and to add objects of various materials (such as water, lava and glass), and to add light sources, among several other actions. Since Cube 2 is open source, the map editor itself can be modified and extended in whatever way necessary. This feature is crucial for our purposes, as we need to inject evolved maps back into the game, and it is one of the main reasons we chose to use Cube 2 rather than a better-known commercial FPS game. The other main reason is that the game engine allows us to run the game in "headless" mode, i.e. without visualization, where it can be speed up to run as fast as the processor permits.

9.3. Evolving Maps for a Multi-Player FPS

In this chapter, we apply evolutionary algorithms to evolve maps for a multi-player FPS game. A multi-player FPS is a game where several players (humans and/or bots) fight on the same map. Several game modes are possible with differences in rules; in the most basic mode, "deathmatch", the rules are simple: when a player character is killed, it will spawn in another point of the map after a few seconds. The game terminates after a fixed amount of time, and the player with most frags (i.e. the player which killed more opponents) wins the game.

A FPS map usually consists of a series of rooms and corridors plus several spawn-points and resource items (either weapons or bonuses). Maps may have several different levels with floors above and below each other, and features such as stairs, ramps and elevators for moving vertically between floors. In the first part of this work we will focus on maps with only a single floor.

The goal of this work is to evolve maps with potential for interesting gameplay. It is generally accepted that some FPS maps allow for more interesting and deeper gameplay than others, for example by rewarding skillful use of complex tactics, and by forcing players to vary their tactics so that they cannot use the same patent trick all the time to win. It is generally accepted that such maps are of better quality than others. Indeed, much work goes into exquisite balancing of those maps that are available as paid downloads for popular FPS games, to make sure that no single strategy will dominate the map. For brevity, we will in the following refer to maps with potential for interesting gameplay as *promising* maps.

9.3.1. Fitness function

Naturally, it is hard to design an accurate estimator of the promise of a map, as this will require predicting the preferences of a great number of players (e.g. as in [61]). In this work, we will settle on a simple theory-driven fitness function.

We assume that the promise of a map is directly linked to the *fighting time of the player*, which is defined as the duration from the moment in which the player starts to fight an opponent to the moment in which the player is killed. Since during an entire match the player will die and spawn multiple times, we can compute the average fighting time value for the game per player, T_f. A small T_f value means that a player gets killed very quickly after starting to fight an opponent. If the T_f value is large it means that after the player first gets damage in a fight, she survives and can prolong the fight because the map affords possibilities to escape, to hide, to find health bonuses or better weapons. Since an FPS map presents several features that, from a strategic point of view, can be exploited to engage the player in longer and more interesting fights the T_f value appears to be a good measure of the promise of a map.

The best way to assess the T_f value of a map would be to play several matches with human players on that map and collect the statistics, yielding an *interactive* fitness function according to the classification in [102]. Unfortunately, it is not practical to use human players in our experiments because it would require them to play thousands of matches (in the future, this might be a possibility using massively multiplayer games, similar to the approach taken in [29]). Instead we simulated matches of 10 minutes among 4 bots and we measured the average T_f value across all bots, $\overline{T_f}$, yielding a *simulation-based* fitness function according to the same classification.

The complete fitness function has another component in addition to the T_f value: the *free space of the map*, S. We explicitly want our fitness to promote the generation of larger maps since very small maps do not leave enough space for the placement of weapons and spawn-points, leading to unrealistic values of T_f. It is worth mentioning that the maximum size of a map is bounded and that in the best maps generated (see Section 9.5) S contributes to less than 20% of the total fitness value. Given the above, the complete fitness function of a map is as follows:

$$f = \overline{T_f} + S \tag{9.1}$$

where $\overline{T_f}$ is measured in milliseconds and it is an integer greater than 0, and S represents the number of free cells in the map and is bounded between 0 and 4096.

To evaluate maps using a simulated match we had to address two main points: generation of way-points and acceleration of the game. While for the latter the solution was as simple as disabling the graphical rendering (enabling the simulation of a 10 minutes match in about 10 seconds using a computer with an 3.00 Ghz Intel Core 2 Duo processor), for the first we had to implement a way-point placement algorithm. The bots available in Cube 2, like the ones in many commercial games, depend on a list of way-points to navigate in a given map. On that basis, points are represented in a graph on which bots apply the usual A^* algorithm to move between bonuses, weapons and enemies. Therefore, it was necessary to generate the way-points on the fly for each new map. Unfortunately, the game gives no support for automatic generation of way-points as they are usually placed by the human designer of the map. To overcome this problem we implemented an algorithm for way-point generation that follows these steps: (i) compute the free cells of the map; (ii) place a way-point on every free cell; (iii) connect each way-point with its four neighbors (up, down, left, right) if there are no obstacles between them; (iv) align every resource on the map to the grid of the way-points.

9.4. Map Representations

After the fitness function, the other important design choice in search-based PCG is how to represent the content. The representation must allow for the expression of a wide range of interesting content, but also enable the search algorithm to easily find content with high fitness. In this thesis, we propose and compare four different representations for FPS maps.

First, we need to distinguish between the *genotype* and *phenotype* for these maps. The phenotype is isomorphic to an actual map in the game. For all map representations, the phenotype structure is the same: a matrix of 64×64 cells. Each cell is a small piece of the map with a size suitable to contain a player character. Each cell can be either a free space or a wall (with a fixed height). Each free space can be empty or can contain a spawning point or a resource item (weapon or health bonus). The phenotype is saved in a text file and loaded in the game for playing using a specific loader that we implemented in Cube 2.

The structure of the genotype, on the other hand, is different for each representation. The genotype is what is evolved by the genetic algorithm. Each representation comes with a procedure for constructing a phenotype from the genotype; the simpler this procedure is, the more *direct* the representation is said to be. When a genotype is evaluated, the following happens:

1. the genotype is used to build the phenotype;

2. then the phenotype yields a specific map for Cube 2 and a simulated match starts in that map;
3. the statistics collected during the match are used to compute the fitness.

The four representations are described below in order of decreasing directness.

The most direct representation is named *Grid* and assumes that the initial configuration of the map is a grid of walls. In particular a 9 by 9 grid is used to divide the map into 81 squares. Each of the wall segments of a square can be removed to create rooms and corridors. The genome represents which of these wall segments are active (on) or not (off). According to the *Grid* representation scheme each gene encodes the state of two wall segments of the cell: the one on the top and the one on the right. Thus, each gene can take four possible values: 0, if both wall segments are off; 1, if only the top wall segment is on; 2, if only the right wall segment is on; and 3, if both segments are on.

The second, less direct representation is named *All-White*. It assumes that the initial map is empty (with walls only at the borders) and searches for fit maps by adding wall elements on the empty space. The genome encodes a number of wall blocks, N_w (N_w equals 30 in this work), each represented by three values, $< x, y, l >$, where x and y define the position of the top-left corner of the wall and l represents the length of the wall. If $l > 0$, the resulting wall is aligned to the x-axis; otherwise it is aligned to the y-axis. According to this representation, the width of each wall block equals 1.

The third representation is named *All-Black* and is, in a sense, the exact opposite to the *All-White* representation. This representation starts with an initial map full of wall blocks, and gradually adds free spaces in search of fitter maps. The genome encodes a number of free spaces of two types: the *arenas* and the *corridors*. Arenas are square spaces defined by the triplet $< x, y, s >$, where x and y represent the coordinates of the center of the arena and s represents the size of it. Corridors are rectangular-shaped free spaces with width fixed to 3 cells. Corridors are encoded in the same way as wall blocks via the three values of $< x, y, l >$. In the experiments presented in this work, the *All-Black* representation builds on 30 corridors and 5 arenas.

The most indirect representation is called *Random-Digger* and as the *All-Black* representation, it builds maps on an initial map configuration which is full of wall blocks. The genome encodes the policy of a very simple agent that moves around and frees up the space in every cell it visits, in a way reminiscent of turtle graphics. The resulting map is a trace of the path the agent followed for a fixed number of steps. The agent starts at the center of the map and its policy is represented by four probability values: $< p_f, p_r, p_l, p_v >$ where p_f is the probability of going forward along the current direction; p_r is the probability of turning right; p_l is the probability of turning left; and p_v is the probability of visiting an already visited cell. The last probability is very important since it controls the exploration rate of the digger.

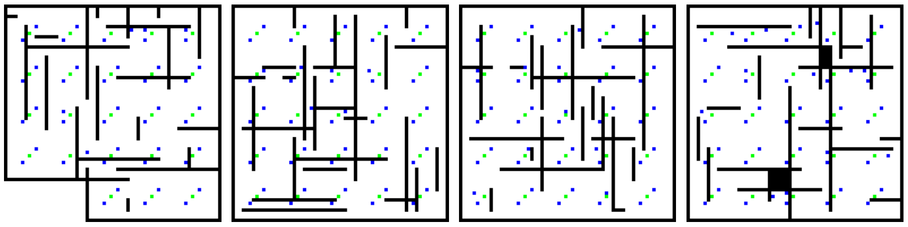
The first three representations can generate maps with some parts that are not reachable from the all other parts of the map. There are two main approaches to overcome this problem. The first approach attempts to repair the map so that it becomes fully connected. This solution has several drawbacks: it is complex to implement, it can be computationally expensive and it may heavily modify the initial shape of the map. The second approach focuses on simply removing the unreachable parts from the final map. In this work we follow the second approach by identifying all cells that were reachable from the center-point of the map and then remove all cells that are not reachable.

Before a complete Cube 2 map can be generated from the phenotype, we need to add spawning points, weapons and health bonuses. We do this through a simple uniformly-distributed heuristic as follows: (i) the matrix of the phenotype is divided into squared blocks; (ii) for each block the number of free cells is computed; (ii) if this number is bigger than a given threshold two spawn-points and one resource item (a weapon or a health bonus) are placed inside the block.

9.5. Experiments

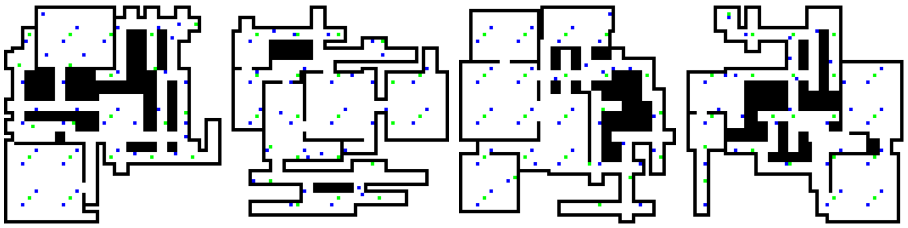
To evolve the maps, we applied a simple genetic algorithm with standard operators: tournament selection with tournament size 2, single point crossover, mutation rate of $\frac{1}{n}$ (where n is the length of the genome), mutation range 0.2 (following a uniform distribution) and a crossover probability of 0.2. The parameters of the genetic algorithm were set empirically. We applied the genetic algorithm with a population size of 50 individuals and let it run for 50 generations. Each evolutionary run took approximately 6 hours to completion and the experiment was repeated 10 times for each representation.

Figures 9.2, 9.3, 9.4 and 9.5 display the four highest performing maps evolved for each representation. The 2D image of each map illustrates walls as black segments, spawn-points as blue dots, and resource items



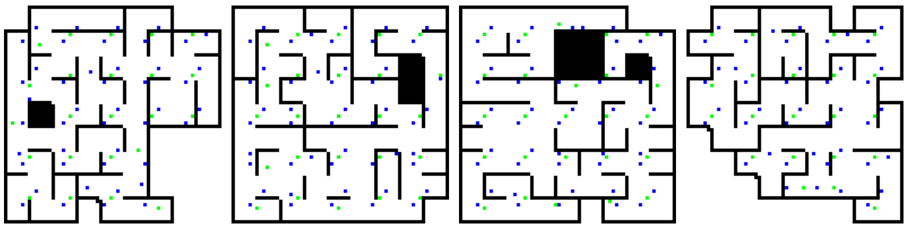
(a) $f = 20705$ (b) $f = 21146$ (c) $f = 21931$ (d) $f = 22882$

Figure 9.2.: Best Maps Evolved using Representation *All-White*



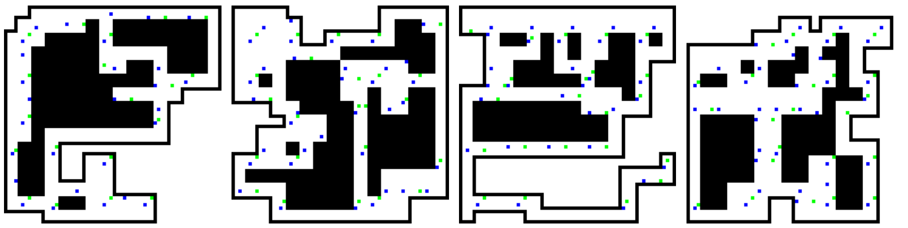
(a) $f = 14520$ (b) $f = 14530$ (c) $f = 15410$ (d) $f = 15801$

Figure 9.3.: Best Maps Evolved using Representation *All-Black*



(a) $f = 17282$ (b) $f = 18028$ (c) $f = 20812$ (d) $f = 22520$

Figure 9.4.: Best Maps Evolved using Representation *Grid*



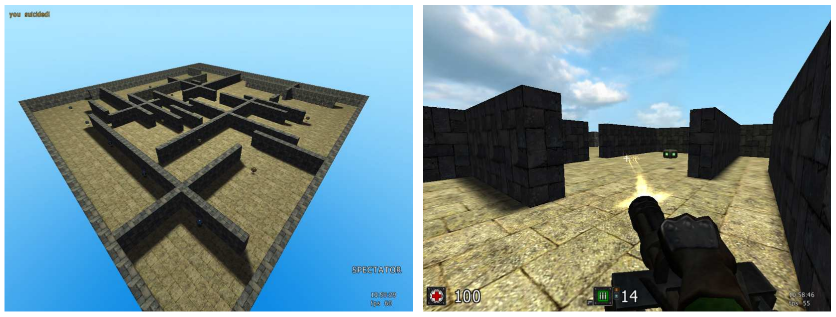
(a) $f = 12500$ (b) $f = 12964$ (c) $f = 12972$ (d) $f = 13202$

Figure 9.5.: Best Maps Evolved using Representation *Random-Digger*

(either weapons or health bonus) as green dots. Figure 9.6 illustrates how one of the best map evolved appear rendered in 3D when they are loaded in Cube-2. To test for significance, we run a t-test on the highest performances obtained via the 10 GA runs among all representations. According to the results: (i) *All-White* generates maps which are statistically better than *All-Black* and *Random-Digger* (p-value < 0.001 in both comparisons); (ii) *All-White* evolved better maps than *Grid* but the difference is not statistically significant (p-value $= 0.141$); (iii) *Grid* maps are statistically better than *All-Black* and *Random-Digger* maps (p-value < 0.001 in both comparisons); and (iv) the maps generated with *All-Black* are statistically better than the *Random-Digger* maps (p-value < 0.001). In addition, we performed experiments to test for the sensitivity of the fitness value by evaluating a map multiple times. Even though the variance of the fitness can be rather large and dependent on the map structure, the initial placement of bots and weapons and bot behavior the fitness order among the four representations is maintained and the statistical effects hold.

As can be seen from the results obtained, all the four representations are able to generate playable maps. Each representation allows the emergence of some peculiar features that strongly characterize the evolved maps. The *Random-Digger* representation generates maps with many long corridors and few small arenas. The *All-Blacks* representation instead, generates several bigger arenas while corridors are usually shorter and may present dead ends. The *Grid* representation generates very interesting maps with a high degree of symmetry. Finally, the *All-White* representation generates the best maps according to the considered fitness function. The high fitness values of *All-White* maps are justified by the coupling of many narrow passages with big arenas which generate many small spaces for a player to hide and trap its opponent or pick health bonuses after a fight.

It is worth noticing that the 2D top-down map images of Figures 9.2, 9.3, 9.4 and 9.5 may be misleading for some readers. For instance, the *All-White* maps are less symmetrical and aesthetically pleasing than the maps of the *Grid* representation; thus, one may infer the inferiority of the *All-White* maps with respect to their gameplay value. However, this aesthetic notion is reverted once the map is viewed and played from a first person perspective (see Figure 9.6) as it is confirmed by our preliminary results from a user study.



(a) 3D map overview (b) First-person perspective

Figure 9.6.: One of the best evolved map ($f = 21931$) loaded in the game *Cube 2*.

9.6. Towards 3D Maps

In the previous sections, we presented 4 different representations devised for 2D maps. In this section we present a more complex representation for fully 3D maps, i.e., maps with more than one floor.

9.6.1. Representation

A 3D map, is not just a collection of independent layers. If it was the case, a 3D representation is simply a genotype which replicates a 2D representation for the number of floors considered. Instead, in a 3D map, there is an important element to consider: the connection among different layers. For each pair of layers, there can be more than one connection, and each connection must connect two points that are both reachable from the respective layer. The connection can be bi-directional like stairs and elevators, or mono-directional like pitfalls (only down) and jump-pads (only up). In this work, we focus only on bi-directional elements, in particular on stairs.

In the design of an effective 3D representation we took inspiration from the Random-Digger which has the nice property of generating a path that by definition is always connected. In this way, we avoid the problem of considering unreachable points of the layers. The new representation, dubbed *3D-Digger*, is a list of movements that are applied in sequence to dig a 3D non-empty space. Thus, the genome encodes a sequence of movements. Each gene of the genome is a pair

$$< \text{XY-Direction} , \text{Z-Direction} >$$

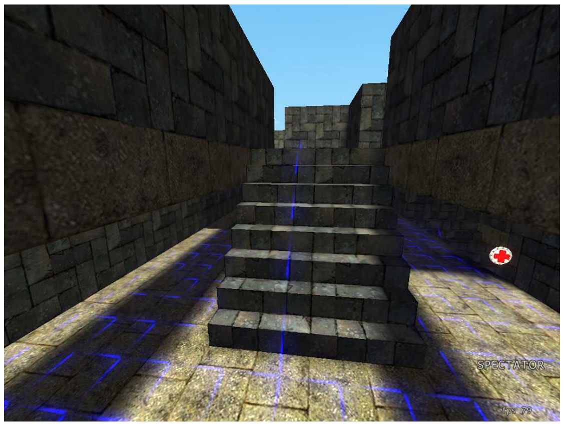


Figure 9.7.: 3D Maps: stairs

where XY-Direction is one of the four cardinal directions *N, E, S, W* while Z-Direction can be Up, Down, or Current respectively represented from the letters *U,D,C*. For example a possible genome can be: SU, SD, NC, etc.

9.6.2. Mapping

In the mapping process from the representation to the actual map, the movements listed in the genome are applied in sequence digging an empty corridor within a 3D cube of blocks. Every time an Up o Down movement is encountered, a small stair is added in the map (see Figure 9.7). An Up or a Down movement is allowed only when the final position is a non-empty block: in this way it is possible to prevent infeasible crosses between stairs. To avoid the generation of huge maps, we limited the number of layers to 3. Please note that the representation is independent from the number of layers: the limit of 3 layers is a constraint added during the mapping process.

After the generation of the map, it is necessary to add the waypoints which allow the bots to navigate the map. In the 3D case, the waypoints generation needs two steps. In the first step, waypoints are generated independently for each layer using the same mechanism presented in the previous sections. In the second pass, all the stairs are detected and then the waypoints are placed in order to connect the two extremities of each stair (see Figure 9.8).

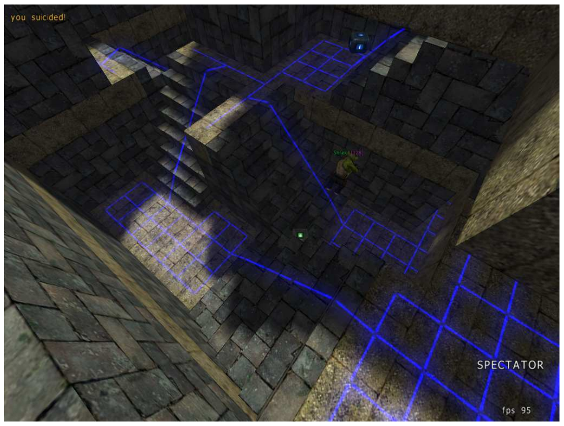


Figure 9.8.: 3D Maps: waypoints

9.6.3. Experimental Results

In this section, we present a preliminary experimental analysis in which we test the proposed representation for 3D maps with the fitness function already introduced in the previous sections. Since the computational cost for generating a 3D map is too heavy, we used a genetic algorithm with population size equal to 10. The evolutionary process terminated after 10 generations.

To get a more accurate fitness we evaluated the maps for more than one time. To speed-up the re-evaluation process we used a techniques similar to the strategy presented for on-line learning. If the the fitness of a map is less than the current best fitness, the map is evaluated only one. If the fitness of a map is the best achieved so far, the map is evaluated again for a total of 4 evaluations. In this way, we reduce the computational cost but we are always sure that the fitness of the best map is very accurate.

Even if a limited population was used, the genetic algorithm is able to effectively improve the fitness of the initial random population. In Figures 9.9, 9.10, 9.11, 9.12 we show some screenshots of one of the best map evolved. It is possible to see that the representation and the fitness used are able to generate maps with a very interesting and complex structure: the layers are connected by more than one passage and some layers are divided in two distinct parts, reachable through the other layers.

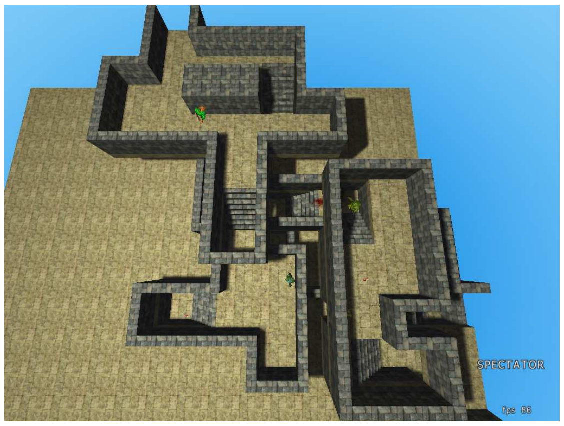


Figure 9.9.: One of the best 3D Maps evolved: top view

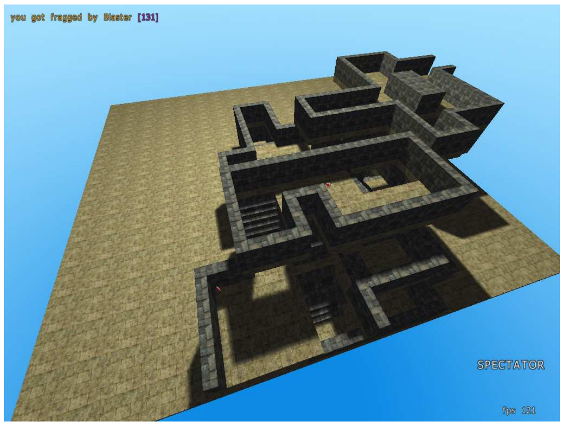


Figure 9.10.: One of the best 3D Maps evolved: 3D view

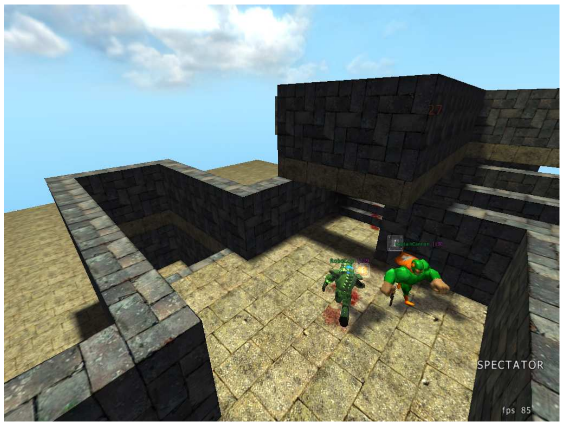


Figure 9.11.: One of the best 3D Maps evolved: zoom

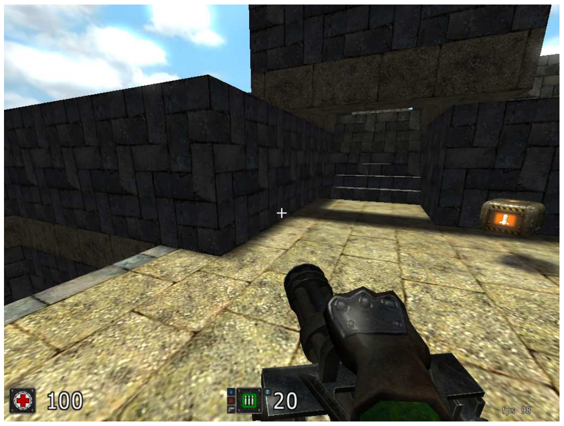


Figure 9.12.: One of the best 3D Maps evolved: first person view

9.7. Summary

In this chapter, we applied search-based PCG to evolve maps in a Multi-player FPS. This problem, at the best of our knowledge, was never tackled in the literature.

We have devised four different representations for multi-player maps, and a fitness function for evaluating their potential for interesting game-play. We have also performed several experiments to verify that we can evolve playable FPS maps using these representations and the fitness function. Further, we have used the fitness function to compare the promise of maps evolved with each of the representation. All the representations are able to generate playable maps and the fitness pushes towards maps with interesting gameplay. This results was confirmed also from an on-going user study. From the results, it seems that the All-White and Grid representations have clear advantages in this respect.

Finally we showed how the representations and the fitness presented for 2D Maps can be extended to evolve fully 3D maps with more than one layer. The representation for the 3D maps takes inspiration from the Random-Digger. From the preliminary experimental analysis, it is possible to see that the evolved maps show interesting and complex structures.

10. Adaptive Game Content Generation

In this chapter, we investigate game content generation using a user preference model. In particular we study how an existing preference model can be adapted to maximize the game experience of a given user. To test our approach we used Super Mario, a very popular platform game. Our analysis starts from an existing preference model for Super Mario developed in a previous work that involved an experimental study with several human subjects. The preference model can be used to generate levels which maximize the player fun. The main problem of this approach happens when the current user is much different from the users involved in the model training. To deal with this problem, we present our approach which applies on-line adaptation to customize the model on the current user. Then, we present the experimental setup that we designed in order to compare game content generation with and without adaptation of the preference model. Finally we present the experimental results which involve AI-controlled players for testing some aspects of the adaptation process and human players for a final validation of the approach.

10.1. Super Mario Platform Game

The testbed platform game used for our study is a modified version of Markus Persson's *Infinite Mario Bros* (see Figure 10.1) which is a public domain clone of Nintendo's classic platform game *Super Mario Bros*. The original Infinite Mario Bros and its source code is available on the web.

The gameplay in Super Mario Bros consists of moving the player-controlled character, Mario, through two-dimensional levels. Mario can walk and run, duck, jump, and shoot fireballs. The main goal is to get to the end of the level. Auxiliary goals include collecting as many coins as possible, and clearing the level as fast as possible. While implementing most features of Super Mario Bros, the standout feature of Infinite Mario

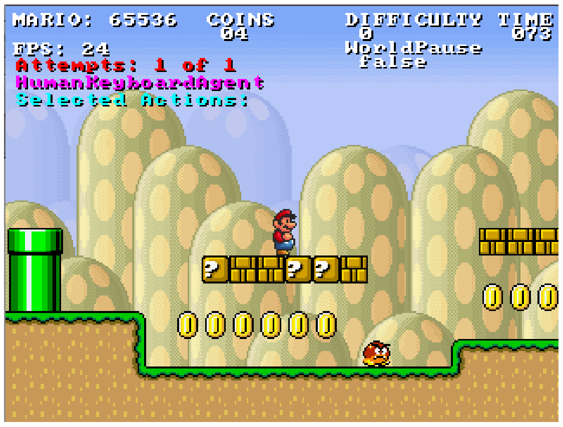


Figure 10.1.: Infinite Mario Bros game screenshot

Bros is the automatic generation of levels. Every time a new game is started, levels are randomly generated by traversing a fixed width and adding features according to certain heuristics as specified by placement parameters.

10.2. Level Generator

On top of the level generator of Infinite Mario Bros, we used a more abstract level generator. This level generator, takes as input some high level parameters, and then generates a random level which try to respect the constraints defined in the parameters. In particular, the level generator used in this work, takes as input 3 parameters: (i) the number of gaps; (ii) the average length of the gaps; (iii) the entropy of the gaps distribution. These parameters allows to focus on the high level structure of the level. These parameters represent the controllable features, i.e., the features of a level that can be directly modified. Instead, the non-controllable features cannot be directly controlled by the level generator. E.g., the number of jumps, is a non controllable feature which mainly depends on the player behavior: however the number of gaps indirectly influences the number of jumps.

10.3. User Preference Modeling

As already introduced in Chapter 3, there are several approaches for evaluating the game content. One of these approach consists in collecting data from a reasonable number of users for building a model of the user preferences. Such a model can be later used for generating content which maximize the user's preferences. Building a model of user preferences, is a very expensive activity since it requires to involve many users that are willing participate to one or more test sessions. For this work, we will start our study from a model of users fun in Super Mario developed in a previous work [76]. In this section, we firstly describe the process for building the model presented in [76] and then we show how such model can be used for content generation.

10.3.1. Initial User Preference Model

In this work, we will use as starting point a model of user preferences which comes from a previous work published in [76]. In [76], Shaker et. al. use human players' data to build models for predicting reported fun in the game *Infinite Mario Bros.* The data was collected from 327 players during a total of 1308 game sessions. For each game session 3 types of data were collected:

1. Controllable features of the game: These parameters are used for level generation, and affect the type and difficulty of the level. This set contains three features that are related to gaps: number of gaps, average width of gaps, and gap entropy.

2. Game-play features: Statistical features of how the user plays the game such as how often the player jumped, ran, died, how much he spent moving left, and how many enemies he killed for the different type of opponents. These features cannot be directly controlled by the game as they depend on the player's skill and playing style.

3. Player preferences: After playing a set of four games, divided into two pairs played in both orders, players were asked to report their preferred game for three user (emotional) states; fun, challenge and frustration, through a 4-alternative forced choice questionnaire protocol.

A set of 58 features was extracted during each game session: 3 controllable and 55 game-play features. In order for the user models to be easy to analyze and construct, sequential forward feature selection (SFS) was applied to extract the minimal feature subsets that yield the highest performance. For each emotional state, the performance of the models was measured through the average classification accuracy of a single layer perceptron (SLP) in three independent runs using 3-fold cross validation.

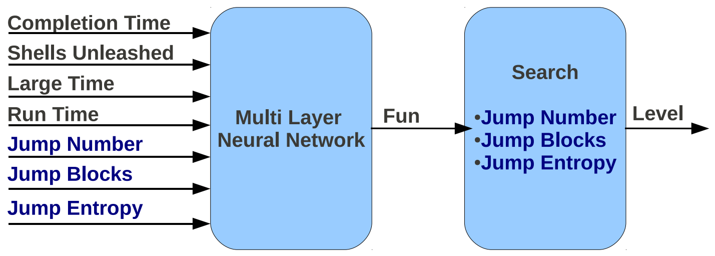


Figure 10.2.: Original approach which use the fun model to search for the best level

Since the purpose of the model is to generate content, all controllable features are forced into the input of a multi layer perceptron (MLP) model and the topologies of the networks are optimized for maximum prediction accuracy. The trained models are able to predict the player's fun preferences with an acceptable accuracy and are able to generalize well across larger sets of data. The prediction accuracy achieved using this approach to predict fun was 69.66%.

10.3.2. Generating Game Content

Once a user preference model is given, it is possible to automatically generate game content that is tailored to player in this way: (i) the input of the model related to non-controllable features (eg. jumps) are set using an estimation based on player's performance of the previous level; (ii) for the remaining input related to the controllable feature, an exhaustive search is performed to find the best features that maximize the output of the model. In this way, the user can play a level that maximize the fun prediction.

Figure 10.2 outlines this approach when it is applied to optimize the fun of the player. The fun model has 7 inputs: 4 non-controllable features (Completion Time, Shell Unleashed, Large Time, Run Time) and 3 controllable features (Jump Number, Jump Blocks, Jump Entropy). The player's reported fun is optimized using exhaustive search on the three controllable parameters that will define the structure of the level to be presented to the player.

10.4. User Model with Dynamic Adaptation

The main drawback of the user preference modeling is that the resulting models are fixed and represent the average player of the game. The models are trained off-line with a data-set of user preferences, and then the models are used for content generation. What happens if a particular user is different from any user preference patterns trained and embedded in the constructed model is that the model will most likely perform poorly for that user. This is a valid hypothesis since the model is built on a limited number of users that cannot cover all the possible user types the user model may face.

User learnability and interaction dynamicity effects are not embedded in the model creation either. Typical game users (players) most likely change the way the play over time but also the way they react emotionally with a game over time: for instance, during the first levels of a game the player focuses on some gameplay aspects and her playing behavior is characterized by particular play patterns while she prefers dissimilar elements and follows a different playing style in the game levels that follow the initial experience. For that purpose, it is necessary to employ an on-line (i.e. during play) learning mechanism that will continuously adapt and tailor the pre-built user model to each particular user. For the user model to be self-adjusted a user feedback signal is required. In general, real-time user experience can be captured and user feedback can be acquired in three ways [115]: (i)subjectively, by asking the user directly; (ii) objectively, via physiological signals; or, (iii) by inferring the feedback signal from interaction i.e. game-play statistics.

In this work we focus on the subjective way of capturing user experience. In particular, at the end of each Super Mario level, the game pops up a window asking the player to give a fun score to the level; the score value lies between 0 and 10. This reward signal is compared to the current output of the existing model and the difference error is used to update the model. Using this approach, the model is continuously updated after each level is completed and is adapted to the user. So the initial model trained with the collected data, is gradually transformed to a new model that predicts the player's reported fun with higher accuracy. This approach is outlined in Figure 10.3. Since the model considered is represented through a multi layer neural network, the adaptation mechanism used in this work relies on the standard back-propagation algorithm.

The adaptation process of the preference model is based on a table of

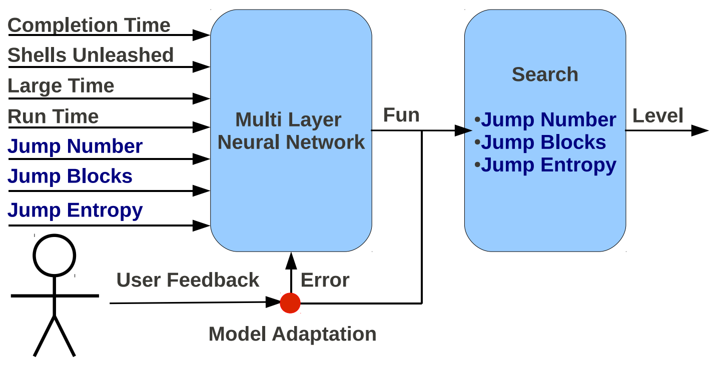


Figure 10.3.: The proposed approach which use the user's feedback to adapt the fun model

samples. A sample is a pair

$$< \text{model-inputs} , \text{user-score} >$$

where the inputs of the model are listed in Figure 10.3 and represent the controllable and the non-controllable features related to the last level. When a level is completed, the following steps occurs:

1. The user gives a score to the level
2. This score with the controllable and non-controllable features constitute a sample
3. The current sample is added to the samples-table
4. All the elements of the samples-table are back propagated in the model
5. The adapted model is used to generate the next level

In this way, we have a model that is gradually shifted to a model that is a better estimate of the preference of the current user. With a more accurate model, it is possible to generate levels that can maximize the user's fun. A samples-table is necessary in order to remember also the previous user's feedbacks. If we back-propagate only the last sample, the previous feedbacks can be gradually lost.

10.4.1. Extending the Model

The off-line feature selection process selects a subset of features that are important for the construction of the user model relying on the users represented by the collected data. However, the model may be missing several features that might be relevant for the prediction of the preferences of a particular user. It is also possible that the game level generator is extended with new patterns making it necessary for the user model to take into account new controllable features. In this section we examine how an existing user model can be further tailored to the particular user by extending both the vector of user characteristics considered and the controllable features used.

To consider new input features in the user model, it is possible to include these features into the input vector of the neural network and connect them with connection links of zero weight. In this way the original model is not altered by the new inputs. Only if there is a correlation with the user feedback the weight of links will be updated and the new inputs will influence the output of the model. If the new input added to the model is a controllable feature, it also possible to add such feature to the search module which generate the level.

The case study considered for validating this approach, is presented in Figure 10.4. In this case study we added two non-controllable features (Coins Collected, Fire Time) and one controllable feature (Enemies Number).

10.5. Experimental Design

In this section we describe the experimental setup that we designed to evaluate the proposed approach. In particular, our aim is to compare our approach, where the preference model is adapted on-line, with the original approach presented in [76] where the model is fixed.

According to the experimental setup, each participant plays two sequences of game levels: in one sequence, the content is generated using the fixed user preferences model which comes from the work presented in [76]; in the other sequence, the content is generated using a model that is continuously self-adapted to the player. To avoid any baseline effect, the first game level of both sequences is the same. This level is fixed for every experiment and represents the level generated by the average values of all controllable features.

At the end of each level of the sequence, the user is asked to score the fun experienced in the level (the score value lies between 0 and 10). Thus,

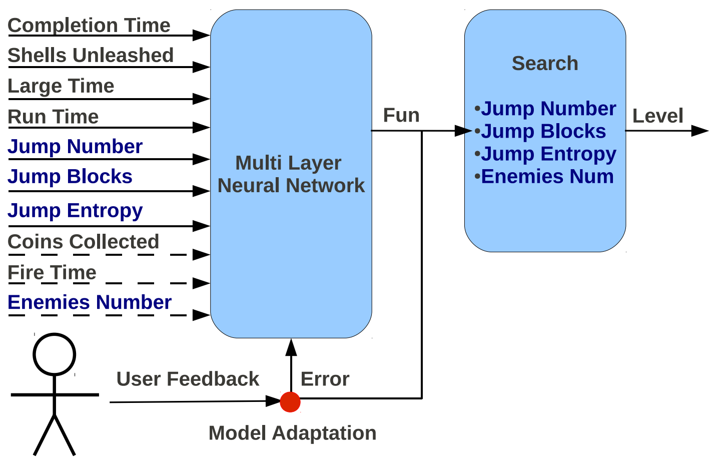


Figure 10.4.: The proposed approach extended with new user inputs and one additional controllable feature

for each sequence of game levels we collect a sequence of reported fun scores $f_1, f_2, ..., f_n$ which is used to compute the average fun score. This value was used as the performance measure for comparing the considered approaches.

We desire the adaptation mechanism to be robust but also real-time efficient. In order to test and evaluate those performance criteria the sequence of levels should be sufficiently long enough for any safe conclusions to be sketched. On the other hand, the n value should represent a human-realistic and a plausible gameplay time window. Thus, we firstly test the approach with AI-controlled simulated players for 50 games and then with human players using a smaller time window of 5 game levels. The following two subsections present the protocol design differences between the two experiments.

10.5.1. AI-controlled Players

This experimental setup aims to test the effectiveness and robustness of the adaptation process across different AI-controlled players and within lengthy (and human-unrealistic) gameplay time windows. To replace a human player and test the adaptive mechanism appropriately two main components are required: (1) a playing behavior, that is the sequence of actions that the player performs in a given level and (2) a user preference model, that is an ad-hoc computational model of reported user preferences utilized by the AI-controlled agent to report a fun score at the end of the level.

Playing Behavior

The playing behavior of the AI-controlled player was taken from the submission of the 2009 edition of the Mario AI Competition[1]. We selected the agent developed by Sergio Lopez. This agent is based on a relatively simple heuristic function that decides when to jump and how high, and otherwise walks left. It was able to complete most of the levels in the competition and its behavior resembles the human playing style much better than the other submissions.

Ad-hoc Player Preference Model

The fun preference model of the AI agents is a function designed ad-hoc which maps some game-play statistics to a score. We devised nine simple linear fun models, each linked to one single level or gameplay feature, which are presented in Table 10.1. For example, the fun model of User-1 (see Table 10.1) implies that type of player is having more fun in game levels with many gaps. Thus, the reported fun score of this player is directly and solely related to the number of gaps placed within the level. The first three users are linked to the three controllable level features which are part of the preference model; User-4, User-5, User-6 are linked to non-controllable gameplay features which are included in the preference model. The remaining three users are linked to features that are not the part of the normal model but they are considered only in the extended model.

10.5.2. Human Players

The procedure followed for human players is similar to the one followed for AI-controlled players. One difference is the length of the experiment which, as mentioned earlier, in the human experiment is two sequences of 5 game levels; one sequence with a fixed user model and one sequence with model adaptation. To minimize any possible order (of play) effects the starting level sequence is selected randomly. In contrast to the AI-controlled players experiment, human players are asked to play a test level before they start the experiment. That level is used to estimate the initial non controllable features that are necessary to the level generator.

[1]www.marioai.org

Table 10.1.: Ad-hoc AI-Player Preference Models of Fun

Player ID	Feature linked to reported fun
User-1	Gaps
User-2	Gaps Width
User-3	Gaps Entropy
User-4	Completion Time
User-5	Large Time
User-6	Run Time
User-7	Coins Collected
User-8	Fire Time
User-9	Enemies

10.6. Results

In this section, we present the results of the experimental analysis in which we compare two approaches for content generation: (i) the first, is the approach proposed in this work which is based on the dynamic adaptation of user preference model; (ii) the second, is the approach presented in [76] which involves a fixed user preference model without adaptation. In particular, in the first part of our analysis we consider AI-controlled players, while in the second part we validate the results with human players.

10.6.1. AI-controlled players

Figure 10.5 reports the results of experiments with the AI-controlled agent incorporating all the 9 dissimilar fun preference models. Each experiment is repeated 10 times and the performance (i.e. the average fun scores during all 50 levels) is averaged across the 10 runs. For comparison purposes we consider two additional baseline content generation approaches into account: the first generates a sequence of random levels (Random) while the second generates a sequence always equal to the same initial level (Fixed).

As shown from Figure 10.5 when the user fun preference model is related to a controllable feature (three left-most columns) the adaptation mechanism performs higher compared to both the no-adaptation mechanism and the other two baseline mechanisms. On the contrary, when the fun preference-model is related to a non-controllable feature the adaptation mechanism performs better than the approach without adaption but

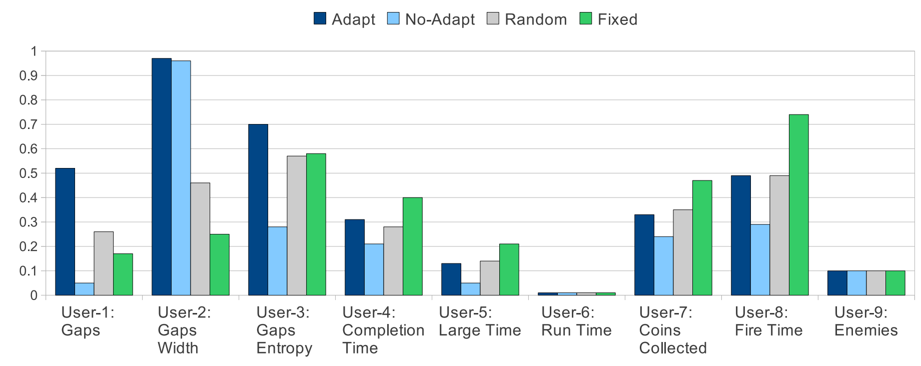


Figure 10.5.: Adapting the model to AI-controlled users

both perform either equally well or worse than the baseline mechanisms. This happens because, even if the approach with adaptation learns the user preference, there is no way to modify the level accordingly to a non-controllable feature. So in this case, the performance of a simple approach that does not try to learn the user preferences, are better.

Figure 10.6, as example, reports the details of the experiment related to User-1. For each methods, we report the score achieved in each level of the sequence. It is possible to see that the approach with adaptation learns and improve the user score very quickly: the learning is noticeable already in levels 4 and 5 and after level 18 is almost finished. Instead the other three methods oscillate around a fixed value.

In the second part of the analysis we consider the performance of the approach based on the extended model. The extended model depicted in Figure 10.4, has the following difference with respect to the previous model: (i) 2 non-controllable features are added as new inputs of the network: Coins Collected and Fire Time; (ii) a new controllable feature is added to the inputs of the model and to the search module: Enemies Number.

The results of the comparison are reported in Figure 10.7. If we take in consideration the 6 features that belong to both models, it is possible to see that the performances are very similar: so the new extra inputs add very few overhead to the learning process. If we take in consideration the 2 new non-controllable features we see that the performance of the extended model are slightly better. If we take in consideration the new controllable features we see that the performance of the extended model are much more better. This is quite an interesting result, since

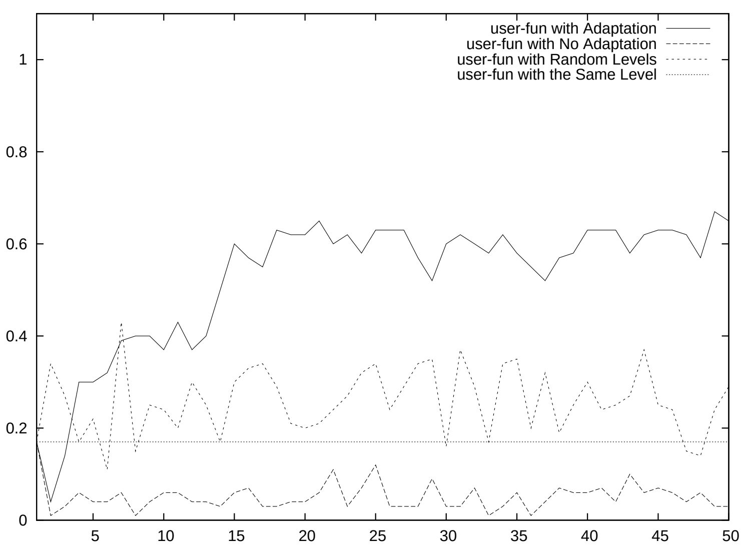


Figure 10.6.: Adapting the model to the AI-controlled User-1. The results are averaged over 10 runs.

we can reliable exploit the previous knowledge gathered during the data collection but in the same time capture the preference of the new users that likes the new features added to the level generator.

10.6.2. Human players

In this section, we present the results of the experimental study which involved human players. Users were recruited among students during some academic events (either at Politecnico di Milano and at IT-University of Copenaghen) and by spreading the test game through the Internet. Unfortunately, we were able to involve only a small number of users. However, the experiments are still ongoing, in order to achieve a better validation of our results.

The data that we present in this work was collected from a total of 48 users: 21 users participated to the experiment with the basic model and 27 users participated to the experiment with the extended model. Each user, played one sequence with model adaptation and one sequence without adaptation. For each sequence, we took into account the score reported from the user at the end of each level (in this case 5 levels) and we computed the average. We used the average score to asses if a user prefer the sequence with adaptation or the sequence with no-adaptation.

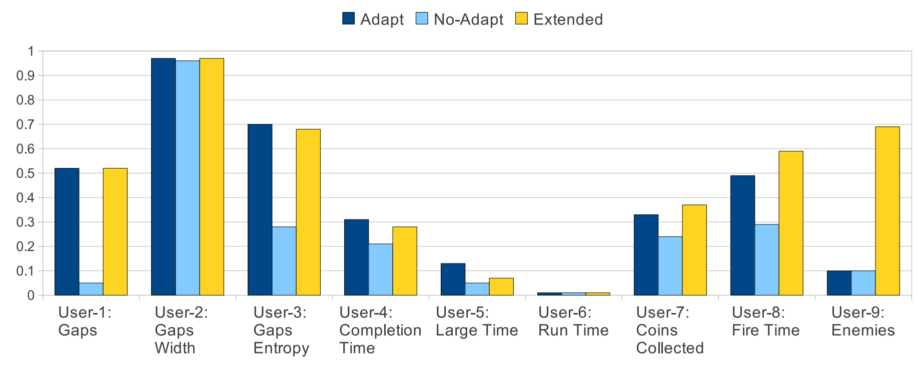


Figure 10.7.: Adapting the extended model to AI-controlled users

The results are presented in Table 10.2 in which we report, for each model type, the number of users which (i) prefer adaptation, (ii) prefer no-adaptation, (iii) equally prefer both techniques. It is possible to see that the majority of users prefer the approach which use adaptation both with the basic model and the extended model. A very small number of users reported the same average score for both techniques. In this case, the difference between adaptation and no-adaption is not so strong as in the case which involve AI-controlled players. This was partly expected, as every experiment which involves human is characterize from a certain amount of noise: loss of focus, fatigue, difficulty in giving consistent scores, etc. If we consider both the basic and the extended model together, the results are stronger: 28 users against 17 experience a greater fun with model adaptation. If we do not consider the 3 users which equally like both techniques, the model adaptation is positive correlated with the user experience. The correlation coefficient is 0.24 and it is significant with p-value 0.067578. Even if, a more accurate evaluation involving more users is required, the preliminary results suggest the our approach works: it is effective to adapt the model and to improve users' fun within a short sequence of levels.

10.7. Summary

In this chapter, we used a user preference model to generate game content which can maximize the player's fun. In particular we analyzed the problem of adapting an existing model to a given user.

The preference model that we used comes from a previous work which

Table 10.2.: Analysis of the users' data. For each model type, the table reports the number of users which (i) prefer adaptation, (ii) prefer no-adaptation, (iii) equally prefer both techniques.

Model Type	**Adaptation**	**Equal**	**No-Adaptation**
Basic	12	2	7
Extended	16	1	10
Total	28	3	17

involved many user for collecting game data. The model is a neural network which maps some feature of the level and some gameplay statistic to a score. This model can be used to evaluate the game content when searching the space of all possible level: the outcome is a level which maximize the score of the model, and so the expected fun of the players.

However, this approach rely on the quality of the model. In particular, there is a problem every time a given player is much different from the users represented from the model. To overcame this problem, we presented our approach for adapting the model, to a given user. The adaptation mechanism is essentially based on a direct feedback received by the user at the end of each level. Since the model is represented by a neural network we use the back-propagation algorithm to adapt the current output of the network to actual score expressed by the user.

To test our approach we performed two kind of experiments. In the first one, we used AI controlled players with 9 simple linear models of fun. In this case, the performance of the approach with adaptation are much better than the approach without adaptation. In the second case, our analysis involved human players: the majority of users likes adaptation, but the difference is less strong. In conclusion, the experimental results suggest that our approach is very promising for the generation of personalized game content but additional investigation with a bigger number of players is still needed.

11. Conclusions

In this thesis, we investigated two important topics for the design of computer games: evolution of non-player characters and content generation.

For the evolution of non-player characters, we investigated three aspects: off-line learning, on-line learning and transfer learning. In off-line learning, we considered the problem of developing a competitive driver. Our approach consisted of applying a behavioral decomposition to learn each of the behaviors separately. The final driver is the result of the combination of the evolved behaviors. The results show that our driver outperforms all the other drivers considered in the comparison. In on-line learning, we considered the task of quickly learning to drive a given track. We presented several selection strategies for achieving a fast learning process which would also mitigate the effect of poor performing individuals. Results show that our approach is better than off-line learning, both in terms of learning speed and final performance of the evolved behaviors. Finally, we considered transfer learning across two racing games with different engines and car dynamics. The results show that transfer learning across games of the same type is possible and is a promising way to quickly develop NPCs starting from knowledge already available in another game. All the aforementioned works are based on neuroevolution which showed itself to be effective with all the learning approaches investigated. In the field of racing games, neuroevolution is a promising approach which could be used in commercial games, both to quickly develop the AI of drivers and for developing sophisticated drivers which can adapt on-line.

For content generation, we applied search-based procedural content generation to different kinds of games. In particular, we took into account three game types: racing games, first person shooters, and platform games. In racing games, we presented our approach for evolving tracks. We introduced a representation for the tracks and we tested two different methods for evaluating their fitness. The first is a theory-driven fitness which aims to evolve tracks with a good amount of variety and challenges. The second fitness comes directly from the judgment of human users and involved the design of a framework which implements an interactive genetic algorithm. Both methods resulted to be able to generate interesting game content. In first person shooters, we introduced four representations for single-floor multi-player maps. We also designed a theory-driven fitness function for evolving maps with interesting game-play. Then, we extended the proposed representations to maps with more than one layer. The results show that the proposed approach is able to generate maps with interesting game-play. The work on platform games, is a sort of wrap-up since it combines content generation with on-line adaptation. In this work, a user's preference model is dynamically adapted to a given player, while it is used to generate content which maximizes the fun. The results, which involved an experimental study with human players, suggest that the adaptation mechanism can be effectively used to improve the game experience.

In conclusion, the results suggest that the approaches presented in this thesis for the evolution of non-player characters and for content generation are almost mature enough to be used in commercial games.

11.1. Future and Ongoing Works

In the following we briefly present the main research directions we are currently pursuing.

11.1.1. Additional analysis of on-line learning

In chapter 6, we investigated on-line learning in a driving task. In particular we considered two problems: (i) learning a new driving policy from scratch; (ii) learning to drive a new track starting from an existing driving policy. In both experiments the environment does not change during the race. We are interested in studying on-line learning when some aspects dynamically change during the experiment. Some aspects which can change dynamically can be the friction of the road bed, engine-power or braking force. These changes can reflect real situations in a game: friction can change because of weather conditions, e.g., the roadbed is wet, while engine-power or braking force can change because of car damages. In particular we are interested in studying how quickly the driving policy can adapt to the new conditions and what happens if some factors change more than once.

11.1.2. Data driven models of users preferences

Undoubtedly, data-driven models of user preferences are a very powerful way for generating game content which can improve the user's fun. Unfortunately, they are very difficult to build as they require the participation of many players for one or more test sessions. We are trying to investigate an alternative way to generate user models. The idea is to exploit the large amount of data available for some on-line games, in which for each map it is reported users' scores and gameplay statistics.

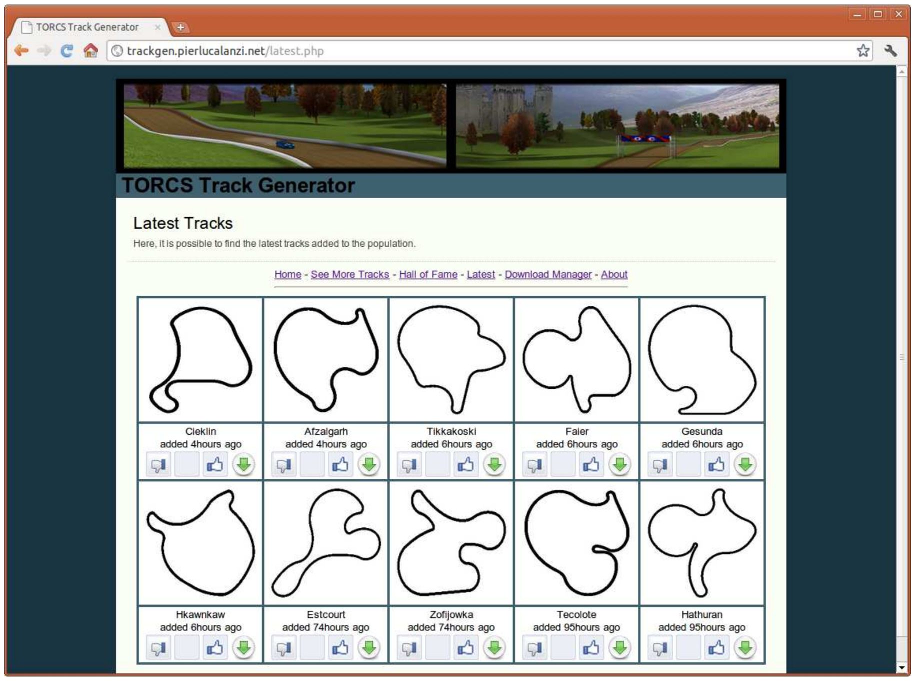
11.1.3. Non-playable Game Content

In this thesis, we mainly focus on the evolution of playable game content such as tracks and maps. As a matter of fact the non playable content also has a big influence on the player experience. Our goal is to investigate the effect of the non-playable content on the game experience. In particular we are investigating which are the most relevant factors: e.g., lights, sounds, textures, decorative objects, etc. The final purpose is to also evolve the non-playable content using data driven fitness functions or interactive genetic algorithms.

11.1.4. Community based Interactive Genetic Algorithms

Some games have a big community of users which support the game by organizing competitions or sharing new content. Our idea is to extend, the framework of the interactive genetic algorithm presented for evolving tracks and implement a real website which can be used by the TORCS community. In this way the generation of new tracks would be based on a wide number of users which could speed-up the generation and allow for a more robust evaluation. However, the implementation of a website for a community based evolution poses additional challenges: (i) all the users are not connected at the same time; (ii) the users will not interact with the system for the same amount of time, i.e., they will evaluate a different number of tracks; (iii) the user may want to download and play the track before giving a score.

A preliminary version of the web system, which implement the proposed interactive genetic algorithm, is available at: *http://trackgen.pierlucalanzi.net*. This version is based on a population of 100 tracks and a steady state evolution is used to update the current population. The home page shows only a part of the entire population and a button is used to browse all the available tracks. For each track, the user can give a score or can download and play the track. The track scores are used as fitness for the evolutionary step. The evolutionary step is automatically triggered after a fixed number of evaluations. To protect new individuals and to allow a uniform evaluation of all tracks we introduced the *number of views* of a track. Each track is associated with a number of views: this number is incremented every time the track is shown in a web page. In order to protect new individuals, a track can be removed from the population only if it has a minimum number of views. The home page always shows the tracks with the least number of views. In this way it is possible to quickly evaluate all the population. The system allows to browse the most recent tracks added to the population (see Figure 11.1) and the best tracks evolved until that moment (see Figure 11.2). The system is currently running to collect the data of the user activity and the evolved tracks over a wide period of time.

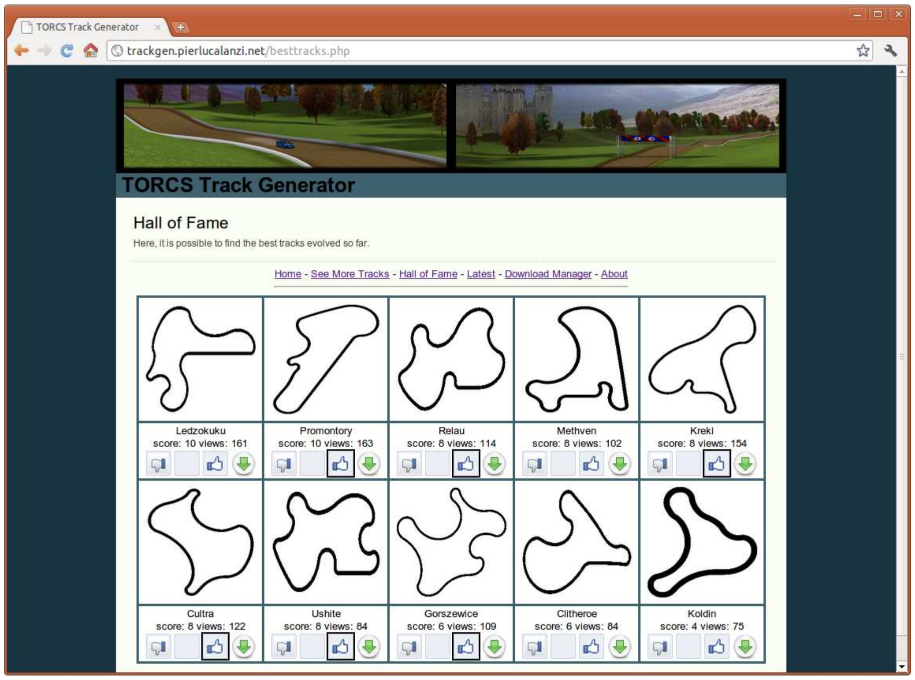

TORCS Track Generator
trackgen.pierlucalanzi.net/latest.php
TORCS Track Generator
Home - See More Tracks - Hall of Fame - Latest - Download Manager - About

Latest Tracks

Here, it is possible to find the latest tracks added to the population.

Cieklin added 4hours ago	Afzalgarh added 4hours ago	Tikkakoski added 6hours ago	Faier added 6hours ago	Gesunda added 6hours ago
Hkawnkaw added 6hours ago	Estcourt added 74hours ago	Zofijowka added 74hours ago	Tecolote added 95hours ago	Hathuran added 95hours ago

Figure 11.1.: Recent tracks in the web version of the IGA



TORCS Track Generator

Hall of Fame

Here, it is possible to find the best tracks evolved so far.

Home - See More Tracks - Hall of Fame - Latest - Download Manager - About

Ledzokuku score: 10 views: 161	Promontory score: 10 views: 163	Relau score: 8 views: 114	Methven score: 8 views: 102	Kreki score: 8 views: 154
Cultra score: 8 views: 122	Ushite score: 8 views: 84	Gorszewice score: 6 views: 109	Clitheroe score: 6 views: 84	Koldin score: 4 views: 75

Figure 11.2.: Best tracks in the web version of the IGA

Bibliography

[1] Alexandros Agapitos, Julian Togelius, and Simon Mark Lucas. Evolving controllers for simulated car racing using object oriented genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1543–1550, New York, NY, USA, 2007. ACM.

[2] Wolf-Dieter Beelitz, Xavier Bertaux, Brian Gavin, Eckhard M. Jaeger, Kristóf Kály-Kullai, Gábor Kmetykó, Enrico Mattea, Jean-Philippe Meuret, Haruna Say, and Andrew Sumner. Speed dreams - a free open motorsport sim and open source racing game. `http://www.speed-dreams.org/`.

[3] A.M. Brintrup, J. Ramsden, H. Takagi, and A. Tiwari. Ergonomic chair design by fusing qualitative and quantitative criteria using interactive genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 12(3):343 –354, june 2008.

[4] Bobby D. Bryant. *Evolving Visibly Intelligent Behavior for Embedded Game Agents.* PhD thesis, Department of Computer Sciences, University of Texas, Austin, TX, 2006.

[5] Michael Buro. From simple features to sophisticated evaluation functions. In H. Jaap van den Herik and Hiroyuki Iida, editors, *Computers and Games*, volume 1558 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 1998.

[6] Martin V. Butz and Thies D. Lonneker. Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 317–324, Sept. 2009.

[7] L. Cardamone, D. Loiacono, and P.L. Lanzi. On-line neuroevolution applied to the open racing car simulator. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 2622–2629, May 2009.

[8] Luigi Cardamone. On-line and off-line learning of driving tasks for the open racing car simulator (torcs) using neuroevolution. Master's thesis, Politecnico di Milano, 2008.

[9] Luigi Cardamone. On-line and Off-line Learning of Driving Tasks for The Open Racing Car Simulator (TORCS) Using Neuroevolution. Master's thesis, Politecnico di Milano — Dipartimento di Elettronica e Informazione, November 2008.

[10] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1179–1186, New York, NY, USA, 2009. ACM.

[11] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Learning drivers for torcs through imitation using supervised methods. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 148–155, Sept. 2009.

[12] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Applying cooperative coevolution to compete in the 2009 torcs endurance world championship. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –8, jul. 2010.

[13] Luigi Cardamone, Daniele Loiacono, Pier Luca Lanzi, and Alessandro Pietro Bardelli. Searching for the optimal racing line using genetic algorithms. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 388 –394, aug. 2010.

[14] Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Chi Keong Goh, Juan J. Merelo Guervós, Ferrante Neri, Mike Preuss, Julian Togelius, and Georgios N. Yannakakis, editors. *Applications of Evolutionary Computation, EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, volume 6024 of *Lecture Notes in Computer Science*. Springer, 2010.

[15] Nicholas Cole, Sushil J. Louis, and Chris Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 139–145, 2004.

[16] Fredrik A. Dahl. Honte, a go-playing program using neural nets. In *In Workshop on Machine learning in Game Playing*, pages 205–223. Nova Science Publishers, 1999.

[17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 6(2), April 2002.

[18] Marc Ebner and Thorsten Tiede. Evolving driving controllers using genetic programming. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 279–286, Sept. 2009.

[19] M.S. El-Nasr, A. Vasilakos, C. Rao, and J. Zupko. Dynamic intelligent lighting for directing visual attention in interactive 3-d scenes. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(2):145–153, June 2009.

[20] Eric Espié, Christophe Guionneau, Bernhard Wymann, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, the open racing car simulator. `http://www.torcs.org`, 2005.

[21] Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience. In Mario Giacobini et al., editor, *Applications of Evolutionary Computing*, volume 4974 of *LNCS*, pages 485–490, Napoli, Italy, 2008. Springer.

[22] Russ Frushtick. Borderlands Has 3,166,880 Different Weapons, July 2009. http://multiplayerblog.mtv.com/2009/07/28.

[23] S. A. Glantz and B. K. Slinker. *Primer of Applied Regression & Analysis of Variance*. McGraw Hill, 2001. second edition.

[24] Dunwei Gong, Jie Yuan, and Xiaoping Ma. Interactive genetic algorithms with large population size. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 1678 –1685, june 2008.

[25] J. Hagelback and S.J. Johansson. Dealing with fog of war in a real time strategy game environment. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 55–62, Dec. 2008.

[26] Jeff Hannan. Interview to jeff hannan, 2001. http://www.generation5.org/content/2001/hannan.asp.

[27] E. Hastings, R. Guha, and K.O. Stanley. Neat particles: Design, representation, and animation of particle system effects. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG 2007*, pages 154–160, 2007.

[28] Erin J. Hastings, Ratan K. Guha, , and Kenneth O. Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):245–263, 2009.

[29] Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley. Evolving content in the galactic arms race video game. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 241–248, Sept. 2009.

[30] Philip Hingston. A new design for a turing test for bots. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[31] Dennis G. Jerz. Somewhere nearby is colossal cave: Examining will crowther's original "adventure" in code and in kentucky. *Digital Humanities Quarterly*, 1(2), 2007.

[32] D B Fogel K Chellapilla. Co-evolving checkers playing programs using only win, lose, or draw. In *AeroSense99, Symposium on Applications and Science of Computational Intelligence II*, 1999.

[33] I.V. Karpov, T. D'Silva, C. Varrichio, K.O. Stanley, and R. Miikkulainen. Integration and evaluation of exploration-based learning in games. In *Proc. IEEE Symposium on Computational Intelligence and Games*, pages 39–44, 2006.

[34] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 IEEE Congress on Evolutionary Computation*, pages 995–1002. IEEE Press, 2001.

[35] Hee-Su Kim and Sung-Bae Cho. Application of interactive genetic algorithm to fashion design. *Engineering Applications of Artificial Intelligence*, 13(6):635 – 644, 2000.

[36] John Koza. *Genetic Programming.* MIT Press, 1992.

[37] John E. Laird and Jonathan Schaeffer, editors. *Procedural Level Design for Platform Games.* The AAAI Press, 2006.

[38] Pier Luca Lanzi. Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 337–344, Orlando (FL), July 1999. Morgan Kaufmann.

[39] Alessandro Lazaric. *Knowledge transfer in reinforcement learning.* PhD thesis, Politecnico di Milano, 2008.

[40] Robert Levinson and Ryan Weber. Chess neighborhoods, function combination, and reinforcement learning. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2000.

[41] Xavier Llorà, Kumara Sastry, David E. Goldberg, Abhimanyu Gupta, and Lalitha Lakshmi. Combating user fatigue in igas: partial ordering, support vector machines, and synthetic fitness. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1363–1370, New York, NY, USA, 2005. ACM.

[42] D. Loiacono, P.L. Lanzi, J. Togelius, E. Onieva, D.A. Pelta, M.V. Butz, T.D. Lonneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, and J. Quadflieg. The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131 –147, jun. 2010.

[43] D. Loiacono, J. Togelius, P.L. Lanzi, L. Kinnaird-Heether, S.M. Lucas, M. Simmerson, D. Perez, R.G. Reynolds, and Y. Saez. The wcci 2008 simulated car racing competition. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 119–126, Dec. 2008.

[44] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship 2009: Competition software manual. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 2009.

[45] Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, and Luigi Cardamone. Learning to overtake in torcs using simple reinforcement learning. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –8, jul. 2010.

[46] Daniele Loiacono, Julian Togelius, and Pier Luca Lanzi. The car racing competition homepage. Online. http://cig.dei.polimi.it/.

[47] Daniele Loiacono, Julian Togelius, and Pier Luca Lanzi. Software manual of the car racing competition. Online. http://mesh.dl.sourceforge.net/sourceforge/cig/CIG2008-Manual-V1.pdf.

[48] Simon Lucas. Evolving a neural network location evaluator to play ms. pac-man. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 203–210, 2005.

[49] Joe Marks and Vincent Hom. Automatic design of balanced board games. In Jonathan Schaeffer and Michael Mateas, editors, *AIIDE*, pages 25–30. The AAAI Press, 2007.

[50] Telmo L. T. Menezes, Tiago R. Baptista, and Ernesto J. F. Costa. Towards generation of complex game worlds. In *Proc. IEEE Symposium on Computational Intelligence and Games*, pages 224–229, 2006.

[51] N. Monmarche, G. Nocent, M. Slimane, G. Venturini, and P. Santini. Imagine: a tool for generating html style sheets with an interactive genetic algorithm based on genes frequencies. In *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*, volume 3, pages 640 –645 vol.3, 1999.

[52] Jorge Munoz, German Gutierrez, and Araceli Sanchis. Controller for torcs created by imitation. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 271–278, Sept. 2009.

[53] Jacob Kaae Olesen, Georgios Yannakakis, and John Hallam. Real-time challenge balance in an RTS game using rtNEAT. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 87–94, December 2008.

[54] E. Onieva, D. A. Pelta, J. Alonso, V. Milanes, and J. Perez. A modular parametric architecture for the torcs racing engine. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 256–262, Sept. 2009.

[55] Enrique Onieva, Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Overtaking opponents with blocking strategies using fuzzy logic. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 123 –130, aug. 2010.

[56] G.B. Parker and M. Parker. The incremental evolution of attack agents in xpilot. pages 969–975, 0-0 2006.

[57] G.B. Parker and M. Parker. Evolving parameters for xpilot combat agents. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG 2007*, pages 238–243, 2007.

[58] G.B. Parker, M. Parker, and S.D. Johnson. Evolving autonomous agent control in the xpilot environment. volume 3, pages 2416–2421 Vol. 3, Sept. 2005.

[59] M. Parker and G.B. Parker. The evolution of multi-layer neural networks for the control of xpilot agents. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG 2007*, pages 232–237, 2007.

[60] C. Pedersen, J. Togelius, and G.N. Yannakakis. Modeling player experience for content creation. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(1):54 –67, mar. 2010.

[61] Chris Pedersen, Julian Togelius, and Georgios N. Yannakakis. Modeling Player Experience for Content Creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):54–67, 2010.

[62] D. Perez, Y. Saez, G. Recio, and P. Isasi. Evolving a rule system controller for automatic driving in a car racing competition. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 336–342, Dec. 2008.

[63] Jordan Pollack, Alan D. Blair, and Mark Land. Coevolution of a backgammon player. In *Proceedings Artificial Life V*, pages 92–98. MIT Press, 1996.

[64] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels. Evolution of human-competitive agents in modern computer games. pages 777–784, 0-0 2006.

[65] Steffen Priesterjahn. Imitation-based evolution of artificial players in modern computer games. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1429–1430, New York, NY, USA, 2008. ACM.

[66] Steffen Priesterjahn, Oliver Kramer, Alexander Weimer, and Andreas Goebels. Evolution of reactive rules in multi player computer games based on imitation. In Lipo Wang, Ke Chen, and Yew-Soon Ong, editors, *ICNC (2)*, volume 3611 of *Lecture Notes in Computer Science*, pages 744–755. Springer, 2005.

[67] J. Quadflieg, M. Preuss, O. Kramer, and G. Rudolph. Learning the track and planning ahead in a car racing controller. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 395 –402, aug. 2010.

[68] John Reeder, Roberto Miguez, Jessica Sparks, Michael Georgiopoulos, Georgios Anagnostopoulos, and Reeder. Interactively evolved modular neural networks for game agent control. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 167–174, Dec. 2008.

[69] R.G. Reynolds and M. Ali. Computing with the social fabric: The evolution of social intelligence within a cultural framework. *Computational Intelligence Magazine, IEEE*, 3(1):18–30, February 2008.

[70] Norman Richards, David E. Moriarty, Paul McQuesten, and Risto Miikkulainen. Evolving neural networks to play go. In Thomas Bäck, editor, *ICGA*, pages 768–775. Morgan Kaufmann, 1997.

[71] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):210–229, 1959.

[72] Kumara Sastry. Single and multiobjective genetic algorithm toolbox in C++. Technical report, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, IlliGAL Report No. 2007016, 2007.

[73] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53:53–2, 1992.

[74] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing 6*, pages 817–824. Morgan Kaufmann, San Francisco, 1994.

[75] Jimmy Secretan and Nicholas Beato. Picbreeder: evolving pictures collaboratively online. In *CHI*, pages 1759–1768, 2008.

[76] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards Automatic Personalized Content Generation for Platform Games. 2010.

[77] Matt Simmerson. Neat4j homepage. Online, 2006. 2006. [Online]. Available: http://neat4j.sourceforge.net.

[78] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In Chio et al. [14], pages 131–140.

[79] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[80] Kenneth O. Stanley, Ryan Cornelius, and Risto Miikkulainen. Real-time learning in the nero video game. In R. Michael Young and John E. Laird, editors, *AIIDE*, pages 159–160. AAAI Press, 2005.

[81] Kenneth O. Stanley, Nate Kohl, Rini Sherony, and Risto Miikkulainen. Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, 2005.

[82] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural network through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[83] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[84] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[85] H. Takagi. Interactive evolutionary computation: fusion of the capabilities of ec optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275 –1296, September 2001.

[86] Matthew E. Taylor and Peter Stone. Behavior transfer for value-function-based reinforcement learning. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, New York, NY, July 2005. ACM Press.

[87] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[88] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Value functions for RL-based behavior transfer: A comparative study. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, July 2005.

[89] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 156–163, May 2007.

[90] G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[91] G. Tesauro and T.J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39(3):357–390, 1989.

[92] Sebastian Thrun. Learning to play the game of chess. In *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, 1995.

[93] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing Systems*, pages 640–646. The MIT Press, 1996.

[94] J. Togelius, R. De Nardi, and S.M. Lucas. Towards automatic personalised content creation for racing games. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG 2007*, pages 252–259, 2007.

[95] J. Togelius and S.M. Lucas. Evolving robust and specialized car racing skills. pages 1187–1194, 0-0 2006.

[96] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 111–118, Dec. 2008.

[97] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making racing fun through player modeling and track evolution. In *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.

[98] Julian Togelius and Simon M. Lucas. Evolving controllers for simulated car racing. In *Proceedings of the Congress on Evolutionary Computation*, 2005.

[99] Julian Togelius and Simon M. Lucas. Arms races and car races. In *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[100] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making racing fun through player modeling and track evolution. In *Proceedings of the SAB Workshop on Adaptive Approaches to Optimizing Player Satisfaction*, 2006.

[101] Julian Togelius, Mike Preuss, Nicola Beume, Johan Hagelbaeck Simon Wessing, and Georgios N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 265–262, Copenhagen, Denmark, 18-21 August 2010., 2010. IEEE.

[102] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In *Proceedings of EvoApplications*, volume 6024. Springer LNCS, 2010.

[103] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In Chio et al. [14], pages 141–150.

[104] V/A. Cube 2: Sauerbraten. http://sauerbraten.org/.

[105] N. van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber. Robust player imitation using multiobjective evolution. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 652–659, May 2009.

[106] Joe Venzon. Vdrift: a cross-platform, open source driving simulation. `http://vdrift.net/`.

[107] P. Walsh and P. Gade. Terrain generation using an interactive genetic algorithm. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –7, july 2010.

[108] Shimon Whiteson and Peter Stone. On-line evolutionary computation for reinforcement learning in stochastic domains. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1577–1584, New York, NY, USA, 2006. ACM.

[109] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 15(1):33–50, 2007.

[110] Mark Wittkamp, Luigi Barone, and Philip Hingston. Using neat for continuous adaptation and teamwork formation in pacman. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 234–242, Dec. 2008.

[111] Krzysztof Wloch and Peter J. Bentley. Optimising the performance of a formula one car using a genetic algorithm. In *Proceedings of Eighth International Conference on Parallel Problem Solving From Nature*, pages 702–711, 2004.

[112] Bin Xu, Shangfei Wang, and Xian Li. An emotional harmony generation system. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –7, july 2010.

[113] Georgios N. Yannakakis and John Hallam. Towards capturing and enhancing entertainment in computer games. In *Proceedings of the Hellenic Conference on Artificial Intelligence*, pages 432–442, 2006.

[114] Georgios N. Yannakakis and John Hallam. Modeling and augmenting game entertainment through challenge and curiosity. *International Journal on Artificial Intelligence Tools*, 16(6):981–999, 2007.

[115] Georgios N. Yannakakis and Julian Togelius. Experience-driven Procedural Content Generation. *IEEE Transactions on Affective Computing*, 2011.

[116] Maliang Zheng and Daniel Kudenko. Automated event recognition for football commentary generation. In *Proc. AISB'09 Symposium: AI & GAMES*, 2009.