Adaptive Controllers for Real-Time Games

by

Diego Perez

A thesis submitted for the degree of Doctor of Philosophy

School of Computer Science and Electronic Engineering

University of Essex

January, 2015

Dedicated to all those that were, are and will be prosecuted for putting evidence and reason before religion and superstition.

Acknowledgements

Having only one name on the front page of this thesis seems ridiculous to me: the number of people that have contributed in a minor or major way, from the most banal to the most deep of conversations, is overwhelming. This page acknowledges all of them, who have shared in some way, at some point, part of these years with me. It is not possible to list all their names on this page, but here there are some: thanks to Philipp, Spyros, David, Aisha, Paul and Samuel, from my research group; and to Georgia, Ays, Ana and Davide, from other University research groups.

In particular, special mention needs to be made of my dearest friends Javi and Pili, for being my hosts in Colchester and at the University, and a constant presence in my life. Thanks to my supervisor, Professor Simon M. Lucas, who has fought innumerable mini-battles to teach, support and fund me. Thanks also to the Game Intelligence Group at the University of Essex for those drinks and useful conversations about research and unemployment. Important mention also should be made of Michelle Hallworth, who has corrected my English in all my papers and this very thesis; and of the EPSRC, for granting the funding that allowed me to carry out this work.

Last, but not least, I need to thank Erika, who has shared this journey with me. I couldn't have done it without your support and comprehension during all those many weekends of working at home. And also thanks to her family, and especially to María and Jesús. And finally, to my own family, parents and sister, who have always been with me, even from a distance. Your support encourages me to carry on every day.

Thank you!

Agradecimientos

Que sólo aparezca un nombre en la portada de esta tesis me parece un poco ridículo: la cantidad de gente que ha contribuido en una menor o mayor forma, desde las más banales hasta las más importantes de las conversaciones, es abrumadora. Esta página sirve de agradecimiento para todas ellas, que han compartido, en algún momento, parte de estos tres años conmigo. No es posible nombrarlos a todos en esta página, pero aquí van algunos: Philipp, Spyros, David, Aisha, Paul y Samuel, de mi grupo de investigación; y Georgia, Ays, Ana y Davide, de otros grupos de la Universidad.

Particularmente, una mención especial merecen mis muy queridos amigos Javi y Pili, por acogerme en Colchester y en la Universidad, y por ser una presencia constante en mi vida. Gracias a mi supervisor, Professor Simon M. Lucas, que ha librado innumerables mini-batallas para enseñarme, apoyarme y conseguir la financiación necesaria. Gracias también al Game Intelligence Group de la Universidad, por todas esas pintas e interesantes conversaciones sobre investigación y desempleo. También quiero dar las gracias a Michelle Hallworth, quien ha corregido mi inglés en numerosos artículos, incluyendo esta tesis; y al EPSRC, por conceder la financiación que me ha permitido llevar a cabo este trabajo.

Finalmente, pero no por ello menos importante, necesito darle las gracias a Erika, que ha compartido este viaje conmigo. No podría haberlo conseguido sin tu apoyo y comprensión, sobre todo durante todos esos fines de semana trabajando en casa. También gracias a su familia, especialmente a María y a Jesús. Y finalmente, a mi propia familia, mis padres y mi hermana, que siempre han estado conmigo, aún en la distancia. Vuestro apoyo me anima a seguir adelante cada día.

¡Gracias!

Abstract

This thesis analyzes the performance of Artificial Intelligence techniques to create adaptive controllers that play real-time games. Research in games has been a prolific field of study in the last decade, and multiple approaches have been suggested in the literature. Real-time games are a subset of the games used as testbeds in the field, and they establish a constraint that make them interesting: a small time budget, mere milliseconds, for the algorithm to decide the next action to take. With this limited time, the algorithm is unable to explore a significant part of the search space, leaving terminal states of the game far from the simulation horizon.

Adaptive controllers for real-time games are proposed in this thesis that mainly employ two algorithms: Monte Carlo Tree Search and Evolutionary Algorithms. These approaches incorporate solutions to the problems posed by real-time scenarios, and they are thoroughly tested in several games. This thesis also proposes the use of adaptive controllers as a tool to automatically generate content for games, and introduces an initial study into employing adaptive controllers for General Video Game Playing, an area of research where game specific heuristics are not available.

The experimental work conducted in this research examines and compares different approaches, showing the benefits of the proposed algorithms and modifications. Among the main contributions of this thesis, it is worth highlighting the coarsening of the action space via macro-actions, the use of evolutionary algorithms with a rolling horizon function for real-time control, the definition of a novel multi-objective algorithm for real-time games as well as knowledge discovery and re-use of past experiences without domain specific information.

Contents

Part I

Introduction and Background

"In all your travels, have you ever seen a star go supernova? . . . "
- Brother Cavil, Battlestar Galactica

Chapter 1

Introduction

This chapter outlines the motivation for this thesis and summarizes its main contributions. It also defines concepts used frequently throughout the document, presents its organization and lists those papers published by the author of this thesis that contribute to this research.

1.1 Introduction

Artificial Intelligence (AI) research in games has become an important field of study in recent years, and within the field there are a diverse range of interesting challenges and benchmarks. Games allow the creation of multiple and varied simulated environments that can mimic diverse situations in real life, without the necessary cost in time and money that performing experiments in physical scenarios could entail. Furthermore, they are benchmarks where new and old algorithms can be compared in a relatively reduced amount of time, allowing replicability and standarization. Games studied in the field vary greatly in complexity, and sometimes several problems have to be tackled to make progress within a short decision time budget.

Traditionally, research in Game AI has focused on 2-player perfect information games where the time allowed to make a move is measured in seconds or minutes. This is the case with classical board games such as Chess, Checkers or Go. Lately, research has started to focus on more complex domains with partial observability, where the whole state of the game is unobserved by the players. Examples of this are most card games, or strategy games with hidden units. Another set of games that entails significant complexity is that of stochastic games, where random events complicate the task of planning future actions and their consequences. Examples are games like Backgammon, Monopoly or Scrabble.

The research conducted for this thesis is focused on another group of games with significant complexity: real-time or video games. Whilst turn-based games usually allow a "thinking" time of at least one second for the agent to decide the next move to make, real-time games require the players to pick actions at a much higher rate, with a decision time typically measured in tens of milliseconds. This constraint requires the search algorithm to converge very fast without much overhead. These games are usually very long and/or complex, and players must solve a wide range of problems, often in very little time, to make progress. Additionally, the real-time component allows for an asynchronous interaction between the player and the other entities: the game progresses even if the player does not take any action.

The term *adaptive controller* is coined in this thesis to refer to the agents that play this type of games. They are *controllers* because they are the entities in control of the action decision mechanism during the game. Furthermore, they are said to be *adaptive* due to the fact that they re-plan the next moves to make at every game cycle (or every few game cycles), being able to overcome unexpected situations that did not occur in previous plans.

Additionally, the complexity of performing research in real-time games is not only the hazard of selecting the next action to make within a small time budget, but also how to act in relation to the whole game. Most of these games are open-ended, which essentially means that players can not devise, for the vast majority of their action decisions, any action or sequence of actions that will lead the agent to win the game. With this in mind, it sounds obvious that a long-term plan of actions or some clever heuristics are required. This thesis analyzes these ideas in the chapters that follow.

It is worthwhile investigating how search methods are able to explore the search space of those games characterized by limited decision time. Concepts like pruning heuristics, coarsening of the action space, and games seen from a multi-objective perspective are explored during the following chapters. These search techniques are very important, as relying on domain-based, game-specific heuristics is not always an available option. General Video Game Playing is another interesting domain within the Game AI field, in particular the goal of creating an agent that is able to play in many different games, even in those that are unknown to the programmer. Faced with such limited information, knowledge discovery becomes an essential requirement in the algorithm, over and above simply acting appropriately. Chapter 9 explores the performance of adaptive controllers within this environment.

These kinds of scenarios, and the algorithms employed, can be tested significantly more easily in games than in real life, and represent an interesting first step towards General Artificial Intelligence.

The idea of adaptive controllers does not have the sole objective of creating intelligent agents that solve puzzles or win games, as they can also help with other aspects of their development. The gaming industry is a powerful market that clearly attracts strong interest: a recent report of the Entertainment Software Association establishes that, in 2011 alone, consumers in the US spent 24.75 billion dollars on video games and hardware. Furthermore, the games industry in the US showed a growth rate of 10.6% for the period 2005-2009 (Siwek, 2010), while the growth rate for the US economy as a whole was 1.4% for the same period. Adaptive controllers for real-time games can also help in other aspects of game development, such as testing, providing an appropriate level of difficulty or even generating content. Chapter 6 of this thesis shows a practical case where an adaptive controller is used to automatically generate levels for a particular game.

Finally, the research conducted in this thesis is closely related to video-game competitions. These kinds of competitions provide an ideal and trustworthy test-bed for practitioners of Game AI, and are especially well suited for students and newcomers to the field. In these contests, different agents are compared under exactly the same circumstances (games and hardware), organized by an independent third party. Most of the different games and benchmarks presented in this document have featured in video-game competitions, organized by the author of this thesis, in collaboration with the Game Intelligence Group at the University of Essex and other external researchers. A brief description of these competitions is given in Appendix A of this document.

1.2 Contributions

The main scientific contributions of this thesis are the ones described in Part II, including both experimental work and the proposal of new algorithms. Generally, all contributions can be seen as different mechanisms to overcome the difficulties imposed by the real-time aspect of the games employed, as mentioned in the previous section. These contributions can be summarized as follows:

- The use of game-dependent heuristics to guide action selection in Monte Carlo Tree Search (MCTS), by means of pruning branches in the tree and biasing Monte Carlo simulations. The

heuristics determine those actions that do not lead to a victory in the game, and prevent the algorithm from selecting them. The experimental work described in Chapter 4 shows that including these heuristics improves the performance of the algorithm considerably.

- The balance between long and short term planning to tackle the problem posed by the open-ended feature of real-time games. Chapter 5 shows how different long-term plans affect the performance of MCTS, and the importance of devising long-term plans that are intrinsically related to short-term action selection. Additionally, Chapter 6 proposes a novel methodology to create levels for games using adaptive controllers, exploiting the balance between these two types of planning. Agents that make a bigger effort to create better long-term plans are generally expected to obtain better results. Covariance Matrix Adaptation - Evolutionary Strategy is employed to evolve maps that fulfil this requirement, using adaptive controllers to evaluate individuals.

- The concept of macro-actions as a solution to the problem caused by the limited time budget of real-time games: simulated moves cannot reach very far into the future, and the consequences of the actions taken are unclear. The experimental work shows that it is possible to perform a more effective search by applying a group or sequence of actions as a single move. Macro-actions are introduced in Chapter 7 as a way to coarsen the action space and therefore improve the search capabilities of the algorithm.

- The proposal of an evolutionary algorithm that can be used online as an adaptive controller. Chapter 7 introduces a rolling horizon evolutionary algorithm for real-time games, evolving a string of actions and evaluating the state of the game at the end of the sequence. The action to apply is the first move in the best sequence found when the time budget is over. Experiments show that its performance is on a par with state-of-the-art algorithms like MCTS.

- A new algorithm is proposed to consider games as multi-objective problems. Chapter 8 describes multi-objective MCTS, a novel algorithm that explores this concept by constructing Pareto fronts on each node in the tree. The proposed algorithm aims to approximate the optimal Pareto front in real-time games. Additionally, it shows that it is possible to tune the search to follow specific points in the discovered front. Experimental work shows that this algorithm provides better results than weighted-sum approaches in some of the games tested.

- An algorithm is proposed in Chapter 9 to deal with situations where game-specific knowledge is not available. MCTS simulations are biased to those portions of the search space that allow the algorithm to discover features and re-use past experiences, improving its performance in comparison to the vanilla version of the algorithm. This algorithm also employs an interesting approach that integrates evolution with MCTS roll-outs, in an attempt to maximize the information gained from the simulations.

1.3 Definitions

This section defines concepts that are used throughout the thesis. As these terms are sometimes used differently by distinct authors in the literature, it is important to declare here how are they used in this document.

Definition 1.1 (Game)**.** A game is a "system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome", as defined by (Salen and Zimmerman, 2003). More formally, a game can also be defined as a Markov Decision Process (MDP, see Definition 1.2). Additionally, in this thesis, games are referred to as each one of the benchmark environments chosen for the agents to play in. The games employed for the research detailed in this thesis are described in Chapter 3.

Definition 1.2 (Markov Decision Process (MDP))**.** A Markov Decision Process models a problem of sequential decisions by defining a tuple $< S, A, P, R, \gamma >$, where:

- S: finite set of states.
- A: finite set of actions.
- $P(s, a, s')$: transition matrix that determines the probability of reaching state *s'* if action a is applied from state *s*.
- R: reward function.
- γ: discount factor $\gamma \in [0, 1]$.

Decisions are modelled as sequences of *(state, action)* pairs, where the next state is given by P. The aim is to find the policy (π, see Definition 1.3) that provides the highest expected reward. For more information about MDPs, the reader is referred to (Russell and Norvig, 2003, Ch.17).

Definition 1.3 (Policy)**.** A policy π is a mapping from states to actions (Russell and Norvig, 2003; Szepesvári, 2010).

Definition 1.4 (Regret)**.** Loss due to the fact that the optimal policy is not followed all the time (Auer, Cesa-Bianchi, and Fischer, 2002).

Definition 1.5 (Tree Policy)**.** Policy used to select actions in a Monte Carlo Tree Search iteration while navigating through nodes that exist in the tree.

Definition 1.6 (Default Policy)**.** Policy used to select actions in a Monte Carlo Tree Search iteration, during the Monte Carlo simulation phase. States encountered during this step are not stored in the tree.

Definition 1.7 (Simulation Depth)**.** Maximum length of the sequence of actions taken from the root on each iteration of the Monte Carlo Tree Search algorithm.

Definition 1.8 (Play-Out)**.** In a Monte Carlo Tree Search iteration, a play-out is a sequence of moves executed from the root of the tree till the end of the game or an established simulation depth. It therefore includes both the choices made during the Tree Policy and the Default Policy.

Definition 1.9 (Roll-out)**.** Sequence of actions picked in a Monte Carlo Tree Search iteration from the expanded node till the end of the game or an established simulation depth. It therefore includes only the choices made during the default policy.

Definition 1.10 (Value Function or Score Function)**.** Mechanism that determines the reward (or rewards) of a given state, normally calculated on specific features extracted from it.

Definition 1.11 (Game Cycle, Game Step or Time Step)**.** Each one of the turns a game is divided into. On each game cycle, the agent is supposed to move once, by supplying which action must be executed after an established decision time budget.

Definition 1.12 (Agent, Controller or Player)**.** A participant in a game, that takes actions every game cycle. In this thesis, the agent/controller/player is the one that makes moves guided by a particular algorithm, subject of study, within the game environment.

Definition 1.13 (Move, Action or Ply)**.** Each one of the possibilities for action available to an agent at each game step, corresponding to the branching factor at that particular level of the tree.

Definition 1.14 (End-Game State or Terminal State)**.** The game-state reached when the game is finished, stating if the player won or lost the match, and probably the final score.

Definition 1.15 (Real-Time or Video Game)**.** Games where the time budget allowed for a controller to decide the next move is significantly reduced, usually less than 50 milliseconds.

Definition 1.16 (Open-Ended Game)**.** A game where its end-game states are usually farther ahead in the future than the states the controller can visit during its simulations. In other words, most lines of play that are simulated from the current state will never reach an end-game state. It is worthwhile highlighting that many traditional games (such as Chess) share this characteristic with real-time games, although the limited time budget of the latter accentuates the problems this imposes.

Definition 1.17 (Level, Maze or Map)**.** Each one of the different scenarios of a game played by agents. Rules and game-play elements are shared across all levels of a game, only the initial positions of the entities vary from one another.

Definition 1.18 (Match)**.** An instance of a level played by an agent, including its result.

Definition 1.19 (Multivariate Normal Distribution (MND))**.** A random vector of real numbers $X = (x_0, x_1, \dots x_n), x_i \in \mathcal{R}$, is said to be n-variate normally distributed iff, for every n-vector $W = (w_0, w_1, \dots w_n)$, $w_i \in \mathcal{R}$, the (one-dimensional) random variable $Y = W'X$ is normal. $Y = W'X$ has a (univariate) normal distribution, where a univariate normal distribution with zero variance is a point on its mean (Gut, 1995).

Additionally, when numeric results are given as averages, the standard error of the measure is provided between parenthesis (i.e 10.00 (0.25)). To finish, it is important to highlight that all experiments from this research, unless otherwise stated, have been run on an Intel Core i5 PC, with 2.90GHz and 4GB of memory.

1.4 Organization

This thesis is divided into two main parts. Part I provides an introductory background for the thesis. The current chapter outlines the motivation for the thesis, details its contributions to the field, defines common terms used throughout the document and indicates the papers that constitute the backbone of the thesis. Chapter 2 provides background on the techniques used throughout this research, including the relevant literature review. Chapter 3 concludes Part I with the definition of all the games and frameworks used as benchmarks for the experiments described later in the document.

Part II is centred on the experimental work conducted. Chapter 4 explores the use of heuristics and pruning in Monte Carlo Tree Search, both in the default and in the tree policy. Chapter 5 tackles the problem of long-term versus short-term planning in real-time games. Later, Chapter 6 showcases the use of an adaptive controller to automatically generate content for a real-time game. Chapter 7 introduces the concept of macro-actions and rolling horizon evolutionary algorithms for adaptive controllers in real-time games. Later, Chapter 8 proposes a novel algorithm to tackle real-time games as Multi-Objective optimization problems. The last experimental study, in Chapter 9, analyzes the problems of and proposes solutions to controllers in general video game playing. Part II finishes with Chapter 10, where final conclusions are drawn and future work is proposed.

This document ends with Appendix A, that summarizes the video-game competitions organized by the author of this thesis, closely related to the conducted research.

1.5 List of papers

The following list of papers constitutes the main contributions to this thesis. They have all either been published or accepted for publication. The specific contributions of each paper are listed in bold font.

1. Diego Perez, Philipp Rohlfshagen and Simon Lucas. Monte-Carlo Tree Search for the Physical Travelling Salesman Problem. *Proceedings of EvoApplications*, pp. 255–264, 2012.

   Algorithm, heuristics, experiments and writing. The second and third authors helped with discussions and the writing of the paper, and the third author provided the game implementation.

2. Diego Perez, Philipp Rohlfshagen and Simon Lucas. Monte Carlo Tree Search: Long Term versus Short Term Planning. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pp. 219–226, 2012.

   Algorithm, experiments and writing. The second and third authors helped with discussions and the writing of the paper. This paper received the best paper award of the conference.

3. Diego Perez, Spyridon Samothrakis and Simon Lucas. Online and Offline Learning in Multi-Objective Monte Carlo Tree Search. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pp. 121–128, 2013.

   Novel algorithm, implementation of the games, experiments and writing. The second and third authors helped with discussions and the writing of the paper.

4. Diego Perez, Spyridon Samothrakis, Simon Lucas and Philipp Rohlfshagen. Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games. *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 351–358, 2013.

   Algorithm, experiments and writing. The other authors helped with discussions and the writing of the paper.

5. Diego Perez, Julian Togelius, Spyridon Samothrakis, Philipp Rohlfshagen and Simon Lucas. Automated Map Generation for the Physical Travelling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, pp. 1–14, 2013, doi: 10.1109/TEVC.2013.2281508.

   Controller, experiments and writing of the paper. The second author helped with the background on Procedural Content Generation, and the third author implemented CMA-ES to create maps. All authors helped with the writing of the paper.

6. Diego Perez, Edward J. Powley, Daniel Whitehouse, Philipp Rohlfshagen, Spyridon Samothrakis, Peter I. Cowling and Simon Lucas. Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6:1, pp. 31–45, 2013.

   Implementation of the game, writing of the paper and experiments. The second and

third authors implemented the algorithms tested, and all authors helped with the writing of the paper.

7. Diego Perez, Philipp Rohlfshagen and Simon Lucas. The Physical Travelling Salesman Problem: WCCI 2012 Competition. *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1–8, 2012.

   Implementation of the framework, server software and sample controllers, and writing. The second and third authors helped with discussions and the writing of the paper.

8. Diego Perez, Edward Powley, Daniel Whitehouse, Spyridon Samothrakis, Simon Lucas and Peter I. Cowling. The 2013 Multi-Objective Physical Travelling Salesman Problem Competition. *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. to appear, 2014.

   Implementation of the framework, server software and sample controllers, and writing. The second and third authors implemented and wrote the description of their controller, winner of the 2013 MO-PTSP Competition. All authors helped with the writing of the paper.

9. Diego Perez, Spyridon Samothrakis and Simon Lucas. Knowledge-based Fast Evolutionary MCTS for General Video Game Playing. *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, 2014, pp 68–75.

   Implementation of the framework, novel algorithm, experiments and writing. The second author helped with the development of the server software for the experiments. The second and third authors helped with discussions and the writing of the paper.

The following papers have been accepted for publication at the time of the writing of this thesis:

10. Diego Perez, Sanaz Mostaghim, Spyridon Samothrakis, Simon Lucas. Multi-Objective Monte Carlo Tree Search for Real-Time Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

    Novel algorithm, implementation of the games, experiments and writing. The second

author helped with the Multi-Objective background. The second and third authors helped with the writing of the paper.

Finally, the following papers were published during the writing of the thesis, although they do not contribute directly to it:

11. Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4:1, pp. 1–43, 2012.

12. Spyridon Samothrakis, Diego Perez, Philipp Rohlfshagen and Simon Lucas. Predicting Dominance Rankings for Score-based Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

13. Spyridon Samothrakis, Samuel Roberts, Diego Perez and Simon Lucas. Rolling Horizon methods for Games with Continuous States and Actions. *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, 2014, pp 224–231.

14. Simon Lucas, Spyridon Samothrakis and Diego Perez. Fast Evolutionary Adaptation for Monte Carlo Tree Search. *Proceedings of EvoGames*, pp. to appear, 2014.

15. Simon Lucas, Philipp Rohlfshagen and Diego Perez. Towards More Intelligent Adaptive Video Game Agents: A Computational Intelligence Perspective. *Proceedings of ACM International Conference on Computing Frontiers*, pp. 293–298, 2012.

"No? Well, I have. I saw a star explode and send out the building blocks of the Universe. Other stars, other planets and eventually other life. . . ."
- Brother Cavil, Battlestar Galactica

Chapter 2

Background

This chapter gives the background on the algorithms on which this thesis is based: Monte Carlo Tree Search, Multi-Objective Optimization, Covariance Matrix Adaptation, Procedural Content Generation and General Video-Game Playing.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS), a tree search algorithm, was first proposed in 2006 (Chaslot, Bakkes, Szita, and Spronck, 2006; Coulom, 2006; Kocsis and Szepesvári, 2006). It was originally applied to board games and is closely associated with Computer Go, where it has led to a breakthrough in performance (Lee, Wang, Chaslot, Hoock, Rimmel, Teytaud, Tsai, Hsu, and Hong, 2009). Computer Go is played in a square grid board, of 19×19 in the original game, and 9×9 in its reduced version. The game is played in turns, and the objective is to surround the opponent's stones by placing stones in any available position on the board. Due to the very large branching factor of the game and the absence of clear heuristics to tackle it, Go became object of study for many researchers. MCTS was the first algorithm able to reach professional level play in the reduced board size version (Gelly and Silver, 2011). MCTS rapidly became popular due to its significant success in this game, where traditional approaches had been failing to outplay experienced human players.

Since then, MCTS has been applied to a wide range of games, including games of uncertain information, general game playing, single-player games and real-time games. An extensive survey that includes a description of MCTS, a complete collection of MCTS variants, and multiple applications of this algorithm to games and other domains, can be found at (Browne, Powley, Whitehouse, Lucas,

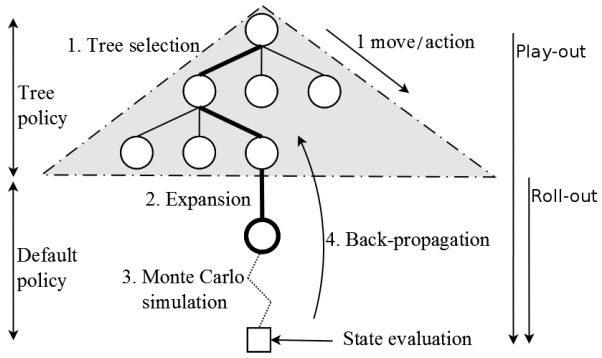


Figure 2.1: *MCTS algorithm steps.*

Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, and Colton, 2012).

MCTS is a tree search algorithm that incrementally builds an asymmetric tree in memory by adding a single node at a time, estimating its game-theoretic value by using self-play from the state of the node to the end of the game. Each node in the tree maintains statistics that indicate how often a move is played from a given state ($N(s, a)$), how many times each move is played from there ($N(s)$) and the empirical average ($Q(s, a)$) of the rewards obtained when choosing action a at state s. The tree is built iteratively by simulating actions in the game, making move choices based on these statistics.

Each iteration of MCTS can be divided into several steps (Gelly, Wang, Munos, and Teytaud, 2006): *Tree selection*, *Expansion*, *Monte Carlo simulation* and *Back-propagation* (all summarized in Figure 2.1). When the algorithm starts, the tree is composed only of the root node, which is a representation of the current state of the game. During the *selection* step, the tree is navigated from the root until a maximum depth or the end of the game has been reached.

If, during the *selection* step, actions are picked uniformly at random, it is expected that each action is selected one n-th of the time (where n is the number of different actions that can be selected from a given state), producing an evenly distributed search. This approach, however, is limited by the fact that one or more actions could be better than others, as the scores obtained by the evaluations at the end of the simulations may reflect. To overcome this problem, a *multi-armed bandit* selection policy is employed. Multi-armed bandits is a policy taken from probability theory in which each one of the multiple slot machines produces consecutive rewards $r_t : r_1, r_2, \dots$ (for time $t = 1, 2, \dots$) driven from an unknown probability distribution. The objective is to minimize the regret, i.e. the losses, of not choosing the optimal arm. Multi-armed bandit policies select actions in this problem by balancing the

exploration of available actions (pulling the arms) and the exploitation of those that provide better rewards (optimization in the face of uncertainty).

Auer et al. (Auer et al., 2002) proposed the *Upper Confidence Bound* (UCB1, see Equation 2.1) policy for bandit selection. Then, Kocsis and Szepesvári applied UCB1 as tree policy within MCTS, a version of the algorithm known as UCT − Upper Confidence bounds for Trees (Kocsis and Szepesvári, 2006).

$$a^* = \text{argmax}_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{2.1}$$

The objective is to find an action a that maximizes the value given by the UCB1 equation. $Q(s,a)$ is the *exploitation* term, while the second term (weighted by the constant C) is the *exploration* term. Note that, if the balancing weight $C = 0$, UCB1 follows greedily the action that has provided the best average outcome so far. The exploration term relates to how many times each action a has been selected from the given state s, and the amount of selections taken from the current state ($N(s,a)$ and $N(s)$, respectively). When action a is chosen from state s, the values of $N(s,a)$ and $N(s)$ increase. The effect of this is that the exploration term for other actions different than a increases, allowing a more diverse exploration of the different available actions in future iterations of the algorithm.

The value of C balances between these two terms. If the rewards $Q(s,a)$ are normalized in the range $[0,1]$, a commonly used value for C in single-player games is $\sqrt{2}$. The value of C is application dependant, and it may vary from game to game. It is worth noting that MCTS, when combined with UCB1, reaches asymptotically logarithmic regret on each node of the tree (Coquelin and Munos, 2007).

It is also important to highlight that, when $N(s,a) = 0$ for any action, that action must be chosen. In other words, it is not possible to apply UCB1 if all actions from a state have not been picked at least once. Therefore, if during the *tree selection* phase a node has fewer children than the available number of actions from a given position, a new node is added as a child of the current one (*expansion* phase) and the *simulation* step starts. At this point, MCTS executes a Monte Carlo simulation (or roll-out; *default policy*) from the expanded node. This is performed by choosing random (either uniformly random, or biased) actions until an end-game state or a pre-defined depth is reached, where the state of the game is evaluated.

Finally, during the *back-propagation* step, the statistics $N(s)$, $N(s,a)$ and $Q(s,a)$ are updated for

Algorithm 1 General MCTS approach.

```
function MctsSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TreePolicy(v_0)
        Δ ← DefaultPolicy(s(v_l))          ▷ s(v_l): State in node v_l, Δ: reward for state s.
        Backup(v_l, Δ)
    return a(BestChild(v_0))

function TreePolicy(v)
    while v is nonterminal do
        if v not fully expanded then
            return Expand(v)
        else
            v ← UCB1(v)                    ▷ Eq. 2.1.
    return v

function Expand(v)
    choose a ∈ untried actions from A(s(v))   ▷ A(s(v)): Available actions from state s(v).
    add a new child v' to v
        with s(v') = f(s(v), a)              ▷ f(s(v), a): State reached from s(v) after applying a.
    return v'

function DefaultPolicy(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s

function Backup(v, Δ)
    while v is not null do
        N(s(v)) ← N(s(v)) + 1
        N(s(v), a(v)) ← N(s(v), a(v)) + 1   ▷ a(v): Last action applied from state s(v).
        Q(s(v), a(v)) ← Q(s(v), a(v)) + Δ
        v ← parent of v
```

each node visited, using the reward obtained in the evaluation of the state. These steps are executed in a loop until a termination criteria is met (such as number of iterations, or when the time budget is consumed).

Once all iterations have been performed, MCTS returns the next action the agent must take, usually according to the statistics stored in the root node. Examples of these policies include taking the action chosen more often (a for the highest $N(s, a)$), the one that provides a highest average reward ($Q(s, a)$), or simply to apply Equation 2.1 at the root node. The pseudocode of MCTS is shown in Algorithm 1.

MCTS is considered to be an *anytime* algorithm (Gelly et al., 2006): during the *Back-propagation* step, MCTS updates the outcome of each playout immediately. This guarantees that all values are always updated on each iteration of the algorithm. Hence, MCTS is able to provide a valid next move

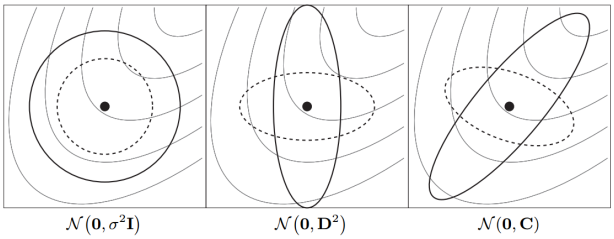


Figure 2.2: *Distributions with $m = 0$ and different C, from (Hansen, 2006)*

to choose at any moment in time (Browne et al., 2012). This is true independently from how many iterations the algorithm is able to make (although, in general, more iterations usually produce better results). This differs from other algorithms (such as A* in single player games, and standard Min-Max for two player games) that normally provide the next play only after they have finished. This makes MCTS a suitable candidate for real-time domains, where the decision time budget is limited, affecting the number of iterations that can be performed.

2.2 Covariance Matrix Adaptation - Evolutionary Strategy

The Covariance Matrix Adaptation - Evolutionary Strategy (CMA-ES) is an evolutionary algorithm designed for continuous domains specially suited for non-linear and non-convex optimization problems (Hansen, 2006). In general, this algorithm is applied to those problems that are not constrained and are made of up to 100 variables.

CMA-ES creates a population of individuals by sampling from a multivariate normal distribution $\mathcal{N}(m, C)$ (see Definition 1.19). The peak corresponds to the distribution mean, $m \in \mathcal{R}^n$, which also determines the translation of the distribution. $\mathcal{N}$ is uniquely defined by m and its covariance matrix $C \in \mathcal{R}^{n \times n}$, positive definite and symmetric. C determines the shape of the distribution and its graphical interpretation: it defines an iso-density ellipsoid $\{x_i \in \mathcal{R} | (x - m)^T C^{-1} (x - m) = 1\}$. Figure 2.2 shows the representations for MND with $m = 0$ and different covariance matrices.

The covariance matrix is isotropic if $C = \sigma^2 I$, with I being the identity matrix (on the left of Figure 2.2). D represents a diagonal matrix whereas C stands for a positive definite full covariance matrix. The contour lines define a potential objective function. The goal of CMA-ES is to adapt the shape of the distribution to the contour lines of the objective function to be minimized. In the case of Figure 2.2, the distribution on the right achieves this better. This goal is achieved by iteratively

updating the mean vector m, the covariance matrix C and a step-size σ.

The mean vector m of the distribution is updated as in Equation 2.2, where each x_i is sampled from the distribution: $x_i \sim m + \sigma\mathcal{N}(0, C)$, μ is the number of individuals taken for recombination, λ the population size and w_i are the weighted coefficients which sum to 1.

$$m = \sum_{i=1}^{\mu} w_i x_{i:\lambda} \qquad w_i > 0, \quad i = 1, 2, \ldots, \mu \tag{2.2}$$

An *Evolution Path* is defined as a sequence of consecutive steps over several generations, and they contain relevant information about the correlation between them. If two consecutive steps are taken in the same direction, the evolution path becomes longer. Evolution paths are used to guide the updates of both C and σ, in order to provide an evolution mechanism that does not converge prematurely but leads to an optimum. *Cumulation* (or *Exponential Smoothing*) is used to build the evolution path at generation g for C (denoted $p_c^{(g)}$), as described by the equation 2.3, where c_c is a defined constant ≤ 1 and $N(x) = \sqrt{x(2-x)\mu_{ef}}$ is a normalization factor for $p_c^{(g)}$.

$$p_c^{(g)} = (1 - c_c)p_c^{(g-1)} + \frac{(m - m^{(g-1)})}{\sigma^{(g-1)}} N(c_c) \tag{2.3}$$

Similarly, an evolution path is updated for step-size control, as shown in Equation 2.4. For a more detailed description of these equations, the interested reader is referred to (Hansen, 2006).

$$p_\sigma^{(g)} = (1 - c_\sigma)p_\sigma^{(g-1)} + \frac{(m - m^{(g-1)})}{\sigma^{(g-1)}} N(c_\sigma) C^{(g)^{(-\frac{1}{2})}} \tag{2.4}$$

Once the cumulations have been calculated, both C and σ can be updated. The value of C for the next generation is obtained by applying the *Rank-σ update*, which is shown in Equation 2.5. The *Rank-σ update* C_μ extends the rank-one update ($(p_c^{(g)} p_c^{(g)})^T$) for large population sizes.

$$C^{(g+1)} = (1 - c_1 - c_\mu)C^{(g)} + c_1(p_c^{(g)} p_c^{(g)})^T + c_\mu C_\mu \tag{2.5}$$

$$\text{where} \quad C_\mu = \sum_{i=1}^{\mu} w_i y_{i:\lambda} y_{i:\lambda}^T, \qquad y_i \sim \mathcal{N}(0, C) \tag{2.6}$$

Finally, the step-size σ is updated using the cumulative path length control method (CSA, or cumulative step-size adaptation). The reasoning is the following: if the evolution path $p_\sigma^{(g)}$ is long, the steps take similar directions, ergo the same distance can be covered with longer but fewer iterations.

Algorithm 2 The CMA-ES Algorithm.

function CMA-ES(m, σ, λ)
$C \leftarrow I$, $p_c \leftarrow 0$, $p_\sigma \leftarrow 0$
$c_c, c_\sigma \approx 4/n, c_1 \approx \frac{2}{n^2}, c_\mu \approx \frac{\mu_w}{n^2}, c_1 + c_\mu \leq 1$
$d_\sigma \approx 1 + \sqrt{\frac{\mu_w}{n}}, \mu_w = \frac{1}{\sum_{i=1}^{\mu} w_i^2} \approx 0.3\lambda$
while not terminate **do**
SamplePopulation ▷ From $m + \mathcal{N}(m, C)$
UpdateMean ▷ Eq. 2.2.
GetCumulation(C) ▷ Eq. 2.3.
GetCumulation(σ) ▷ Eq. 2.4.
Update(C) ▷ Eq. 2.5.
Update(σ) ▷ Eq. 2.7.

Hence, the step size must be increased. However, if $p_\sigma^{(g)}$ is short, the steps take opposite directions (they cancel each other) and their size should be decreased. Then, given $p_\sigma^{(g)}$, σ is updated as in Equation 2.7

$$\sigma^{(g+1)} = \sigma^{(g)} \quad \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|p_\sigma^{(g)}\|}{E\|\mathcal{N}(0,I)\|} - 1\right)\right) \tag{2.7}$$

An initial mean m, step-size σ and population size λ must be provided at the beginning of the algorithm. The pseudo-code from Algorithm 2 summarizes CMA-ES.

2.3 Procedural content generation

Procedural Content Generation (PCG) refers to the generation of game content (e.g. levels, maps, items, quests, puzzles, texts) with no or limited human intervention. PCG has occasionally been featured in video games during the last three decades, but it has only become a topic of academic interest within the last few years (Togelius, Yannakakis, Stanley, and Browne, 2011). PCG is becoming more relevant owing to its wide applicability to different areas within video-games. One of the earliest uses of PCG for games is the space trading game *Elite* (Acornsoft, 1984), which employed the procedural representation of star systems to reduce memory consumption. More recently, PCG has been used extensively for level or map generation in games such as *Spelunky* (Mossmouth, 2012) and those in the *Civilization* (Firaxis Games, 2012), *Diablo* (Blizzard, 1998) and *Borderlands* (Gearbox Software, 2009) game series. PCG has also been used to create literally endless games, with an infinite number of levels and open spaces that extend for as long as the player bothers to look. Some examples include *Elite* (Acornsoft, 1984), *The Sentinel* (Geoff Crammond, 1987) and *Malevolence: The Sword of Ahkranox* (Visual Outbreak, 2012).

In academic PCG research, the focus is often on ways of searching a space of game content for artefacts that satisfy certain criteria or objectives. For instance, Hastings et al. (Hastings, Guha, and Stanley, 2009) created weapons in the game *Galactic Arms Race* (GAR) using a collaborative evolutionary algorithm. Whitehead (Whitehead, 2010) suggests that PCG can improve game aesthetics, and an example of this use of PCG is given by Liapis et al. (Liapis, Yannakakis, and Togelius, 2012), who study the automatic generation of game spaceships through interactive neuroevolution. PCG can also be used to generate rules in a game or even complete games. This has been extensively addressed by C. Browne (Browne, 2008). The author describes a game description language that can be used to specify the complete rule set that defines a board game, including starting position, valid moves, winning conditions, type of board, number of players, etc. He then proposes an evolutionary algorithm that creates complete new games using the description language developed.

Adaptation to the type of player is another motivation for PCG (Yannakakis and Togelius, 2011). There are many different types of players, ranging from hardcore to occasional players, and not all of them play the same types of games. This is especially true in the last few years, when new games have focused on more unexplored genres, such as "family" or fitness games. PCG could be used to adapt the game to the type of player, producing more appropriate content by, for example, adjusting the difficulty to the skill of the player, or the type of challenge to that preferred by the player.

Another important motivation for the use of PCG, especially in the development of high-budget commercial video games, is the possibility of reducing production time and costs. An example of this is the SpeedTree software ((2012), IDV), a tool for automatic creation of vegetation. This tool has been used in numerous recent games, and some games that employ this algorithm are very popular in the gaming community, such as *Grand Theft Auto IV* (Rockstar Games, 2008) and *Fallout 3* (Bethesda Game Studios, 2008).

Problems of game content generation share many characteristics with design problems in other application areas involving interactive or complex systems. For example, in circuit board design, logistics, and road network planning, intricate path networks have to be designed while taking into account constraints relating to order, speed and interference. Robot and vehicle design problems involve designing for dynamical systems with non-deterministic behaviour, reaffirming the potential for such techniques being applicable to automatically solving other design problems.

The two key considerations when using evolution or some similar search/optimization algorithm to generate content are content evaluation (fitness function) and representation. A survey defining these problems, approaches to them, and the application of PCG in the literature can be found in (Togelius et al., 2011). Regarding content representation, this topic is central to several related fields, and surveys have been written from the perspectives of evolutionary computation (Stanley and Miikkulainen, 2003) and of artificial life (Ashlock, McGuinness, and Ashlock, 2012). Typically, there are many possible ways of representing some types of game content (Ashlock, Lee, and McGuinness, 2011), ranging from direct to more indirect approaches (Togelius et al., 2011, p. 4). For example, a dungeon could be represented in many different ways including the following:

1. Grid cell: each element in the game is specifically placed in a source matrix.
2. List of positions, orientations, and sizes of areas.
3. Repository of segments or *chunks* of levels to combine.
4. Desired properties: number of corridors, rooms, etc.
5. Random number seed: the level generator creates a level, taking only a number as input.

On the other hand, the generator usually needs a procedure to evaluate the quality of fitness of the generated levels. This function should rate or rank, in order to be able to sort them from best to worst. There are several different ways this could be done:

- Direct evaluation: some objective features are retrieved from the map and a score for the level is provided by an evaluation function.
- Simulation-based evaluation: a programmed bot plays the game and a measure of its performance is taken, which is used to score the maps. This bot can be fully hand coded or it can imitate a human player using machine learning techniques.
- Interactive evaluation: a human plays the game and feedback is obtained from him. This can be done either explicitly, with a questionnaire, or implicitly, measuring features of the gameplay, such as death locations, actions taken, distance travelled, etc.

2.4 Multi-Objective Optimization

A multi-objective optimization problem (MOP) represents a scenario where two or more conflicting objective functions are to be optimized at the same time and is defined as:

$$optimize \quad \{f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x})\} \tag{2.8}$$

subject to $\vec{x} \in \Omega$, involving $m(\geq 2)$ conflicting objective functions $f_i : \Re^n \to \Re$. The *decision vectors* $\vec{x} = (x_1, x_2, \cdots, x_n)^T$ belong to the feasible region $\Omega \subset \Re^n$. We denote the image of the feasible region by $Z \subset \Re^m$ and call it a feasible objective region. The elements of Z are called objective vectors and they consist of m objective (function) values $\vec{f}(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x}))$. Therefore, each solution $\vec{x}$ provides m different scores (or rewards, or fitness) that are meant to be optimized. Without loss of generality, it is assumed from now on that all objectives are to be maximized.

It is said that a solution $\vec{x}$ *dominates* another solution $\vec{y}$ if and only if:

1. $f_i(\vec{x})$ is not worse than $f_i(\vec{y})$, $\forall i = 1, 2, \ldots, m$.

2. For at least one objective j: $f_j(\vec{x})$ is better than its analogous counterpart in $f_j(\vec{y})$.

When these two conditions apply, it is said that $\vec{x} \preceq \vec{y}$ ($\vec{x}$ dominates $\vec{y}$). The *dominance* condition provides a partial ordering between solutions in the objective space.

However, there are some cases where it cannot be said that $\vec{x} \preceq \vec{y}$ or $\vec{y} \preceq \vec{x}$. In this case, it is said that these solutions are indifferent to each other. Solutions that are not dominated can be grouped in a *non-dominated set*. Given a non-dominated set P, it is said that P is the *Pareto-set* if it is the largest *non-dominated set* P such that no points outside the set dominate any other point in P. The corresponding objective vectors of the members in the Pareto-set build a so called *Pareto-front*. The relations between decision and objective space, dominance and the non-dominated set are depicted in Figure 2.3.

There are many different algorithms in the literature that are proposed to tackle multi-objective optimization problems (Deb, 2001). A weighted-sum approach is one of the traditional methods in which the objectives are weighted according to user preference. The sum of the weighted objectives builds one objective function to be optimized. This so-called **linear scalarization approaches** transform a set of objectives into a single objective by multiplying each objective with a user-supplied weight (Deb,

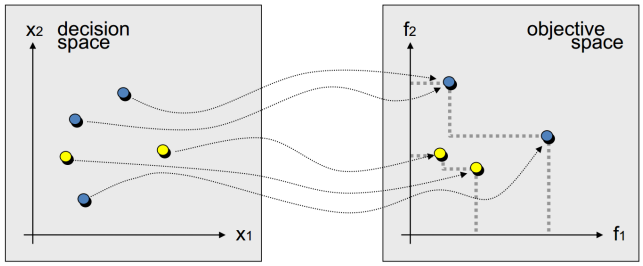


Figure 2.3: *Decision and objective spaces in a MOP with two variables (x_1, x_2) and two objectives (f_1, f_2). In the objective space, yellow dots are non-optimal objective vectors, while blue dots form a non-dominated set. From (Brockhoff, 2013).*

2001). The solution to this single-objective problem is one certain solution, ideally on a Pareto-front. By varying the weights, it is possible to converge to different optimal solutions. Such linear scalarization approaches fail to find optimal solutions for problems with non-convex-shape Pareto-fronts (Deb, 2001).

A popular choice for multi-objective optimization problems are evolutionary multi-objective optimization (EMO) algorithms (Coello, 1999; Zhou, Qu, Li, Zhao, Suganthan, and Zhang, 2011). The goal of EMO algorithms is to find a set of optimal solutions with a good convergence to the Pareto-front, as well as a good spread and diversity along the Pareto-front. One of the most well-known algorithms in the literature is the Non-dominated Sorting Evolutionary Algorithm II (NSGA-II) (Deb, Pratap, Agarwal, and Meyarivan, 2002), the pseudocode of which is shown in Algorithm 3. NSGA-II is known to deliver solutions with very good diversity and convergence to the Pareto-front for 2-objective problems. However, for problems with more than 3 objectives, other approaches such as NSGA-III (Jain and Deb, 2013), SMS-EMOA (Beume, Naujoks, and Emmerich, 2007) and MOEA/D (Zhang and Li, 2007) are shown to outperform NSGA-II. As in any evolutionary algorithm, NSGA-II evolves a set of N individuals in a population denoted as Pop with the difference that they are ranked according to a dominance criterion and a crowding distance measure, which are used to maintain a good diversity of solutions. After each iteration of the algorithm, only the best N individuals according to this ranking are maintained to the next generation.

The three main pillars of the NSGA-II algorithm are:

- Fast non-dominated sorting: The function $F = \text{FASTNONDOMINATEDSORT}(U(t))$ performs a non-dominated sorting and ranks the individuals stored in a set U into several non-dominated fronts denoted as F_i, where the solutions in F_0 are the non-dominated solutions in the entire set. F_i is the set of non-dominated solutions in $R \setminus (F_0 \cup \cdots \cup F_{i-1})$.

Algorithm 3 NSGA-II Algorithm (Deb et al., 2002)

Input: MOP, N
Output: Non-dominated Set F_0
$t = 0$
$Pop(t) = NewRandomPopulation$
$Q(t) = breed(Pop(t))$ % Generate offspring
while Termination criterion not met **do**
    $U(t) = Pop(t) \cup Q(t)$
    $F = \text{FASTNONDOMINATEDSORT}(U(t))$
    $Pop(t+1) = \emptyset$, $i = 0$
    while $|Pop(t+1)| + |F_i| \leq N$ **do**
        $\text{CROWDINGDISTANCEASSIGNMENT}(F_i)$
        $Pop(t+1) = Pop(t+1) \cup F_i$
        $i = i + 1$
    $\text{SORT}(Pop(t+1))$
    $Pop(t+1) = Pop(t+1) \cup F_i[1 : (N - |Pop(t+1)|)]$
    $Q(t+1) = breed(Pop(t+1))$
    $t = t + 1$
return F_0

- Crowding distance: The function $\text{CROWDINGDISTANCEASSIGNMENT}(F_i)$ assigns to each one of the individuals in F_i a crowding distance value, which is the distance between the individual and its neighbours. The individuals with the smallest crowding distances are selected with a lower probability than the ones with larger values.

- Elitism: The individuals from the first front F_0 are always ranked first, according to the dominance criterion and their crowding distance. This ensures that the best solutions always survive to the next generation.

For more details about EMO approaches, please refer to (Deb, 2001).

2.4.1 Multi-objective Reinforcement Learning (MORL)

Reinforcement Learning (RL) algorithms have also been used with MOPs. RL (Sutton and Barto, 1998) is a broad field in Machine Learning that studies real-time planning and control situations where an agent has to find the actions (or sequences of actions) that should be used in order to maximize the reward from the environment.

The dynamics of an RL problem are usually captured by a Markov Decision Process (MDP) which is a tuple (S, A, T, R), in which S is the set of possible states in the problem (or game), and s_0 is the initial state. A is the set of available actions the agent can make at any given time, and the transition model $T(s_i, a_i, s_{i+1})$ determines the probability of reaching the state s_{i+1} when action a_i is applied in

state s_i. The reward function $R(s_i)$ provides a single value (*reward*) that the agent must optimize, representing the desirability of the state s_i reached. Finally, a policy $\pi(s_i) = a_i$ determines which actions a_i must be chosen from each state $s_i \in S$[1]. One of the most important challenges in RL, as previously shown in MCTS, is the trade-off between exploration and exploitation while trying to act. While learning, a policy must choose between selecting the actions that provided good rewards in the past and exploring new parts of the search space by selecting new actions.

Multi-objective Reinforcement Learning (MORL) (Vamplew, Dazeley, Berry, Issabekov, and Dekker, 2010) changes this definition by using a vector $R = r_1, \ldots, r_m$ as rewards of the problem (instead of a scalar). Thus, MORL problems differ from RL in having more than one objective (here m) that must be maximized. If the objectives are independent or non-conflicting, scalarization approaches such as the weighted sum approach, could be suitable to tackle the problem. Essentially, this would mean using a conventional RL algorithm on a single objective where the global reward is obtained from a weighted-sum of the multiple rewards. However, this is not always the case, as usually the objectives are conflicting and the policy π must balance among them.

Vamplew et al. (Vamplew et al., 2010) propose a *single-policy* algorithm which uses a preference order in the objectives (either given by the user or by the nature of the problem). An example of this type of algorithm can be found at (Gabor, Kalmar, and Szepesvari, 1998), where the authors introduce an order of preference in the objectives and constrain the value of the desired rewards. Scalarization approaches would also fit in this category, as shown in the work performed by S. Natarajan et al. (Natarajan and Tadepalli, 2005).

The second type of algorithms, *multiple-policy*, aims to approximate the optimal Pareto-front of the problem. An example of this type of algorithm is the one given by L. Barrett (Barrett and Narayanan, 2008), who proposes the Convex Hull Iteration Algorithm. This algorithm provides the optimal policy for any linear preference function, by learning all policies that define the convex hull of the Pareto-front.

2.4.2 Metrics

The quality of an obtained non-dominated set can be measured using different diversity or/and convergence metrics (Deb, 2001). The Hypervolume Indicator (HV) is a popular metric for measuring both the

[1]This is a deterministic policy and only valid during acting, not learning

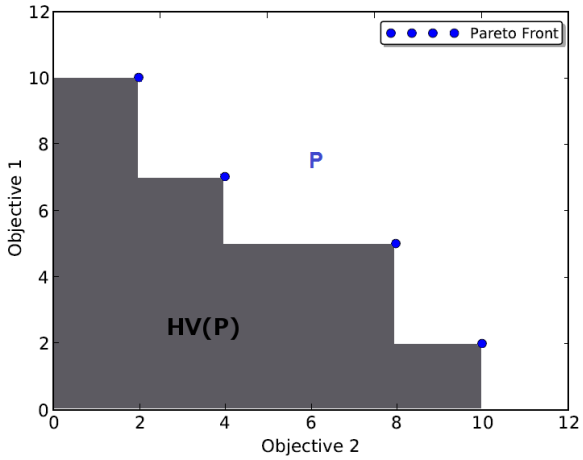


Figure 2.4: *$HV(P)$ of a given Pareto-front P*

diversity and convergence of non-dominated solutions (Zitzler, 1999). This approach can be additionally used to compare different non-dominated sets. Given a Pareto front P, $HV(P)$ is defined as the volume of the objective space dominated by P. More formally, $HV(P) = \mu(x \in \mathbb{R}^d : \exists r \in P \quad s.t \quad r \preceq x)$, where μ is the de Lebesgue measure on $\mathbb{R}^d$. If the objectives are to be maximized, the higher the $HV(P)$, the better the front calculated. Figure 2.4 shows an example of $HV(P)$ where the objective dimension space is $m = 2$.

2.5 General Game and Video-Game Playing

Traditionally, research in Game AI presents algorithms that determine strategies for agents that play a particular game. For instance, advances in algorithms employed to play the game of Go focus on how new proposals, modifications and heuristics improve the winning rate of the agents that implement these algorithms. The proposed approaches can count on heuristics that are tailored to the game of Go, usually suggested or designed by an expert in this specific game. These heuristics are used to guide the search to states with high chances of victory.

In contrast to this, research on General Game Playing (GGP) and General Video Game Playing (GVGP) focuses on algorithms for agents that play a set of significantly different games. In fact, the agent must be able to play any game, even some of them that are encountered for the first time, without the possibility of a previous training on them. Consequently, it is not possible to provide the algorithm with domain knowledge, as in Game AI research focused on a specific game. Therefore, the

main challenge of GGP and GVGP is that the agent needs to be general enough to learn the structure of any game, guide the search to relevant states of the search space, and be able to adapt to a wide variety of situations in the absence of game-dependent heuristics.

The goal of GGP and GVGP is to provide a framework for algorithm testing where the usage of game-specific knowledge is significantly limited. While general players are able to deal with a wide diversity of games, specialized players tend to employ abundant domain knowledge for solving the task at stake, resulting in algorithms that play that particular game well, but are of little interest to AI. The idea of having general agents that are able to deal with a large variety of situations is one of the most important goals of AI (Russell, 1997).

One of the first attempts to develop and establish a general game playing framework was carried out by the Stanford Logic Group of Stanford University, when they organized the first AAAI GGP competition in 2005 (Genesereth, Love, and Pell, 2005). In this competition, players would receive declarative descriptions of games at runtime (hence, the game rules were unknown beforehand), and would use this information to play the game effectively. These competitions feature finite and synchronous games, described in a Game Definition Language (GDL), and include games such as Chess, Tic-tac-toe, Othello, Connect-4 or Breakthrough.

Among the winners of the different editions, the Upper Confidence bounds for Trees (UCT) and Monte Carlo Tree Search (MCTS) approaches deserve a special mention. CADIA-Player, developed by Hilmar Finnsson in his Master's thesis (Finnsson and Björnsson, 2008, 2009), was the first MCTS based approach to be declared winner of the competition, in 2007. The agent used a form of historic heuristic and parallelization, capturing game properties during the algorithm roll-outs.

The winner of two later editions, 2009 and 2010, was another MCTS based approach, developed by J. Méhat and T. Cazenave (Méhat and Cazenave, 2011). This agent, named Ary, studied the concept of parallelizing MCTS in more depth, implementing the *root parallel algorithm*. The idea behind this technique is to perform independent Monte-Carlo tree searches in parallel in different CPU. When the decision time allowed to make a move is over, a master component decides which action to take among the ones suggested by the different trees.

Other interesting MCTS-based players for GGP are *Centurio*, from Möller et al. (Möller, Schneider, Wegner, and Schaub, 2011), that combines parallelized MCTS with Answer Set Programming

(ASP), and the agent by Sharma et al. (Sharma, Kobti, and Goodwin, 2008), that generates domain-independent knowledge and uses it to guide the simulations in UCT.

In GGP, the time allowed for each player to pick a move can vary from game to game. The time allowed is usually indicated in seconds. Therefore, any player will be able to spend at least 1 second of decision time in choosing the next action to make. Whilst this is an appropriate amount of time for the types of games that feature in GGP competitions, a similar decision time cannot be afforded in real-time games. Here the agents perform actions at a much higher rate, making them appear almost continuous to a human player, in contrast to turn-based games. During the last few years, important research has been carried out in the field of general video-game playing, specifically in arcade games where the games are clearly not turn-based and where the time allowed for the agents to pick an action is measured in milliseconds.

Most of this research has been performed in games from the Atari 2600 collection. Bellemare et al. (Bellemare, Naddaf, Veness, and Bowling, 2013) introduced the Arcade Learning Environment (ALE), a platform and a methodology to evaluate AI agents in domain-independent environments, employing 55 games from Atari 2600. In this framework, each observation consists of a single frame: a two dimensional (160×210) array of 7-bit pixels. The agent acts every 5 frames (12 times per second), and the action space contains the 18 discrete actions allowed by the joystick. The agent, therefore, must analyze the screen to identify game objects and perform a move accordingly, in a decision time of close to 80ms.

Several approaches have been proposed to deal with this kind of environment, such as Reinforcement Learning and MCTS (Bellemare et al., 2013). M. Hausknecht et al. (Hausknecht, Lehman, Miikkulainen, and Stone, 2013) employed evolutionary neural networks to extract higher-dimensional representation forms from the raw game screen. Yavar Naddaf, in his Master's thesis (Naddaf, 2010), extracted feature vectors from the game screen, that were used in conjunction with Gradient-descent Sarsa(λ) (Szepesvári, 2010) and UCT.

Also in this domain, Bellemare et al. (Gendron-Bellemare, Veness, and Bowling, 2012) explored the concept of contingency awareness using Atari 2600 games. Contingency awareness is "the recognition that a future observation is under an agent's control and not solely determined by the environment" (Gendron-Bellemare et al., 2012, p. 2). In this research, the authors show that contingency

awareness helps the agent to track objects on the screen and improve on existing methods for feature construction.

Similar to the Atari 2600 domain, J. Levine et al. (Levine, B. Congdon, Bída, Ebner, Kendall, Lucas, Miikkulainen, Schaul, and Thompson, 2013) recently proposed the creation of a benchmark for General Video Game playing that complements Atari 2600 in two ways: the creation of games in a more general framework, and the organization of a competition to test the different approaches to this problem. Additionally, in this framework, the agent does not need to analyze a screen capture, as all information is accessible via encapsulated objects. It is also worthwhile highlighting that the game rules are never given to the agent, something that is usually done in GGP.

This new framework is based on the work of Tom Schaul (Schaul, 2013), who created a Video Game Description Language (VGDL) to serve as a benchmark for learning and planning problems. The first edition of the General Video Game AI (GVGAI) Competition, will be held at the IEEE Conference on Computational Intelligence and Games (CIG) in 2014 (Perez, Samothrakis, Togelius, Schaul, and Lucas, 2014d). Research work described in Chapter 9 uses the framework of this competition as a benchmark, as well as some of the games that feature in the contest.

"A supernova! Creation itself! I was there. I wanted to see it and be part of the moment. And you know how I perceived one of the most glorious events in the Universe? ..."
- Brother Cavil, Battlestar Galactica

Chapter 3

Games in this Thesis

This chapter explains the different games used in this research.

3.1 The 2005 Physical Travelling Salesman Problem (PTSP-2005)

The Physical Travelling Salesman Problem (PTSP) is an extension of the Travelling Salesman Problem (TSP). The TSP is a very well known combinatorial optimization problem in which a salesperson has to visit n cities exactly once using the shortest route possible, returning to the starting point at the end (Johnson and McGeoch, 1997). The PTSP converts the TSP into a single-player game and was first introduced as a competition at the Genetic and Evolutionary Computation Conference (GECCO) in 2005[1]. In the PTSP-2005, the player always starts in the centre of the map and cities are usually distributed uniformly at random within some rectangular area; the map itself is unbounded.

Although the original game was not time constrained, the goal of PTSP-2005 is to find the best solution in real-time. At each game tick the agent selects one of five force vectors to be applied to accelerate a point mass around the map with the aim of visiting all cities. The optimality of the route is the time taken to traverse it which differs from its distance as the point-mass may travel at different speeds. At any moment in time, a total of 5 actions may be taken: forward, backward, left, right and neutral. At each time step, the position and velocity of the point-mass is updated using Newton's equations for movement: $v = v_i + a\Delta t$ and $s = s_i + v_i\Delta t + \frac{1}{2}a(\Delta t)^2$ with $\Delta t = \sqrt{0.1}$.

There are at least two high-level approaches to confront this problem: one possibility is to address the order of cities and the navigation (steering) of the point mass independently. However, it is important

[1]This game is referred to in this thesis as *PTSP-2005*. It is fundamentally different to the 2012 edition of the same game, described in Section 3.2, which is referred to as simply *PTSP*.

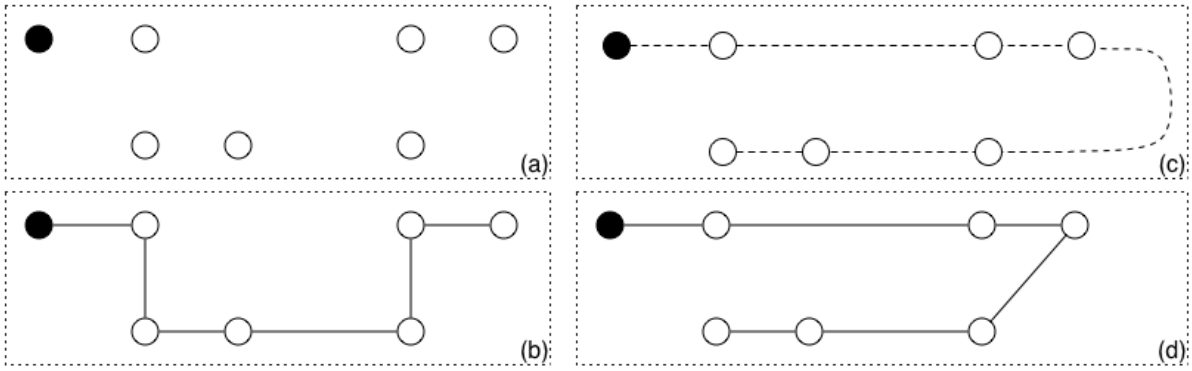


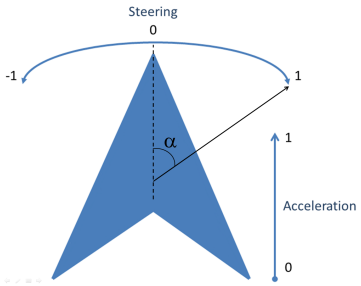
Figure 3.1: *Example of a 6 city problem where the optimal TSP route differs from the optimal PTSP-2005 route: (a) the six cities and starting point (black circle); (b) the optimal TSP solution to this problem without returning to the start; (c) optimal PTSP-2005 route and (d) equivalent TSP route which is worse than the route shown in (b).*

to keep in mind that the physics of the game make the PTSP-2005 quite different from the TSP. In particular, the optimal order of cities for a given map which solves the TSP does not usually correspond to the optimal set of forces that can be followed by an agent in the PTSP-2005. This is illustrated in Figure 3.1. Another possible approach to tackle the PTSP-2005 is thus to attempt to determine the optimal set of forces and order of cities simultaneously.

The PTSP-2005 can be seen as an abstract representation of video games characterised by two game elements: order selection and steering. Examples of such games include CrystalQuest (Macintosh, 1987), XQuest (Mark Mackey, 1994) and Crazy Taxi (Hitmaker, 1999). In particular, the PTSP-2005 has numerous interesting attributes that are commonly found in these games: players are required to act quickly as the game progresses at every time step. Furthermore, the game is open-ended as the point-mass may travel across an unbounded map. Therefore, it is highly unlikely that algorithms that evaluate states via simulated moves (such MCTS), with a limited time budget to pick an action to take, would be able to reach a terminal game state. This requires the algorithm to (a) limit the number of actions to take on a simulation; and (b) implement a value function that scores terminal and non-terminal states.

3.2 The 2012 Physical Travelling Salesman Problem (PTSP)

The Physical Travelling Salesman Problem (PTSP) is a variation of the 2005 Physical Travelling Salesman Problem (PTSP-2005, see Section 3.1), that at the same time is an adaptation of the classical Travelling Salesman Problem (TSP).



Action ID	Acceleration	Steering
0	No (0)	No (0)
1	No (0)	Left (-1)
2	No (0)	Right (1)
3	Yes (1)	No (0)
4	Yes (1)	Left (-1)
5	Yes (1)	Right (1)

Figure 3.2: *Action space of the PTSP*

In this renewed version of the PTSP, the player (i.e. the salesman) governs a spaceship that must visit a series of waypoints scattered around a maze as quickly as possible. The PTSP is a real-time game, implying that an action must be supplied every 40 milliseconds. The available actions in this version are summarized in Figure 3.2. These actions are composed of two different inputs applied simultaneously: acceleration and steering. Acceleration can take two possible values (*on* and *off*), while steering can turn the ship to the *left*, *right* or keep it *straight*. This leads to a total of six different actions. In case the controller fails to return an action after the time limit, an *NOP* action (ID: 0) is applied, which performs no steering and no acceleration.

The state of the ship is kept from one game step to the next, and three different vectors are modified after applying a single action. The orientation of the ship is changed as shown in Equation 3.1, given the ship's orientation in the last step d_t and the rotation angle α (a fixed angle that can be positive or negative, depending on the sense of the steering input, or 0 if the ship is told to go straight).

$$d_{t+1} \coloneqq \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} d_t \tag{3.1}$$

Equation 3.2 indicates how the velocity vector is modified, given the previous velocity v_t, the new orientation, an acceleration constant (K) and a frictional loss factor (L). In this case, the acceleration input determines the value of T_t: being 1 if the action implies thrust or 0 otherwise.

$$v_{t+1} \coloneqq (v_t + (d_{t+1} T_t K))L \tag{3.2}$$

Finally, Equation 3.3 updates the position of the ship by adding the velocity vector to its location in the previous game step (p_t). This physics model keeps the inertia of the ship between game cycles,

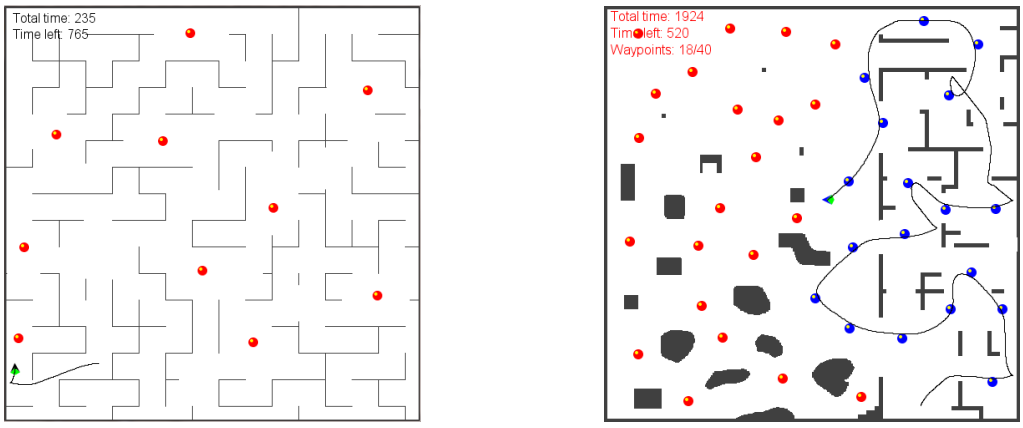


Figure 3.3: *PTSP maps where the ship has to visit all waypoints (red dots) scattered around the maze. The map on the left contains only* 10 *waypoints, whereas the one on the right presents* 40*.*

making the task of navigating the ship more challenging.

$$p_{t+1} \coloneqq p_t + v_{t+1} \tag{3.3}$$

The obstacles of the maze do not damage the ship, but hitting them produces an elastic collision, which modifies the velocity of the ship (both in direction and magnitude). Figure 3.3 shows examples of maps distributed with the game.

In addition to returning an action every 40 milliseconds, a controller designed for the PTSP must respect two other time limits: an initialization time (1 second), and a limit to reach the next waypoint in the game (1000, 700, 550 and 400 time steps for maps with 10, 30, 40 and 50 waypoints respectively). The latter counter, T, is reset to this limit every time a waypoint is visited, and is decreased by 1 at every cycle. If the value ever reaches 0, the game is over.

The final score of the game is defined by the tuple (W, T) where W is the number of waypoints visited and T is the total time spent. Between two matches, the one that gets more waypoints is considered to be the best. In case of a draw, the one that finished the game in fewer time steps wins.

At every game step, the controller is supplied with a copy of the current game state and point in time when the move is due. The game state contains information about the ship itself (position, velocity, orientation), the game (time left, number of waypoints visited and remaining), the map (position of obstacles and path-finding queries) and waypoints (positions and whether they have been visited or not). However, the most important feature of this game state is that it provides a *forward model* to run simulations. In other words, it is possible to check what the future states of the game would be

if a specific sequence of moves were played. The PTSP is a deterministic and single-player game and the simulations performed are completely accurate. This makes tree search and simulation approaches especially well suited for this kind of problem.

As in the PTSP-2005, the physics of the game have an important effect on the problem because it requires an ordering of waypoints that may not correspond to the optimal, Euclidean distance-based TSP route (see Section 3.1, Figure 3.1).

This version of the PTSP featured in two PTSP competitions (Perez, Rohlfshagen, Robles, and Lucas, 2012d) held at major international conferences: the IEEE 2012 World Congress on Computational Intelligence (WCCI) and the IEEE 2012 Conference on Computational Intelligence and Games (CIG). Details of these competitions and their results are given in Appendix A.

Despite its simplicity, the PTSP retains some of the most important aspects of modern video-games: it requires the player to deal with navigation, obstacle avoidance and pathfinding. Furthermore, as it is a real-time game, it does not allow much time to decide what is the next move to make (as would happen in other games such as Chess or Go) or to plan an overall strategy for the game. These characteristics make the PTSP a very good benchmark to try a wide range of techniques that could be exported to real video-games in the future.

3.2.1 Maps and Controllers

The PTSP framework contains multiple maps with 10 (as featured in the WCCI 2012 competition), 30, 40 and 50 (employed in the CIG 2012 competition) waypoints. All these maps were designed by hand, keeping in mind that a PTSP map is valid if and only if it respects the following requirements:

1. Connectivity of waypoints: there must be a valid path between each pair of waypoints in the map. This can be achieved by computing the A* path between them using the pathfinding library included in the framework.

2. Initial position for the ship: the position where the ship starts must be at a safe distance from the closest wall.

3. Positions for the waypoints: as in the previous case, each waypoint's position must be at a safe distance from the closest obstacle.

4. Ship vs. waypoint distance: the initial distance of the ship from all waypoints must be significant to avoid visits being too quick. The minimum distance for this, and for the two previous points (2 and 3 above), is set to 10 times the radius of the ship.

5. Waypoint vs. waypoint distance: the distances between every pair of waypoints cannot be too low to avoid multiple visits at once. The minimum distance is set to 5 times the radius of the ship.

As mentioned above, the code distributed with the PTSP framework includes a pathfinding library that can be used by the agents that play the game. This library creates a grid graph on the navigable sections of the map and may be used to obtain the shortest path between the position of the ship and any other point in the map, accounting for obstacles in the way. Finally, the software also includes several sample controllers that behave as follows:

- `RandomController`: executes a random action at every time step (from the six actions available).
- `LineOfSight`: if there is no unvisited waypoint in the ship's line of sight, the controller executes a random action. Otherwise, the ship moves in a straight line towards the visible waypoint.
- `GreedyController`: this controller makes use of the pathfinding library included in the framework. It calculates and follows the shortest path from the ship to the closest waypoint.

3.3 Deep Sea Treasure (DST)

The Deep Sea Treasure (DST) is a popular multi-objective problem introduced by Vamplew et al. (Vamplew et al., 2010). In this single-player puzzle, the agent controls a submarine with the objective of finding a treasure located at the bottom of the sea. The world is divided into a grid of 10 rows and 11 columns, and the vessel starts at the top left board position. There are three types of cells: empty cells (or water), that the submarine can traverse; ground cells that, as the edges of the grid, cannot be traversed; and treasure cells that provide different rewards and finish the game. Figure 3.4 shows the DST environment.

The ship can perform four different moves: *up*, *down*, *right* and *left*. If the action applied takes the ship off the grid or into the sea floor, the vessel's position will not change. There are two objectives in

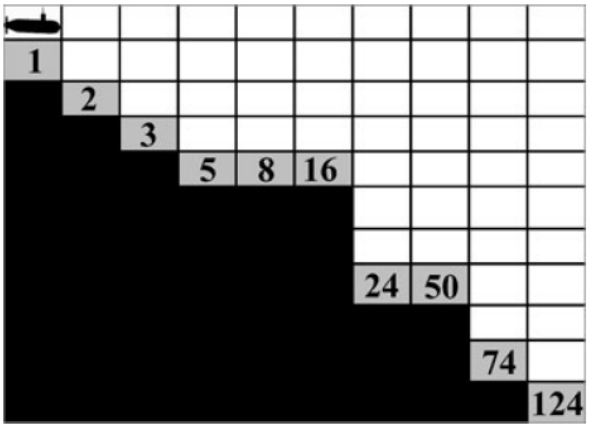


Figure 3.4: *Environment of the Deep Sea Treasure (from (Vamplew et al., 2010)): grey squares represent the treasure (with their different values) available in the map. The black cells are the sea floor and the white ones are the positions that the vessel can occupy freely. The game ends when the submarine picks one of the treasures.*

this game: the number of moves performed by the ship, which must be minimized, and the value of the treasure found, which should be maximized. As can be seen in Figure 3.4, the most valuable treasures are at a greater distance from the initial position, so the objectives are in conflict.

Additionally, the agent can only make up to 100 plies or moves. This allows the problem to be defined as the maximization of two rewards: $(\rho_p, \rho_v) = (100 - plies, treasureValue)$. Should the ship perform all moves without reaching a treasure, the result would be (0,0). At each step, the score of a location with no treasure is $(-1, 0)$.

The optimal Pareto front of the DST is shown in Figure 3.5. There are 10 non-dominated solutions in this front, one per each treasure in the board. The front is globally concave, with local concavities at the second $(83, 74)$, fourth $(87, 24)$ and sixth $(92, 8)$ points from the left. The HV value of the optimal front is 10455.

Section 2.4 introduced the problems of linear scalarization approaches when facing non-convex optimal Pareto fronts. The concave shape of the DST's optimal front means that those approximations converge to the non-dominated solutions located at the edges of the Pareto front: $(81, 124)$ and $(99, 1)$. Note that this happens independently from the weights chosen for the linear approximation: some solutions of the front just can't be reached with these approaches. Thus, successful approaches should be able to find all elements of the optimal Pareto front and then be able to converge to any of the non-dominated solutions.

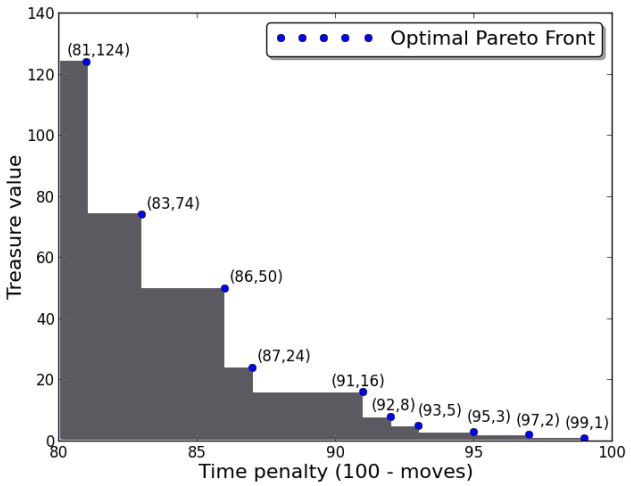


Figure 3.5: *Optimal Pareto Front of the Deep Sea Treasure, with both objectives to be maximized.*

3.4 Puddle Driver (PD)

The second benchmark used in this thesis for Multi-Objective optimization is the Puddle Driver (PD). This problem, created explicitly for this research, is a single-player real-time puzzle that takes its influences from the Multi-Objective Puddle World (MOPW) and the Physical Travelling Salesman Problem (PTSP). The Multi-Objective Puddle World, presented also by Vamplew et al. (Vamplew et al., 2010), offers a puzzle where an agent has to reach a goal in a grid based world where puddles lie between its starting position and the destination. The target is to minimize two objectives: the number of movements to reach the goal and the presence of the agent in the puddles.

The Puddle Driver presents a real-time scenario where the player drives a ship by supplying 6 different actions, like in the PTSP (see Section 3.2), combining two different inputs: *thrust* (with values *on* and *off*) and *steer* (rotating *left*, *right* or *straight*). The ship must apply an action every $40ms$, and this move will alter its position, velocity and orientation. The initial position of the ship is located at the bottom left corner of the map, while a target waypoint is placed at the top right corner. The size of the map is 512×512 pixels. There are several regions (puddles) randomly placed between these two positions, and they cause 1 unit of damage for each time step the ship drives over them. Analogously to MOPW, the objective is to minimize both quantities: time taken to reach the goal, and damage suffered in the puddles. The game ends if, after 1000 time steps, the goal is not reached. Figure 3.6 shows an example of a map for this game.

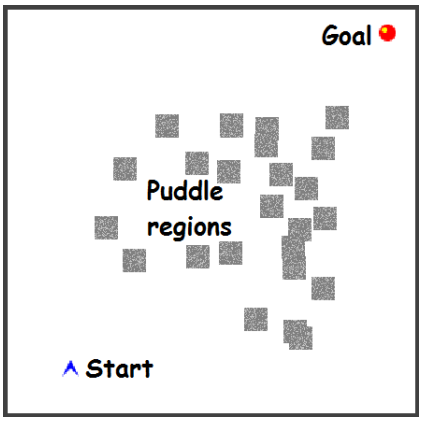


Figure 3.6: *Example map from Puddle Driver.*

3.5 Multi-Objective Physical Travelling Salesman Problem (MO-PTSP)

The Multi-Objective Physical Travelling Salesman Problem (MO-PTSP) is a game that was employed in a competition held at the IEEE Conference on Computational Intelligence in Games (CIG) in 2013 (Perez and Lucas, 2013). Details of this competition and its results are given in Appendix A.

MO-PTSP is a modification of the Physical Travelling Salesman Problem (PTSP). The MO-PTSP is a real-time game where the agent navigates a ship and must visit 10 waypoints scattered around the maze. All waypoints must be visited to consider a game as complete, and a game-tick counter is reset every time a waypoint is collected, finishing the game prematurely (and unsuccessfully) if 800 game steps are reached before visiting another waypoint.

This needs to be accomplished while optimizing three different objectives: 1) **Time**: the player must collect all waypoints scattered around the maze in as few time steps as possible; 2) **Fuel**: the fuel consumption by the end of the game must be minimized; and 3) **Damage**: the ship should end the game with as little damage as possible.

As in the PTSP (see Section 3.2), the agent must provide an action every 40 milliseconds and the available actions are combinations of two different inputs: *throttle* (that could be *on* or *off*) and *steering* (that could be *straight*, *left* or *right*), as in Figure 3.2.

The ship starts with 5000 units of fuel, and one unit is spent every time an action supplied has the throttle input *on*. There are, however, two ways of collecting fuel: each waypoint visited grants the ship 50 more units of fuel; and there are also fuel canisters scattered around the maze that provide 250 more units.

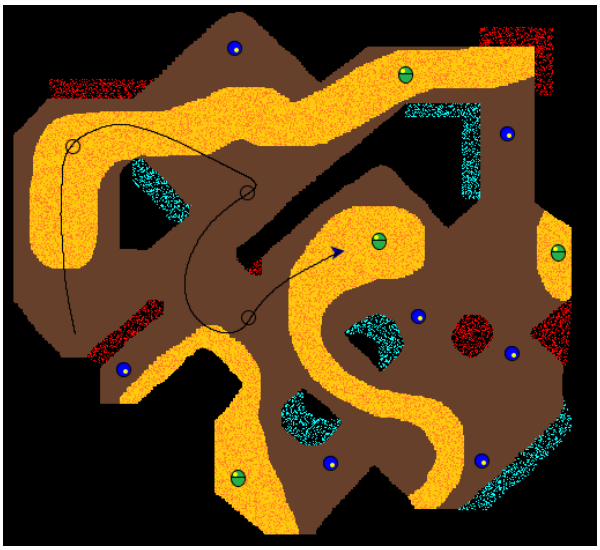


Figure 3.7: *Sample MO-PTSP map.*

Regarding the third objective, the ship can suffer damage in two different ways: by colliding with obstacles and driving through lava. In the former case, the ship can collide with normal obstacles (that subtract 10 units of damage) and specially damaging obstacles (30 units). In the latter, lava lakes are abundant in MO-PTSP levels and, in contrast with normal surfaces, they deal one unit of damage for each time step the ship spends over this type of surface. All these subtractions are deducted from an initial counter of 5000 points of damage.

Figure 3.7 shows an example of an MO-PTSP map, as drawn by the framework. Waypoints not yet visited are painted as blue circles, while those already collected are depicted as empty circles. The green ellipses represent fuel canisters, normal surfaces are drawn in brown and lava lakes are printed as red-dotted yellow surfaces. Normal obstacles are black and damaging obstacles are drawn in red. Blue obstacles are elastic walls that produce no damage to the ship. The vessel is drawn as a blue polygon and its trajectory is traced with a black line.

3.6 The GVGAI Framework

This section describes the General Video Game AI (GVGAI) framework and its games.

3.6.1 Games

The GVGAI competition (as described in Appendix A) presents several games divided into three sets: training, validation, and test. The first set is public to all competitors, in order for them to train and prepare their controllers, and is the one used in the experiments conducted for this thesis. The objective of the validation set is to serve as a hidden set of games where participants can execute their controllers in the server. Finally, the test set is another set of secret games for the final evaluation of the competition.

Each set of games is composed of 10 games, and there are 5 different levels available for each one of them. Most games have a non-deterministic element in the behaviour of their entities, producing slightly different playouts every time the same game and level is played. Also, all games have a maximum number of game steps to be played (set to 2000), in order to avoid degenerate players never finishing the game. If this limit is violated, the result of the game will count as a loss.

The 10 games from the training set are described in Table 3.1. As can be seen, the games differ significantly in winning conditions, different number of non-player characters (NPC), scoring mechanics and even in the available actions for the agent. For instance, some games have a timer that finishes the game with a victory (as in *Survive Zombies*) or a defeat (as in *Sokoban*). In some cases, it is desirable to collide with certain moving entities (as in *Butterflies*, or in *Chase*) but, in other games, those events are what actually kill the player (as in *Portals*, or also in *Chase*). In other games, the agent (or *avatar*) is killed if it collides with a given sprite, that may only be killed if the avatar picks the action `USE` appropriately (in close proximity, as when using the sword in *Zelda*, or at a greater distance, as when shooting in *Aliens*). Figure 3.8 shows some of the games from the training set.

These differences in game play make the creation of a simple game-dependent heuristic a relatively complex task, as the different mechanisms must be handled on a game per game basis. Furthermore, a controller created following these ideas, would probably fail to behave correctly in other unseen games in the competition.

3.6.2 The framework

All games are written in VGDL, a video-game description language that is able to fully define a game, typically in less than 50 lines. For a full description of VGDL, the reader should consult (Schaul, 2013).

Game	Description
Aliens	Similar to traditional Space Invaders, Aliens features the player (avatar) in the bottom of the screen, shooting upwards at aliens that approach Earth, who also shoot back at the avatar. The player loses if any alien touches it, and wins if all aliens are eliminated. **Score:** 1 point is awarded for each alien or protective structure destroyed by the avatar. −1 point is given if the player is hit. **Actions:** LEFT, RIGHT, USE.
Boulder-dash	The avatar must dig in a cave to find at least 10 diamonds, with the aid of a shovel, before exiting through a door. Some heavy rocks may fall while digging, killing the player if it is hit from above. There are enemies in the cave that might kill the player, but if two different enemies collide, a new diamond is spawned. **Score:** 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned. −1 point is given if the avatar is killed by a rock or an enemy. **Actions:** LEFT, RIGHT, UP, DOWN, USE.
Butterflies	The avatar must capture butterflies that move randomly around the level. If a butterfly touches a cocoon, more butterflies are spawned. The player wins if it collects all butterflies, but loses if all cocoons are opened. **Score:** 2 points are awarded for each butterfly captured. **Actions:** LEFT, RIGHT, UP, DOWN.
Chase	The avatar must chase and kill scared goats that flee from the player. If a goat finds another goat's corpse, it becomes angry and chases the player. The player wins if all scared goats are dead, but it loses if is hit by an angry goat. **Score:** 1 point for killing a goat. −1 point for being hit by an angry goat. **Actions:** LEFT, RIGHT, UP, DOWN.
Frogs	The avatar is a frog that must cross a road, full of tracks, and a river, only traversable by logs, to reach a goal. The player wins if the goal is reached, but loses if it is hit by a truck or falls into the water. **Score:** 1 point for reaching the goal. −2 points for being hit by a truck. **Actions:** LEFT, RIGHT, UP, DOWN.
Missile Command	The avatar must shoot at several missiles that fall from the sky, before they reach the cities they are directed towards. The player wins if it is able to save at least one city, and loses if all cities are hit. **Score:** 2 points are given for destroying a missile. −1 point for each city hit. **Actions:** LEFT, RIGHT, UP, DOWN, USE.
Portals	The avatar must find the goal while avoiding lasers that kill him. There are many portals that teleport the player from one location to another. The player wins if the goal is reached, and loses if killed by a laser. **Score:** 1 point is given for reaching the goal. **Actions:** LEFT, RIGHT, UP, DOWN.
Sokoban	The avatar must push boxes so they fall into holes. The player wins if all boxes are made to disappear, and loses when the timer runs out. **Score:** 1 point is given for each box pushed into a hole. **Actions:** LEFT, RIGHT, UP, DOWN.
Survive Zombies	The avatar must stay alive while being attacked by spawned zombies. It may collect honey, dropped by bees, in order to avoid being killed by zombies. The player wins if the timer runs out, and loses if hit by a zombie while having no honey (otherwise, the zombie dies). **Score:** 1 point is given for collecting one piece of honey, and also for killing a zombie. −1 point if the avatar is killed, or it falls into the zombie spawn point. **Actions:** LEFT, RIGHT, UP, DOWN.
Zelda	The avatar must find a key in a maze to open a door and exit. The player is also equipped with a sword to kill enemies existing in the maze. The player wins if it exits the maze, and loses if it is hit by an enemy. **Score:** 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it. −1 point if the avatar is killed. **Actions:** LEFT, RIGHT, UP, DOWN, USE.

Table 3.1: Games in the training set of the GVGAI Competition.

The framework used in the competition (Perez et al., 2014d), and in this research, is a Java port of the original Python version of VGDL, originally developed by Tom Schaul.

Figure 3.8: *Four of the ten games from the GVGAI training set. From top to bottom, left to right, Boulderdash, Survive Zombies, Aliens and Frogs.*

Each controller in this framework must implement two methods, a constructor for initializing the agent (only called once), and an `act` function to determine the action to take at every game step. The real-time constraints of the framework determine that the first call must be completed in 1 second, while every `act` call must return a move to make within a time budget of 40 milliseconds (or the agent will be disqualified). Both methods receive a timer, as a reference, to know when the call is due to end, and a `StateObservation` object representing the current state of the game.

The state observation object is the window the agent has to the environment. This state can be *advanced* with a given action, allowing simulated moves by the agent. As the games are generally stochastic, it is the responsibility of the agent to determine how to trust the results of the simulated moves.

The `StateObservation` object also provides information about the state of the game, such as the current time step, score, or whether the player won or lost the game (or is still ongoing). Additionally, it provides the list of actions available in the game that is being played. Finally, two more involved pieces of information are available to the agent:

- A history of avatar events, sorted by time step, that have happened in the game so far. An avatar event is defined as a collision between the avatar, or any sprite produced by the avatar (such as bullets, shovel or sword), and any other sprite in the game.

- Lists of observations and distances to the sprites in the game. One list is given for each sprite type, and they are grouped by the sprite's *category*. This category is determined by the *apparent* behaviour of the sprite: static, non-static, NPCs, collectables and portals (doors).

Although it might seem that the latter gives too much information to the agent, the controller still needs to figure out how these different sprites affect the game. For instance, no information is given as to whether the NPCs are friendly or dangerous. The agent does not know if reaching the portals would make him win the game, would kill the avatar, or simply teleport it to a different location within the same level.

Part II

Adaptive Controllers for Real-Time Games

" With these ridiculous gelatinous orbs in my skull! With eyes designed to perceive only a tiny fraction of the EM spectrum. With ears designed only to hear vibrations in the air."
- Brother Cavil, Battlestar Galactica

Chapter 4

Heuristics for Online Learning

This chapter analyzes how introducing heuristics to bias the Monte Carlo Tree Search play-outs improves results in the 2005 Physical Travelling Salesman Problem. Most of the material in this chapter has been published in the paper (Perez, Rohlfshagen, and Lucas, 2012a).

4.1 Introduction

As described at the beginning of this thesis, real-time games pose several challenges to creating agents that play them. The 2005 Physical Travelling Salesman Problem (PTSP-2005, see Section 3.1), the game employed in this initial study, is an open-ended real-time game that features just such hazards, like the limited time budget the agent has to decide the next move to make, and the fact that end-game scenarios are rarely seen within the algorithm's simulations. The consequence of this is that MCTS, in its vanilla form, it is not able to know if a given ply will lead to a win or a loss in the game.

The research described in this chapter studies the influence of domain knowledge to bias the action decision process in Monte Carlo Tree Search (MCTS), both in the tree and in the default policy. Additionally, different budget times (from 10 to 50 milliseconds per action) are employed in the experiments, with the objective of comparing how this factor affects the quality of the solutions found.

Biasing MCTS play-outs is not new in this field, as other authors have attempted similar techniques in the past for real-time and turn-based games. For instance, a game where this technique has been applied is SameGame, a puzzle played in a 15×15 board where the player has to remove groups of tiles of the same colour. The objective is to clear the board in the least number of moves possible. Matsumoto et al. (Matsumoto, Hirosue, Itonaga, Yokoo, and Futahashi, 2010), who incorporated do-

main knowledge to guide the MC roll-outs, obtained better results with a little more computational effort. Chaslot et al. (Chaslot, Winands, van den Herik, Uiterwijk, and Bouzy, 2007), and Winands and Björnsson (Winands and Björnsson, 2010) incorporate an heuristic value on the UCB1 formula (Equation 2.1) to introduce a *progressive* bias, in such a way that the influence of this term decreases as the number of visits of the node augments.

Guiding play-outs has also been a focus of study in real-time games, such as Tron and Ms. Pac-man, two games that have been objects of study in the past years. Samothrakis et al. (Samothrakis, Robles, and Lucas, 2010) applied a standard implementation of MCTS to Tron, including knowledge to avoid self-entrapment in the MC roll-outs. The authors found that, although MCTS works well, a significant number of random roll-outs produce meaningless outcomes due to ineffective play. Den Teuling (Teuling and Winands, 2012) applied UCT to Tron with some modifications, such as progressive bias, simultaneous moves, game-specific simulation policies and heuristics to predict the score of the game without running a complete simulation. The enhancements proposed produce better results only in certain situations, depending on the board layout.

The objective of Ms. PacMan is to control the player to clear the maze by eating all pills, without being captured by the ghosts. This game, similar to the PTSP-2005, is open-ended, with game-end situations far ahead in the future and very rarely devised by the algorithm during its iterations. Robles et al. (Robles and Lucas, 2009) expand a tree with the possible moves that Ms. Pac-Man can perform, evaluating the best moves with hand-coded heuristics, and a flat MC approach for the end game prediction. Samothrakis et al. (Samothrakis, Robles, and Lucas, 2011) used MCTS with a 5-player max^n game tree, where each ghost is treated as an individual player. The authors show how domain knowledge produces smaller trees and more accurate predictions during the simulations. Also, Ikehata et al. (Ikehata and Ito, 2011) propose a version of MCTS that identifies dangerous moves and zones in the map where the player is likely to be captured.

This chapter is structured as follows: first, Section 4.2 describes an initial study on the performance of Monte Carlo search and MCTS on this game. It compares the vanilla form of these algorithms with other versions that use a simple heuristic, and analyzes the main hazards found. A more involved heuristic is presented in Section 4.3, providing a deeper experimental study, using a different number of cities and time budget for the problem. Finally, Section 4.4 draws some conclusions.

4.2 Preliminary Experimental Study

One of the building blocks for the application of MCTS to games is the definition of a state. In a game like the PTSP-2005, each state can be almost uniquely described by the position of the point-mass, its velocity, and the minimum distance ever obtained to all cities. Another aspect that needs to be defined is the set of actions available for the controller. In this game, the actions are identical across all states, and correspond to the game's actions: forward, backward, left, right and neutral.

Finally, as most of the states reached at the end of the roll-outs will not be end-game scenarios, it is necessary to declare a value function that indicates the quality of each state. To obtain this score function, a set of values v_i is calculated for each city i as shown in Equation 4.1:

$$v_i = \begin{cases} 0 & \text{if } d_i < c \\ f_m - \frac{f_m}{d_i - (c-2)} & \text{otherwise} \end{cases} \tag{4.1}$$

Here, d_i represents the minimum distance between the point-mass and the city i, c is the radius of each city and f_m is the maximum value estimated for the fitness. This equation forces the algorithm to weight more heavily those positions in the map that are very close to the cities.

The final value function $V = P + T + \sum_{i=1}^{N} v_i$ is calculated as the summation of the values v_i, where N is the number of cities in the map, and two more factors: P and T. P is a penalty set to a high positive value if the agent is travelling outside the boundaries of the map. T is the number of time steps played so far in the game, and it is set to a high value if it exceeds certain number of game cycles, to heavily penalize unnecessary long runs.

The score V associated with each state is normalised: the maximum fitness is equivalent to the number of cities multiplied by f_m, plus the penalties for travelling outside the boundaries of the map and the number of steps performed so far. The normalisation is very important to identify useful values of C in the UCB1 Equation 2.1, which has been set to 0.25 in this study following systematic trial and error.

Two default policies have been tested. The first one uses uniform random action selection (hence, roll-outs are not biased in this setting). The second one, `DriveHeuristic`, includes domain knowledge to bias move selection: it penalises actions that do not take the agent closer to any of the unvisited cities. This implies that actions which minimise the fitness value are more likely to be selected.

Algorithm	Average	Not solved	Average simulations
1-ply Monte Carlo Search	539.65 (3.29)	1	816
Heuristic Monte Carlo	532.63 (3.30)	0	726
MCTS UCB1	531.25 (3.52)	2	878
Heuristic MCTS UCB1	528.77 (3.67)	0	652

Table 4.1: Results for 10 city maps compared in terms of the number of time steps required to solve the problem. *Not solved* shows the number of trials where the algorithm was unable to find a solution. Finally, *Average simulations* indicates how many MC simulations were performed, on average, in each execution step.

Four algorithms are considered in this preliminary experiment:

- **1-ply Monte Carlo Search**: uniformly random simulations are run from the current state, without storing any tree. The action with the highest expected reward is selected when the time budget is consumed.
- **Heuristic Monte Carlo**: this is a slight modification of the previous one, biasing the simulations using the `DriveHeuristic`.
- **MCTS UCB1**: the first MCTS variant, using UCB1 as its tree policy and uniform random rollouts.
- **Heuristic MCTS UCB1**: the second MCTS variant, that uses UCB1 as its tree policy, but biases the rollouts using the `DriveHeuristic`.

To compare the different configurations, the following experiments have been carried out on 30 different maps that have been constructed uniformly at random. A minimum distance between cities prevents overlap. The same set of 30 maps has been used for all experiments and configurations were tested over a total of 300 runs (10 runs per map). Finally, the time to make a decision at each stage of the problem has been set to 10 milliseconds.

The results are shown in Table 4.1. It is evident that MCTS outperforms, with respect to the number of best solutions found, both 1-ply MC Search and MCTS with a heuristic in the roll-outs. The differences in the average scores are, however, insignificant. The observation that 1-ply MC Search is achieving similar results to MCTS suggests that the information obtained by the roll-outs is either not utilised efficiently or is simply not informative enough. If this is the case, the action selection and/or tree selection mechanism cannot be effective.

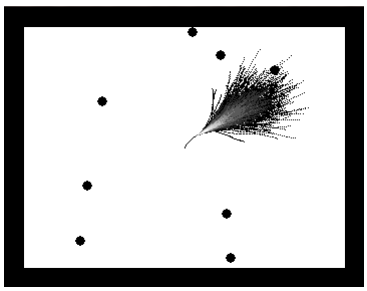


Figure 4.1: *Tree exploration at the start.*

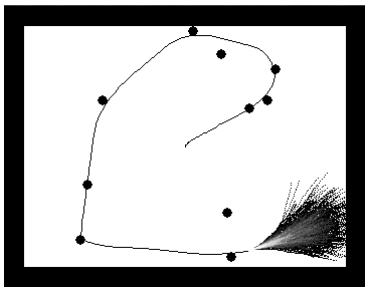


Figure 4.2: *No cities close to tree exploration*

Figures 4.1 and 4.2 depict a visualisation of the MCTS tree at a particular iteration: each position, represented in the figure as a pixel on the map, is drawn using a grey scale that represents how often this position has been occupied by the point-mass during the MC simulations. Lighter colours indicate more presence of those positions in the simulations, while darker ones represent positions less utilised. Figure 4.2 is of special interest: the simulations are taking the point-mass to positions where no cities can be found. This happens because no other portion of the map (where cities are located) is explored and thus MCTS is unable to steer the point-mass towards regions of higher fitness.

4.3 Extended Experimental Study

The objective of the extended experimental study is to analyse the impact of additional domain knowledge on the performances of the algorithms. In particular, it exploits the concept of a *centroid*, calculated as the centre of all unvisited cities and the centre of the map. The idea of the new heuristic is to detect and prevent the selection of actions that take the point-mass away from that point. Therefore, especially in the tree policy, this can be considered as a **pruning heuristic**, as some of the branches of the tree will not be explored.

Pruning branches of the tree with domain knowledge is something that has been used before in the literature, albeit not for real-time games. For instance, Huang (Huang, Liu, Lu, and Xiao, 2010) recently used the concept of territory in the game of Go to improve the performance of the program *LinGo* against *GNU Go 3.8*. Arneson et al. (Arneson, Hayward, and Henderson, 2010) used domain knowledge to prune branches in their world champion Hex program *MoHex*, showing that this enhancement was roughly equivalent to doubling the number of simulations.

In PTSP-2005, however, pruning as defined here cannot be used all the time, because otherwise the point-mass would always be drawn towards the centroid, not visiting any cities. In order to decide

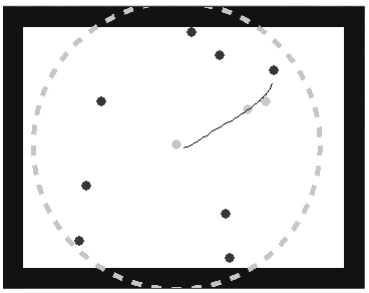


Figure 4.3: *Centroid and influence.*

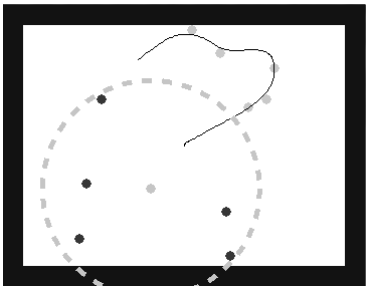


Figure 4.4: *Centroid and influence update.*

when this heuristic is to be enabled, a circle with a certain radius r is defined, centred on the centroid, to be the centroid's influence.

The maps depicted in Figures 4.3 and 4.4 show a set of cities, the centroid (located near the center of the map) and the centroid's area of influence. The value of r used is the distance to the farthest city from the centroid, multiplied by a factor e; the value of e can be used to modulate how far the point-mass is allowed to go from the centroid. In this study the value is set to 1.05, determined empirically.

The following new algorithms are defined, using the `CentroidHeuristic`:

- **Centroid Heuristic Monte Carlo**: uniformly random simulations are run from the current state, without storing any tree. The `CentroidHeuristic` is used to guide the MC simulations, ignoring actions that do not take the point-mass towards the centroid (if within the centroid influence).
- **Centroid Heuristic MCTS**: MCTS algorithm that uses UCB1 as tree policy, and the *CentroidHeuristic* during the default policy.
- **Centroid MCTS only UCT**: MCTS that uses UCB1 as tree policy, but only considering those actions that are not discarded by the *CentroidHeuristic*. The default policy uses uniformly random action selection.
- **Centroid MCTS & UCT**: both tree and default policies are biased by the *CentroidHeuristic*.

4.3.1 Random Maps of 10 Cities

The results for random maps of 10 cities are shown in Table 4.2. It is evident that solution quality was improved by the `CentroidHeuristic`. The average solution quality, using the centroid heuristic for both the tree selection and MC roll-outs, is 481.85, with a low standard error and a very good

Algorithm	Average	Not solved	Average simulations
1-ply Monte Carlo Search	539.65 (3.29)	1	816
Heuristic Monte Carlo	532.63 (3.30)	0	726
MCTS UCB1	531.25 (3.52)	2	878
Heuristic MCTS UCB1	528.77 (3.67)	0	652
Centroid Heuristic MC	552.87 (4.16)	0	915
Centroid Heuristic MCTS	524.13 (3.44)	0	854
Centroid MCTS only UCT	599.38 (10.68)	76	1009
Centroid MCTS & UCT	**481.85** (**6.52**)	0	659

Table 4.2: 10 city result comparison, 10ms limit.

count of best solutions found: both *Centroid MCTS only UCT* and *Centroid MCTS & UCT*, with $C = 0.05$, achieve more than the 50% of the best scores. The Kolmogorov-Smirnov test confirms that these results are significant. The test provides a *p-value* of 1.98×10^{-22} when comparing *1-ply MC Search* and *Centroid MCTS only UCT*, and 1.98×10^{-22} for *1-ply Monte Carlo Search* against *Centroid MCTS & UCT*.

Similar results have been obtained for time steps of 50 milliseconds. In this case, the *1-ply Monte Carlo Search* algorithm achieves an average time of 514.31, while *MCTS UCB1*, *Centroid MCTS only UCT* and *Centroid MCTS & UCT* obtain 511.87, 556.53 and 469.72 respectively. Several things are worth noting from these results: first, the algorithms perform better when the time for simulations is increased. This was already anticipated in the description of MCTS in Section 2.1. Second, it is interesting to see how the different MCTS configurations (specially *MCTS UCB1* and *Centroid MCTS only UCT*) improve more than the Monte Carlo techniques when going from 10 to 50 ms.

The third MCTS configuration (*Centroid MCTS & UCT*), which obtains the best results for both time limits, does not improve its solution quality as much as the other algorithms when increasing the simulation time. However, it is important to note that the results obtained by this configuration given 10 ms are better than the best solution found by any other algorithm given 50 ms. It is highly significant that the solutions obtained by this algorithm are the best ones found for this problem, showing an impressive performance even when the available time is very limited. This makes the approach very suitable for time-constrained real-time games.

Algorithm	Average	Not solved	Average simulations
1-ply Monte Carlo Search	1057.01 (6.45)	0	562
Heuristic Monte Carlo	1133.10 (6.58)	11	319
MCTS UCB1	1049.16 (6.24)	0	501
Heuristic MCTS UCB1	1105.46 (5.73)	3	302
Centroid Heuristic MC	1119.93 (6.86)	1	441
Centroid Heuristic MCTS	1078.51 (5.97)	0	428
Centroid MCTS only UCT	**1032.94** (**6.36**)	14	481
Centroid MCTS & UCT	1070.86 (6.68)	0	418

Table 4.3: 30 city result comparison, 10ms limit.

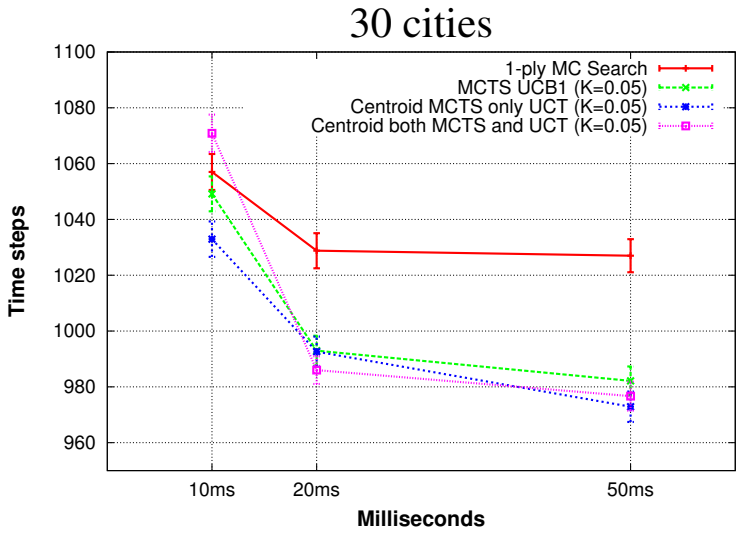


Figure 4.5: *Performance of the algorithms when time limit changes (30 cities).*

4.3.2 Random Maps of 30 Cities

To check if the results of the previous section are consistent, some experiments were performed with 30 cities. Table 4.3 shows the results of these algorithms for a time limit of 10 ms. The results are similar to the ones recorded in the 10 cities experiments, although in this case *Centroid MCTS only UCT* with $C = 0.05$ is the algorithm that solves the problem in the least number of time steps. Figure 4.5 shows the performance of some of the configurations tested for the different time limits considered. Comparing *1-ply Monte Carlo Search* with the *Centroid MCTS only UCT* using the Kolmogorov-Smirnov test gives the following p-values for 10 ms, 20 ms and 50 ms respectively: 0.349, 0.0005 and 1.831×10^{-7}. This confirms significance in the case of 20 ms and 50 ms, but not in the case of 10 ms.

4.4 Conclusions

The experimental study outlined in this chapter focuses on the impact of domain knowledge on the performance of the algorithms investigated and highlights how a good heuristic can significantly impact the success rate of an algorithm when the time to select a move is very limited.

The results show that the *CentroidHeuristic* helps the algorithm to find better solutions, especially when the time allowed is very small (10 ms). As shown in the results, when the time limit is 10 ms, some approaches, like MCTS without domain knowledge, are not able to provide significantly better results than 1-ply Monte Carlo Search. They are, however, able to produce superior results when the time limit is increased.

The main contribution of the research described in this section is evidence to show that it is possible to effectively utilise simple domain knowledge to produce acceptable solutions, even when the time to compute the next move is heavily constrained. It is also worthwhile mentioning that the design of an appropriate value function, especially for games like the PTSP where the real-time constraints usually prevent the algorithm from reaching an end-game state, is crucial in order to obtain good results.

The off-line version of the PTSP-2005 has also been solved with evolutionary algorithms, notably in the GECCO 2005 PTSP Competition. In fact, the winner of that competition utilised a genetic algorithm, using a string with the five available forces as a genome for the individuals[1]. This solution format, a string of forces, is a suitable representation for evolutionary algorithms that may be applied to this problem. Chapter 7 compares MCTS and an evolutionary approach that uses this representation, providing some interesting conclusions.

The following chapters extend the problem analyzed here and propose further improvements. To start, a more complex game is used (PTSP, as defined in Section 3.2), with more interesting maps (by introducing obstacles) and modified game-physics to steer a vehicle rather than a point-mass.

Finally, a particular challenge faced by MCTS when this algorithm is applied to PTSP-2005 is the lack of a long-term plan that considers the order in which to visit the cities, together with the short-term plan of how best to steer to the next city or two. The short budget time the algorithm has to decide the next move increases the difficulty of creating long-term plans in this scenario. However, it is reasonable to think that a high level long-term plan, independent from the action decision process at

[1]Results and algorithms employed can be found at `cswww.essex.ac.uk/staff/sml/gecco/ptsp/Results.html`

each step, would help improve the performance of the algorithm. This concept is explored in the next chapter.

"We designed you to be as human as possible."

- Ellen Tigh, Battlestar Galactica

Chapter 5

Planning in Real-Time Games

This chapter introduces the concept of long-term versus short-term planning. Most of the material in this chapter has been published in (Perez, Rohlfshagen, and Lucas, 2012b).

5.1 Introduction

Simulation-based approaches like Monte Carlo Tree Search (MCTS) struggle with real-time games as end-game situations are rarely seen during the simulated moves. Concretely, if the game is long (i.e. the amount of simulated steps in a game cycle is orders of magnitude smaller than the typical number of game steps a game has), the percentage of game-end states the algorithm is able to explore can be really small.

Examples of games where this happens are real-time strategy games (RTS), where the controller typically has to balance micro and macro-management: control of units versus planning. In (Ontañón, Synnaeve, Uriarte, Richoux, Churchill, and Preuss, 2013), S. Ontañón et al. present a survey of research works in the RTS game *Starcraft*, detailing the different approaches attempted in this game regarding long-term (strategy) and short-term planning (tactics and reactive control).

Weber et al. (Weber, Mateas, and Jhala, 2011) proposed the use of case-based planning to determine the strategy of a Starcraft bot, while unit management was performed by hand introducing domain knowledge heuristics, that resulted in a controller of a quality similar to amateur human players.

Another RTS game that has been an object of study is Open RTS (ORTS). In this benchmark, Naveed et al. (Naveed, Kitchin, and Crampton, 2010) applied Upper Confidence Bounds for Trees (UCT) and Rapidly-exploring Random Trees (RRT) for solving only a specific task within the com-

plexity of the game: finding paths between points in the game scenario. The authors concluded that UCT finds solutions with less search effort, but that the RRT agent obtains better results in terms of playing strength.

Whereas, in the previous study, the authors used tree search techniques to perform a micro-management task (moving units), M. Chung et al. (Chung, Buro, and Schaeffer, 2005) employed Monte Carlo planning to devise the only strategy of the player. The authors use a 'capture the flag' type of game within ORTS, with states that are not fully observable, and employing abstractions to simulate the outcomes of the moves made.

This chapter of the thesis examines how the division between long-term and short-term planning affects the quality of solutions found in the PTSP game (2012 version, see Section 3.2), as introduced at the end of the previous chapter. Section 5.2 analyzes this concept in detail when applying MCTS to the PTSP. Later, Section 5.3 presents an MCTS based driver, and Section 5.4 describes different long-term plans that employ this driver. Section 5.5 draws some final conclusions and determines the next steps to take.

5.2 MCTS for the PTSP

Standard MCTS cannot be applied directly to the PTSP: the open-endedness of the game would prevent any of the Monte Carlo (MC) simulations from reaching a terminal state. It is thus necessary to limit the number of moves (depth) of the default policy. This, however, poses another issue: although one always knows, at any stage of the game, the number of waypoints that were visited up to that stage (and hence one has a perceived notion of quality), this value is typically not fine-grained enough to allow MCTS to distinguish between different rollouts. Many rollouts will not manage to reach an additional waypoint and hence will result in identical scores, making the tree search somewhat akin to the needle-in-a-haystack problem. This issue may be addressed using a more fine-grained value function that assigns distinguishable values to the states encountered by the MC simulations.

The second obstacle that needs to be overcome is the real-time aspect of the game: the number of MC simulations needs to be reduced in order to avoid the algorithm running out of time. It is thus necessary to find the perfect balance between the length of the roll-outs (i.e., the number of moves per MC simulation) and the number of roll-outs achievable. If the roll-outs are too short it is not possible

to obtain enough information to connect a subset of the waypoints. On the other hand, if the MC simulations are too long, only a few roll-outs can be performed at each time step and the uncertainty associated with the action to take is too significant. It is possible to reduce this variance by using domain-specific knowledge to bias the roll-outs, albeit at an additional speed reduction.

A central issue is thus the amount of information that can be extracted from the roll-outs. There are two main approaches that can be taken. On one hand, it is possible to concentrate the MC simulations on the navigation part in order to get a very fast driver, making use of an ordering of cities obtained *a priori* by a higher level solver. On the other hand, heuristic information can be included in the roll-outs to produce longer and more useful simulations that entail both short-term and long-term planning. The work performed in this chapter concentrates on the first approach, whereas Chapter 7 focuses on the second one. In the present chapter, the controller makes use of the 1 second allowed for initialisation time to calculate the order of waypoints to visit: the objective is to obtain a very good driver (short-term look ahead) that is directed by a long-term planner (TSP solver) which has to indicate the order of waypoints to follow.

Since the route connecting all waypoints has to be calculated only once, the remaining problem is reduced in complexity. However, this reduction in complexity comes at the cost of accuracy: since the route and driving are interdependent, any approach that tackles them independently might be sub-optimal. However, it is possible to take the physics of the game (and the ability of the algorithm) into account in various ways when searching for the order of waypoints. Different approaches are investigated in Section 5.4.

Furthermore, there are two more parameters of the MCTS algorithm that will be tested: keeping or discarding the tree and saving the best route. The former one, keeping the tree instead of discarding it, is used in order to avoid losing the statistics gathered in previous execution steps. This process is performed by selecting the appropriate child of the root node (depending on the action taken in the last step) and making it the new root. Then, the rest of the tree is discarded (with the exception of the sub-tree under the new root) and a new MCTS iteration starts. Nevertheless, in many games, the whole tree is discarded at every execution step, as the tree can be too large due to a high branching factor.

The latter parameter consists of saving the best route found during the simulations. As the PTSP

is a deterministic single-player game, it is possible to keep the best route found and to repeatedly return the actions corresponding to said route until a better one has been found. In two (or more) player games, this is not so useful as the uncertainty of the moves of the opponent must be taken into account. Therefore, two different move selectors are compared here: apply the action of the best route found during the MC simulations or take the action that leads to the best estimated value (standard approach).

5.3 The Baseline Driver

First, an MCTS baseline driver is established that is able to finish the game in most cases. This driver is subsequently used to analyse the differences between short and long-term planning. All experiments have been performed on a set of 20 maps with 5 trials each (i.e., 100 games in total) in order to get some meaningful statistics. The experiments are executed on the competition server under the 2012 PTSP competition's formal rules (see (Perez, Rohlfshagen, and Lucas, 2012c)), allowing a comparison of the results with those from other participants.

MCTS is used with UCB1 (see Equation 2.1) as a tree policy, a uniformly random default policy and a roll-out depth set to 100 actions. The tree is discarded after every execution step, and the best route is not saved. It is important to have a reliable value function for non-terminal states; the objective of the algorithm is to `maximize` this value function, which is normalised between 0 and 1. Equation $s = w + d + c + t$ is used, where each one of the terms corresponds to the following features of the game state:

- $w = w_n * f$, where w_n is the number of waypoints visited and f a constant ($f = 1000$).
- $d = max(900 - d_e, 0)$, where d_e is the Euclidean distance between the ship's position and the closest waypoint in the map.
- $c = -10c_n$, where c_n is the number of collisions of the ship during the simulation.
- $t = 10000 - T$, where T is the time spent in the game (10000 is the maximum duration of a game: the number of waypoints multiplied by the maximum number of steps per waypoint).

This approach always drives towards the nearest waypoint until it has been collected. There is no long-term planning that may allow the algorithm to approach a particular waypoint in such a way that the next waypoint after that may be reached in minimum time.

Given the specifications above, the algorithms managed to visit 6.59 (0.32) waypoints on average, taking 2172 (76.61) time steps on average. The performance of a controller on the PTSP is primarily evaluated by the number of waypoints visited (i.e., 65.9% in this case). The time taken to do so is taken into account as a secondary criterion. In order to improve the number of waypoints visited, some changes are required to the basic setup.

The first modification regards the calculation of the distance d to the closest waypoint: since there are obstacles in the map, a straight line (Euclidean) distance may not be able to provide a reliable estimate of the true cost of reaching a waypoint (and may cause the ship to get stuck against a wall). In the new configuration, the way to determine the closest waypoint to the ship is still the one with the smaller Euclidean distance. However, the value d_e is instead calculated as the cost of the A* path (using the game's built-in graph) from the ship to the waypoint. Choosing the closest waypoint with the Euclidean distance is an approximation that has been observed to provide better results for efficiency reasons: as it is a faster calculation, more MC simulations are able to be executed at each step.

The second modification was introduced after analyzing the behaviour of the driver. In some cases, the ship got stuck against a wall and none of the MC simulations were able to discover an escape route. Hence an `unstuck` mechanism was implemented. This mechanism rotates the ship, once it has been in the same position for more than a specific amount of time, so it faces in the direction of the next point in the path leading to the closest waypoint. Concretely, this is done firstly identifying the angle the ship should be facing so its orientation is aligned with the direction that leads to the closest waypoint, given by its A* path. Then, the actions left or right (depending on which one reduces the angle between the current and the target orientation) are applied, with no thrust pressed. This process continues until the orientation of the ship is the desired one, which finalizes the unstuck mechanism.

These two modifications raise the number of waypoints visited on average to 94.6%: the number of waypoints visited is now 9.46 (0.18) in an average time taken of 2387 (86.38). It is important to highlight that there is a direct correlation between the time spent in the game and the number of waypoints collected: although the second experiment resulted in longer execution times, more waypoints were

visited and hence it is impossible to draw conclusions regarding the actual speed of the ship. The only fair comparison, attending only to the time spent, must be made between algorithms that obtain similar amounts of waypoints visited. However, an approximation can be obtained by observing the time spent per waypoint (time spent divided by number of waypoints visited). In this case, these algorithms would obtain respective values of 329 and 252 time steps per waypoint, showing that the algorithm with the modifications seems to be faster.

5.4 Long term versus short term planning

This section analyzes the different plans that can be used for the driver described above to tackle the PTSP in a more effective manner.

5.4.1 Short-term Planning: Driving without a Route

It is possible to further improve the results of the `Baseline Driver` by systematically tuning the algorithm's parameters. The goal is to find the best possible MCTS controller, given no long-term planning at all. The order of cities is determined entirely by the (greedy) selection of the nearest unvisited waypoint at all times.

The following parameters have been tested:

- Two values for C (see equation 2.1) are compared: 0.025 and $\sqrt{2}$.
- MC simulation depth: 50, 100 and 200 time steps.
- Discarding or keeping the tree after each execution step.
- Saving and applying the actions that take to the best route found, or taking the actions that lead to the best estimated value.

The results of this comparison are shown in Table 5.1. Interestingly, much shorter rollouts tend to lead to overall better performances. In particular, the variants that use a depth of 50 for the MC simulations (specially the top two rows) collect almost 100% of the waypoints, with less than 2000 time steps on average (i.e., shorter rollouts also seem to improve speed). On the other hand, variants that

MCTS Parameters				Waypoints	Time Spent
C value	Roll-out depth	Keep tree	Follow best route	Average	Average
0.025	50	No	No	**9.99** (**0.01**)	**1955.9** (**53.67**)
$\sqrt{2}$	50	No	No	**9.98** (**0.01**)	**1941.57** (**54.84**)
0.025	100	Yes	Yes	9.57 (0.13)	2737.24 (85.93)
$\sqrt{2}$	100	No	No	9.52 (0.17)	2376.81 (84.57)
0.025	100	No	No	9.46 (0.18)	2386.76 (86.38)
0.025	100	No	Yes	9.42 (0.18)	2606.64 (87.72)
0.025	50	Yes	No	9.4 (0.18)	1965.97 (57.51)
0.025	100	Yes	No	8.93 (0.24)	2308.33 (82.07)
0.025	200	Yes	Yes	8.57 (0.26)	2977.58 (88.50)
0.025	200	No	Yes	8.46 (0.27)	2925.74 (101.52)
0.025	50	Yes	Yes	8.2 (0.27)	3282.57 (95.99)
0.025	200	Yes	No	8.06 (0.28)	3024.54 (104.98)
0.025	200	No	No	8.05 (0.27)	3049.67 (98.63)
0.025	50	No	Yes	7.92 (0.32)	3112.79 (95.51)
$\sqrt{2}$	200	No	No	7.64 (0.29)	2770.56 (90.71)

Table 5.1: MCTS, Short-term planning, ordered by average of waypoints visited.

use much deeper roll-outs produce significantly worse results and collect less than 86% of waypoints, requiring in excess of 3000 time steps to do so.

The value of C does not have a big impact on the results. $C = \sqrt{2}$ produces only slightly faster drivers and, being a value widely used in the literature, it will be used in the next experiments.

The remaining two parameters analysed, keeping the tree and following the best route, have a somewhat negative impact on the overall performance of the algorithm. The equivalent of the leading configuration (i.e., row one in Table 5.1) that keeps the tree ranks only 7th. Similarly, following the best route found during the simulations increases the time taken to visit the waypoints.

A possible explanation for this may rely on the fact that the ship overshoots a waypoint because of travelling at a very high speed. For instance, the best route could have a high score value because the simulation ended very close to a waypoint. However, the speed of the ship at that point is probably quite high and if the distance to the waypoint is very small, it is likely that the sequence of moves taken during the roll-out of the best route involves a lot of acceleration. If the speed is too high, the ship might not be able to visit the waypoint when the MC simulations actually reach it in their next steps. On the contrary, more conservative sequences of actions have more possibilities to change the trajectory of the ship in further steps, and eventually visit the waypoint.

5.4.2 Long-term Planning: Pre-computing the Route

The second part of this empirical study focuses on approaches that make use of pre-computed routes: instead of aiming at the nearest waypoint at the time, the controller now makes use of an *a priori* ordering of cities. This may be established during the initialisation time of the controller before the game starts. This problem essentially corresponds to the classical travelling salesman problem although, as will be shown later, it is possible to take the physics of the game into account to obtain routes that are somewhat longer but more efficient to drive. The method chosen to solve the TSP is the Branch and Bound (B&B) algorithm (Land and Doig, 1960) which is able to provide the optimal solution for 10 cities within the time given. Two variations of B&B are used, depending on how the distances (cost) between the waypoints are computed:

- B&B with Euclidean cost: the cost of going from waypoint A to waypoint B is the Euclidean distance. This algorithm does not take into account the presence of obstacles and is thus expected to be sub-optimal in many cases.

- B&B with A* path cost: using the pathfinding library of the framework, the path between every pair of waypoints is calculated. The travelling cost from one waypoint to another is the length of the path that joins them. This solution resembles an optimal TSP solution.

The long-term planning feature of the algorithm provides an order of waypoints that the ship has to follow. The low-level navigation to do so is handled by the most promising MCTS algorithm from the previous section: a roll-out depth of 50, discarding the tree each time step, not saving the best route found so far and a value of $C = \sqrt{2}$. The value function is modified to take into account the distance to the next unvisited waypoint in the pre-computed route (rather than the nearest waypoint). Furthermore, a penalty is imposed when the ship accidentally visits a waypoint that is not meant to be collected at that time, in order to follow strictly the established order of cities.

The focus of this set of experiments is to establish the efficiency with which the controller is able to collect all the waypoints and the results are shown in Table 5.2. A first analysis of the results shows that by including any reasonable order of cities (even if sub-optimal), the time spent to solve the problem is reduced significantly (from 1941.57 (54.84) to 1830.25 (54.25)). These results suggest that the addition of long-term planning improves the overall performance significantly. Furthermore, it is evident that

Algorithm	Waypoints Average	Time Spent Average
Short-term planning	9.99 (0.01)	1941.57 (54.84)
B&B (Euclidean)	9.4 (0.22)	1830.25 (54.25)
B&B (A*)	9.84 (0.10)	1744.47 (39.29)

Table 5.2: Results with different TSP solvers.

the routes computed with A* are more efficient than those computed using Euclidean distance, since the latter does not take the presence of obstacles into account.

5.4.3 Long Term Planning: Changes of Direction

The previous experiment ignored the inter-dependency between long-term and short-term planning. In this game, an optimal TSP route is not necessarily an optimal PTSP route. In order to address this issue, a different algorithm has been implemented where the cost associated with travelling from one waypoint to another takes into account the changes of direction required in the ship's trajectory. This approach is based on the premise that the ship can visit a series of waypoints connected in a straight (or almost straight) line much more quickly than a series of waypoints that require many changes in direction (and hence loss of momentum).

The most accurate way of computing the optimal ordering of waypoints would be to actually drive the different routes (or sub-routes) using the same controller that is used when playing the game: the base MCTS driver. This, however, would be prohibitively costly given the short initialisation time the controller has to compute the route in the first place (it would essentially solve the problem offline). Instead, the physics (and driving styles) of the game are approximated by taking the angles into account, producing a route that is likely to be sub-optimal yet superior to routes obtained using the classical TSP approach.

Figure 5.1 depicts a portion of a map that shows the path that a (classical) TSP solver would suggest if it were only distances between nodes to be taken into account. This route requires two significant changes of direction (120 and 90 degrees respectively). A PTSP-optimised route is shown in Figure 5.2: this route takes into account the changes of direction and reduces the number of sharp turns (a single sharp turn of just above 90 degrees is now required).

To obtain PTSP-optimised routes, it is necessary to account for the number of turns in a route by introducing a cost factor when searching the optimal TSP route. The final cost of a path is thus

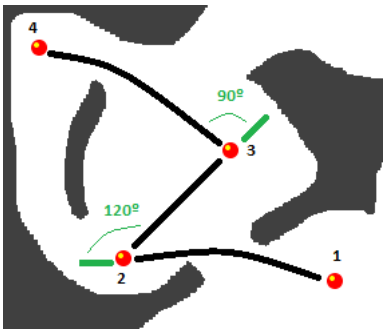


Figure 5.1: *B&B using costs as lengths of the shortest path.*

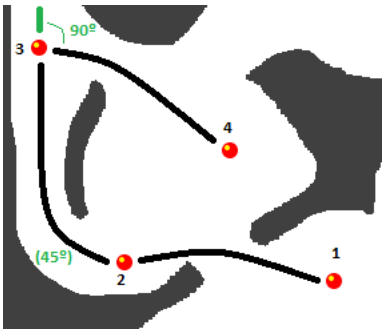


Figure 5.2: *B&B adding changes of direction to costs calculation.*

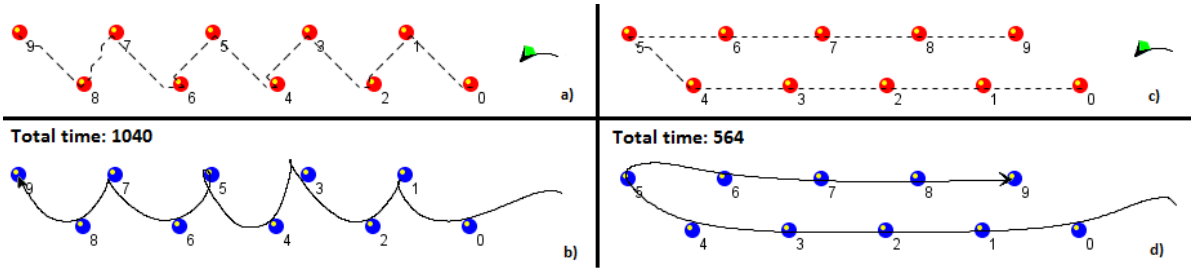


Figure 5.3: *This figure shows the results of following different TSP routes. The top row depicts the order of cities to follow, whilst the bottom row shows an actual game played by the same driver. The left column (figures a and b) presents the problem solved by a TSP that only takes into account the distances between the waypoints. The right column, figures c and d, shows the performance of the controller that includes the changes of direction in the route calculation. As can be seen, the latter algorithm is able to solve the problem in* 564 *steps, while the former can only do it in* 1040*.*

obtained by multiplying the normal length of the A* path by a factor that depends on the summation of angles from the route. To analyse if there is a significant difference after including the changes of direction in the path cost, Figure 5.3 shows a base case. In this test map, it is clear that the order of the waypoints affects the quality of the solution significantly.

The experiment from the previous section is repeated using the new TSP solver that takes the angles of the route into account. It is evident that the performance is slightly better, as the results have been improved: 1703.07 (40.12) against 1744.47 (39.29), although the difference is not particularly pronounced. The reason for the relatively small rate of improvement is that the new TSP solver produced routes that differ in only 9 of the 20 maps. In other words, even accounting for the angles in the route, always driving towards the nearest waypoint results in the same overall order of waypoints in 11 of the 20 maps. If the analysis is concentrated only on those nine maps, the improvement is more noticeable: a run that executes 15 times on each of these nine maps shows that the TSP solver that considers angles outperforms the standard TSP solver, with an average time taken of 1753.98 (39.25) compared to 1820.76 (37.81); it obtains better results in seven out of nine maps.

A final evaluation of comparison includes the best controllers submitted to the PTSP competition: all controllers are evaluated 5 times per map to obtain averages of the number of waypoints and time spent, discarding the two worst games on each map. The controllers are then ranked for each map by the number of waypoints visited and the time required to do so. Points are awarded for each rank as follows: 10 points for the best controller, 8 for the second, 6 for the third, 5 for the fourth and so on until 1 point for the eighth. The results of this comparison are described in Table 5.3, showing eight controllers. These were evaluated in 20 maps, during the first phase of the WCCI PTSP competition, which corresponds to controllers submitted from 1st March to 1st April 2012. The first two controllers are the ones shown in this study: the TSP solver that considers direction changes (`TSP-A`) and the TSP solver that does not (`TSP-N`). Then, the three best participants' controllers are listed: `st3f1` is an heuristic based controller (not using any computational intelligence technique), and both `Purofvio` and `Philster` employ MCTS to drive the ship and a TSP solver for the order of waypoints. An improved version of the `Purofvio` entry compared here was submitted later on, becoming the winner of the competition. Finally, these rankings include the three sample controllers distributed with the software (`greedy`, `lineofsight` and `random` controllers), as described in Section 3.2. Final results and scores of the competition can be seen in Table A.2, in Appendix A.

These results indicate that the algorithms presented here are competitive, as they achieved better results than other approaches during the early stages of the competition. Nevertheless, it is important to mention that this does not imply that these algorithms are the best for this problem, as the other participants kept improving their controllers during the competition, obtaining better performances. This, however, is not the ultimate goal since the focus of this empirical study lies on the relative differences in performance obtained using different variants and combinations of short-term and long-term planning. Nonetheless, it is reassuring to see that the results are good enough to place these algorithms at the top of the rankings. This also demonstrates that taking into account the changes of direction when computing the order of waypoints produces excellent results.

5.5 MCTS for PTSP: Conclusions

This study demonstrates the importance of long-term planning when using Monte Carlo Tree Search (MCTS) for real-time and open-ended (video) games, as exemplified on the Physical Travelling Salesman

Controller	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Tot**
TSP-A	8	**10**	**10**	**10**	**10**	8	**10**	8	**10**	**10**	8	**10**	**10**	**10**	**10**	8	6	8	8	**10**	**182**
TSP-N	**10**	8	8	8	8	**10**	6	**10**	8	8	**10**	8	8	8	8	6	**10**	**10**	**10**	8	170
st3f1	6	5	6	6	5	6	8	6	6	6	4	6	6	5	4	5	5	5	6	6	112
Purofvio	4	6	5	5	6	4	4	5	4	4	5	5	5	6	6	10	8	6	5	4	107
Philster	5	4	4	4	4	5	5	4	5	5	6	4	4	4	5	4	4	4	4	5	89
greedy	3	3	3	3	3	3	3	3	3	2	3	3	3	3	3	3	3	3	2	2	57
lineofsight	2	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	3	3	43
random	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20

Table 5.3: Rankings of controllers following the point award scheme of the PTSP competition.

Problem (PTSP). The real-time element of these games, where the time to compute the next move is severely limited, prevents sample-based algorithms such as MCTS from looking far enough ahead to behave optimally. This issue may be addressed by solving the long-term and short-term aspects of the game separately: the solution proposed in this study shows how providing the order of cities (i.e., solving the long-term planning aspect) makes a noticeable difference in the performance of the algorithm, reducing the time spent solving the problem and still providing a high rate of success in visiting all the waypoints of the maps.

Interestingly, the study performed here brought to the surface what could be considered a poor PTSP map design: in some of the levels, all the effort put into creating a more involved long-term plan responsible for the order of the waypoints to visit was worthless. This is because following the closest waypoint at all times produced similar results. The next chapter explores a possible use of the controllers developed in this chapter as a way to produce more interesting maps for the PTSP.

Following that, Chapter 7 analyzes two other aspects suggested up to this point: the use of evolutionary algorithms for action control in comparison with MCTS, as was first introduced in the conclusions of the previous chapter (see Section 4.4), and the idea of using *macro-actions* to produce longer and more useful simulations.

"I don't want to be human! I want to see gamma rays! I want to hear X-rays! And I want to - I want to smell dark matter! . . . "
- Brother Cavil, Battlestar Galactica

Chapter 6

A use-case: MCTS for PCG

This section shows an interesting use of adaptive controllers for real-time games in the field of Procedural Content Generation. Most of the material in this chapter has been published in the paper (Perez, Togelius, Samothrakis, Rohlfshagen, and Lucas, 2013d).

6.1 Introduction

Procedural content generation in games is a growing research field motivated by a real need within game development, and a research goal to enable new kinds of interactive techniques (Togelius et al., 2011). Techniques developed in the field, especially evolutionary techniques, have been employed elsewhere with great success and content generation has even been the focus of some competitions such as the Mario AI Championship (Shaker, Togelius, Yannakakis, Weber, Shimizu, Hashiyama, Sorenson, Pasquier, Mawhorter, Takahashi, Smith, and Baumgarten, 2011).

The testbed game used is the Physical Travelling Salesman Problem (PTSP, Section 3.2), and it is motivated by the findings discovered in Chapter 5, that showed that some PTSP maps did not reward "wiser" controllers that obtained a more clever route of waypoints. Hence, the goal of this study is to find maps that require or advantage particular solution strategies and controller types. Even more interestingly, these maps could potentially have multiple solutions of the same or very similar quality (this is analogous to showing that multiple strategies of similar effectiveness exist for a strategy game, something which is intuitively considered advantageous).

Concretely, the objective is to produce maps where the optimal order of waypoints is neither the optimal distance-based TSP route nor the nearest-city-first route. In other words, a controller that

makes some computational effort at long-term planning should be able to see an improvement, especially if taking the physics of the game into account, against a controller which works in a simple short-term reactive manner. Such maps are relatively easy to construct by hand if the number of waypoints are few and the map's size is modest. However, it is clear that a manual approach does not scale well. It is also limited if many maps are required, which is the case in the PTSP competitions. Additionally, it is interesting to find several routes that obtain similar high quality results following different waypoint orders, that are distinct from the optimal distance-based and nearest-city-first TSP routes. Designing maps with these features by hand is a very complicated task.

This study presents the application of an evolutionary algorithm, the Covariance Matrix Adaptation - Evolutionary Strategy (CMA-ES, Section 2.2), to automatically generate maps. The proposed approach follows a simulation-based evaluation (described in Section 2.3), as agents are used to play PTSP maps.

Sometimes, procedural content generation needs to tackle the problem of infeasible solutions: while many search/optimization problems differentiate solution quality on a continuum, others feature *constraints*, so that some candidate solutions are not just bad, but *infeasible*. In the PTSP example, if some waypoints cannot be reached, the map is clearly infeasible. Therefore, not only must the content generated be good for the problem at stake, but it also needs to be valid material for the domain in which it is applied. In evolutionary computation, several specialised techniques for handling constraints in optimization have been developed (Michalewicz, 1995). Several different approaches can be discerned, including the "naive" approach of simply giving infeasible solutions a score of zero, as well as more "sophisticated" solutions such as repairing infeasible individuals or keeping separate populations of feasible and infeasible individuals. In search-based procedural content generation, the two-population approach has previously been used for generating game content such as platform game levels (Sorenson, Pasquier, and DiPaola, 2011) and spaceships (Liapis, Yannakakis, and Togelius, 2011; Liapis et al., 2012).

An example of this type of approaches is the Feasible-Infeasible Two Population (FI-2Pop) (Kimbrough, Koehler, Lu, and Wood, 2008) genetic algorithm. In this scenario, two different populations are kept, one for feasible and another for infeasible individuals. Breeding is performed in these two populations separately as normal, and new individuals are moved to the correspondent population according to its feasibility (i.e. if a new individual from the feasible population happens to be infeasible, then it

is introduced in the other population, and vice-versa). The feasible population performs its search as if it was an unconstrained problem, while the infeasible population drifts towards the boundary of the feasible solutions, where the optimum solution often lies (Schoenauer and Michalewicz, 1996).

There exist in the literature other examples of evolutionary methods that tackle the problem of infeasible solutions in continuous numerical optimization problems. For instance, several authors took the approach of adding penalty coefficients to the fitness value, proportional to the level of violation for each constraint of the problem (Homaifar, Qi, and Lai, 1994; Joines and Houck, 1994). Another choice is to assign a fitness for infeasible solutions that is always worse than even the poorest feasible solution, establishing two different ranges of fitness values for each type (Powell and Skolnick, 1993). Finally, a sequential procedure can be taken (Schoenauer and Xanthakis, 1993): the population is initially evolved to minimize the violation of one of the constraints, until a given proportion of the population is feasible for this one. Then, the algorithm moves to the next constraint and repeats the process, eliminating individuals that do not satisfy at least one of the constraints examined so far.

CMA-ES has previously been shown to be extremely effective for continuous optimization (Hansen, 2006). The main reasoning behind choosing this algorithm for this task is that, to the best knowledge of the author of this thesis, CMA-ES has not been applied to procedural content generation before. CMA-ES does not handle constraint handling problems, but it is possible to employ a relatively naive approach to deal with infeasible solutions, as shown in Section 6.2. Additionally, the use of a well-tested off-the-shelf continuous optimization algorithm, rather than a specifically tailored constraint optimization algorithm, carries considerable benefits in terms of ease of experimentation and replicability. Furthermore, the focus of this research is set on the use of adaptive controllers to evaluate the maps generated for the game, rather than in the ES that generates the maps. Whether the use of a constraint handling algorithm would provide better results is left for future research.

This research is presented in this chapter in the following way. First, Section 6.2 details the algorithm used for creating new maps. Then, Section 6.3 explains how the maps generated by CMA-ES are evaluated. The experimental work and results are described in Section 6.4 and some conclusions are drawn in Section 6.5.

Element	Data	Description (Coordinates)
Lines	$(x_o, y_o, x_d, y_d) * L$	Starting and ending positions.
Rectangles	$(x_o, y_o, x_d, y_d) * S$	Top-left and bottom-right corners.
Ship	(x, y)	Ship's starting position.
Waypoints	$(x_w, y_w) * 10$	Waypoints' positions.

Table 6.1: Representation of an individual or map.

6.2 CMA-ES for Procedural Content Generation

A solution to the PTSP is obtained by dividing the problem into two very well differentiated tasks: 1) calculate the order of waypoints; and 2) drive the ship to visit the waypoints in the order specified. In the terminology used here, *controller* must be seen as the combination of the *route planner* (which obtains the order of waypoints) and the *driver* (which effectively applies the actions to move the ship).

The individuals evolved by the algorithm proposed in this study are PTSP maps. These employ a relatively direct representation, that corresponds to the second type of representation discussed in Section 2.3: each one of the maps considered in this study is composed of a set of floating point values that encode the starting position of the ship, the waypoints and the location and size of obstacles such as lines and rectangles. Table 6.1 details the representation of an individual of the population. The order of the rows indicates the order of elements in the string of values.

As can be observed, two different parameters are needed to define the contents and length of an individual: the number of lines (L) and the number of rectangles (R). The experiments described in this study are performed with maps that contain $L = 15$ lines and $R = 8$ rectangles, although initial experimentation shows that it is also possible to evolve maps with different values for these parameters. With these settings, the length of the individual is then 114, and each one of these genes takes a real value in the range $[0.05, 0.95]$. When the genome is read to create a map, these values are scaled to the size of the map, which for this study is established to 500×500 pixels.

An important feature that must be determined for each individual is whether it encodes a feasible map or not, according to the conditions described in Section 3.2.1. Some infeasible maps are able to be repaired. One of the reasons why a map can be invalid is because at least one of the waypoints or the starting position is in the same position as an obstacle, or too close to one. Another possibility is that two of these entities (waypoints and starting position) are too close to each other. If a map is invalid because of one of these reasons, a *repair mechanism* tries to change the location deterministically,

moving one of these entities along the vertical and horizontal axis until a valid position is found. It might be the case that this simple repair procedure is not able to fix the problem in the map, or that another of the problems described in Section 3.2.1 is the cause of its infeasibility (i.e.: unreachable waypoints). In this case, the map is considered invalid and it is flagged as such.

If an individual map happens to be infeasible, the algorithm creates a new randomly initialized map and checks for its feasibility, repeating this process until a feasible one is created. In this way, a population is always composed only of feasible individuals. Therefore, CMA-ES creating infeasible individuals only affects how quickly the experiments are run. Although it would be possible to design a more involved repair mechanism that would reduce the number of rejected individuals, the experiments performed in this research show that the number of infeasible individuals sampled from the multivariate distribution reduces as the algorithm converges towards a solution (with a rate of infeasible individuals less than 5%). This phenomenon is not new, and has been reported previously in the literature (Bouzarkouna, Ding, and Auger, 2010).

The number of generations of each experiment is established at a maximum of 1000 generations, although a fitness-based stopping criterion can finish the run earlier: if the range of the best fitness obtained during the last $10 + (30 \times N/\lambda)$ generations is smaller than 10^{-13}, the experiment stops. For the experiments presented here, N is the problem dimension (114) and λ the population size (100), so this condition must be fulfilled in 44 consecutive generations. This termination criteria, known as *TolHistFun*, is a default stopping condition of CMA-ES, and has been used in the literature before (Auger and Hansen, 2005; Hansen, 2009). The default value of the population size in CMA-ES is $4 + 3 \times log(N)$, which for this problem would be 18. However, the population size used in the experiments presented here is set to 100, a value determined empirically.

6.3 Route planners

The route planner is in charge of determining the order of waypoints that the driver must follow during the game. The route planner considers the cost of travelling from position A to B as the distance of the path given by the graph, which is calculated using the A* algorithm.

Three different variants of route planners have been employed in this study:

- **Nearest-first TSP:** This TSP solver, and the route it produces, is referred to in this study as N_{TSP}. The order of waypoints is obtained by applying the nearest first algorithm to solve the TSP. That is, from the current location, the algorithm sets the closest waypoint as the next waypoint to visit, repeating this procedure until all waypoints are in the plan.

- **Distance TSP:** This TSP solver, and the route it produces, is referred to here as D_{TSP}. The planner uses the Branch and Bound (B&B) algorithm to determine the order of waypoints, using the length of the A* as the cost between each pair of waypoints.

- **Physics TSP:** This TSP solver, and the route it produces, is referred to in this research as P_{TSP}. As in the previous case, the B&B algorithm is employed for the order of waypoints, but in this case the cost between two waypoints is affected by physical conditions such as speed and orientation of the ship.

The *Distance TSP* route planner resembles the optimal TSP path, and it would be the perfect choice if certain physics conditions, such as the ship's inertia, were not present in the game. In contrast, *Physics TSP* has been prepared in order to take into account the nature of the game. As seen in Chapter 5, inertia and navigation should be taken into consideration to calculate the optimal PTSP route. The overall idea is based on the fact that the ship can benefit from visiting waypoints that are in the same straight (or quasi-straight) line, even if the distance between them is not short, as this way the ship maintains its velocity. In other words, minimizing changes of direction within the route - that would cause the ship to lose inertia and speed - is important when computing the route.

In this case, the cost of the path is obtained by an approximation of the time needed to drive the route, as described in Algorithm 4. Given an order of waypoints r, a path p is first obtained where each node is in line of sight with the following in the path (GetInSightPath(*route*) function). This way, the controller would be able to drive between each pair of nodes in a straight line. Starting with an initial speed of 0, the algorithm traverses the path whilst calculating the time taken for the ship to go from one node to the next. This calculation uses the real physics of the game, so the time taken between two nodes is completely accurate, given the selected action sequence, though usually not optimal.

The overall calculation is, however, an approximation, owing to the way the speed is kept between each straight line segment. The dot product of the angle of two consecutive segments is calculated and

Algorithm 4 Route cost estimator.

```
function HeuristicSolver(r)
    p ← GetInSightPath(r)
    speed ← 0
    for all Node n_i in p do
        d ← EuclideanDistance(n_i, n_{i+1})
        t ← TimeToTravel(d, speed)
        dot ← Dot(V(n_i, n_{i+1}), V(n_{i+1}, n_{i+2}))
        pen ← GetPenalization(dot)
        speed ← speed * pen
        totalTime ← totalTime + t
    return totalTime
```

used to decrement the speed at each turn. If this value is 1 (0 degrees), the penalization (*pen*) value is 1 and the speed does not decrease. The penalization increases exponentially with the angle, reaching 0 (complete stop) when the turn to make is of 180 degrees. The final estimated cost of the route is the sum of the time taken to drive all segments of the path given.

6.3.1 Evaluating routes: drivers versus estimations

The driver is the agent that makes the moves in the game, trying to visit all waypoints in the order specified by the route planner. The main problem of evolving maps using drivers to evaluate the different routes is the computational cost it involves. Two different approaches have been taken to evaluate the routes provided by the route planners:

- **Estimated Cost (EC)**: The estimated cost of a route r, $EC(r)$, indicates an upfront value that determines the quality of a route in the map. It is calculated by applying Algorithm 4 on the route given. There is no need for a driver to actually play the game to determine the EC of a route, and the cost is an estimation of the time taken by any driver that follows it.

- **Cost (C)**: The cost of a route r, $C(r)$, is determined by using a driver to complete the game following the order of waypoints provided by the route planner. This value is calculated by actually playing the game, and it takes into account the time spent (*TotalTime*), the remaining waypoints still to be visited when the game finishes (*RemWaypoints*) and a penalty P equal to the maximum time allowed to visit the next waypoint in the route (1000 time steps). Hence, the cost C of driving in a map using a route r is:

$$C(r) = TotalTime + RemWaypoints * P \tag{6.1}$$

In both cases, the smaller the values of EC and C, the better the route, as they represent the time taken to complete the game. The main advantage of using a driver is that the evaluation is reliable in terms of playing the game (one can be certain that the map obtained produces a determined output for the driver used to evolve it), while an estimation is an abstraction that might contain errors and be inaccurate. However, using a specific driver impacts on the time needed to evaluate a route and might lead to maps that fit the navigation style of that particular driver. The estimation, on the other hand, provides a faster indication of the cost of the route, which is not tied to any particular driving style. An interesting trade off that has been employed in this study is to use the estimation for the evolutionary algorithm and, once the run has finished, play the game using a real driver in the maps obtained, in order to verify that the results are conclusive and the maps obtained have the desired properties.

The driver presented in this study is the same MCTS driver implemented after the experiments described in Section 5.4.1 (the second configuration from Table 5.1): roll-out depth of 50, $C = \sqrt{2}$, not keeping the tree nor following the best sequence of actions found.

6.3.2 Evaluating maps: fitness functions with 3 routes

The first objective of this research is to be able to obtain maps where the results obtained with the same driver using distinct routes are different. The quality of a map will be better if it rewards more involved routes than simpler waypoint orders.

Each individual (or map) of the evolutionary algorithm is evaluated measuring the EC values of the different routes that the route planners provide. Let us say that in a given map, the three route planners provide three different routes: N_{TSP}, D_{TSP} and P_{TSP}. Then, the objective is to achieve:

$$EC(N_{TSP}) > EC(D_{TSP}) > EC(P_{TSP})$$

In other words, the estimated cost of using the nearest-first TSP route planner is higher than using the distance TSP route planner, and this cost is also worse than employing the physics TSP route

planner. In order to achieve this, two different fitness functions are employed and defined in this section. Given these routes, it is possible to define the fitness as the result of the following equation[1]:

$$f_3 = -Min(EC_N - EC_D, EC_D - EC_P) \tag{6.2}$$

An analogous fitness can be defined using the real cost (as defined in Equation 6.1) of a driver playing the game as:

$$f'_3 = -Min(C_N - C_D, C_D - C_P) \tag{6.3}$$

CMA-ES is set up to minimize this fitness, so high negative values are better. As this fitness measures the distances between the costs of a naive and a more complex route (both $EC_N - EC_D$ and $EC_D - EC_P$), it must be understood as the amount of time steps saved when using a more involved route instead of a simpler one for solving the problem.

6.3.3 Evaluating maps: fitness functions with 5 routes

Additionally, it is also a purpose of this research to obtain several near-optimal solutions so the best route is not too obvious. In other words, the evolutionary algorithm must be able to generate maps where a group of N routes of the type P_{TSP} (those obtained with the Physics TSP solver) produce a similar performance among them, but all better than a D_{TSP} route, which is still better than an N_{TSP} one.

The P_{TSP} route planner presented in Section 6.3 provides only one route: the best achievable, considering the cost between waypoints, derived from taking the physics of the game into account. However, other routes can be derived from this one applying the operators *2-Opt* and *3-Opt*. These operators exchange 2 (or 3, respectively) nodes in the path to create a new solution. Hence, if M additional routes are needed, the first step is to calculate the best one with Algorithm 4, as usual. Then, *all* possible derivatives from this route are obtained applying *2-Opt* and *3-Opt*. They are sorted by ascending cost and the M best ones of this new group of routes are selected.

The fitness function is similar to the one described in Equation 6.2, substituting the best P_{TSP} route for the i_{th} one in the group of routes. For instance, let us say that the objective is to create maps

[1]For the sake of clarity, $EC_X \equiv EC(X_{TSP})$ and $C_X \equiv C(X_{TSP})$

where 3 routes of type P_{TSP} obtain similar performance when followed by a determined driver. If any of these 3 routes is followed, the performance must be better than following a route of type D_{TSP}, and this should outperform an N_{TSP} route. An initial P_{TSP} route is obtained with Algorithm 4 (i.e. $P_{TSP} \equiv r_0$), and *2-Opt* and *3-Opt* operators are employed to derive all routes from this one. These new routes are sorted by ascending cost (i.e.: r_1, r_2, ..., r_m), being $P_{TSP} \equiv r_0$ better than all these by construction. The fitness function is then defined as follows:

$$f_5 = -Min(EC_N - EC_D, EC_D - EC(r_2)) \tag{6.4}$$

As in the previous section, a fitness for real drivers can be defined such as:

$$f'_5 = -Min(C_N - C_D, C_D - C(r_2)) \tag{6.5}$$

By obtaining maps that maximize the cost difference between these three routes, the algorithm provides individuals where route r_2 is better than route D_{TSP}. As r_0 and r_1 are by definition better than r_2, the resultant maps are "separating" the costs of D_{TSP} and the group (r_0,r_1,r_2).

The experiments described in this study show that it is possible to evolve maps that distinguish between three (as in Equation 6.2) and five (Equation 6.4) routes. Initial experiments have shown that it is also possible to evolve maps with 8 different orders of waypoints, five of them forming the group of P_{TSP} routes r_0 to r_4.

6.4 Results and analysis

This section details the results of the experiments performed during this research. A total of 40 independent runs have been executed for each one of the two batches of experiments (for 3 and 5 routes). The following parameters have been set up:

- CMA-ES with an initial mean m of the MND set to 0.5 and the step size $\sigma = 0.17$ ($\sigma = m/3$, in order for CMA-ES to converge with ± 3 standard deviations). The size of the population, 100, was determined experimentally.
- Three different route planners are employed: N_{TSP}, D_{TSP}, P_{TSP}.

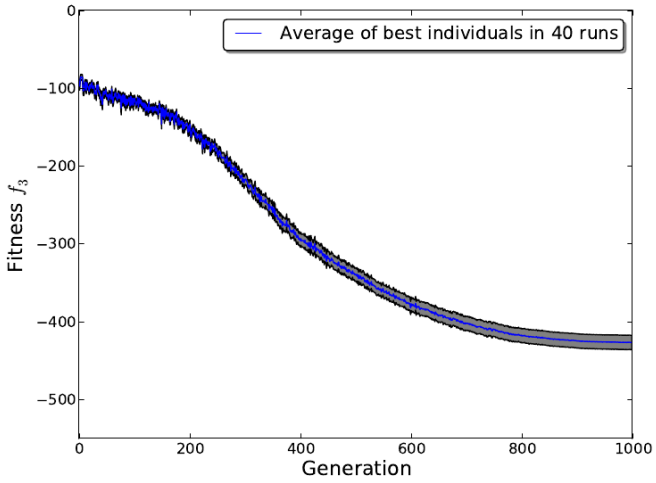


Figure 6.1: *Averaged evolution of the best individuals per generation of the* 40 *runs executed with the heuristic cost estimator for routes, employing* 3 *routes per map. Shadowed area indicates the standard error of the measure.*

- Each route is evaluated using the heuristic cost estimator (Algorithm 4): no driver plays the game to evaluate the routes during evolution.

6.4.1 Heuristic Cost Estimation: 3 routes

For these experiments, the fitness function used is f_3 (Equation 6.2) to evaluate the maps. The three routes used are the ones provided by the three different route planners. Figure 6.1 shows the evolution of the averaged fitness during the experiments run. This picture shows the average (plus standard error), of the best individual of each generation across all experiments run for this setting.

As can be seen, there is a clear evolution in the fitness of the runs. At the beginning of the experiments, the best individuals obtain a fitness close to $f_3 = -100$ (as explained earlier, this represents the minimum difference of cost between each pair of the 3 routes used to evaluate the maps.). By the end of the executions, the average of the best individuals achieve a fitness of $f_3 = -427$ (9.36). Approximately 80% of the runs were stopped by the *TolHistFun* termination criteria, not reaching the maximum number of generations set at 1000.

The MCTS driver has been used to drive the best maps of the final generation of all experiments. The routes taken have been N_{TSP}, D_{TSP} and P_{TSP}, and each one has been played five times (adding up to $3 \times 5 \times 40 = 600$ games played). The average fitness, as described in Equation 6.3, obtained by the MCTS driver in these maps is $f_3' = -321.44$ (24.99). This value cannot be directly compared with

the fitness achieved by the estimated cost heuristic, because they are obtained by a different procedure (playing the game versus not playing it). However, it does show that the maps obtained are sensible: the difference of taking any pair of the given routes is of at least 321.44 (24.99) time steps on average.

A different way to analyze this result is to compute the relative difference of fitness between the routes taken. This is done in the following way: if the time spent by driving route P_{TSP} is taken as a reference, it is possible to calculate the increment of time spent by following routes N_{TSP} and D_{TSP}. Whereas P_{TSP} takes a reference value of 1.0, D_{TSP} obtains a worse performance of 1.43 (0.03), and N_{TSP} spends even more time to obtain a value of 1.692 (0.04).

This analysis is interesting because it provides a more detailed view of the time taken per route. The driver that takes route D_{TSP} spends 43% more time than following P_{TSP}, and taking N_{TSP} spends 69% more game steps than the physical route. This measure also provides a sense of order, that matches the goal of the experiments: taking route P_{TSP} is better than driving through D_{TSP}, which is still better than following N_{TSP}.

The maps obtained by the runs can also be compared with those maps hand-crafted for the WCCI 2012 PTSP Competition. Table 6.2 shows the estimated costs of the three possible routes and the fitness associated with each one of the maps distributed with the competition framework. As can be seen, the fitness of the competition maps is very different from the results obtained at the end of the experiments presented in this study. Indeed, sometimes it is even better not to follow the routes proposed for the physical TSP route planner. This confirms the findings from the last chapter, where controllers that followed P_{TSP} routes were not better than others following an N_{TSP} order, in some of the maps from the competition.

If all 64 maps from the WCCI 2012 PTSP Competition are taken into account, the average fitness of the maps using the estimated cost is $f_3 = -12.06$ (4.06). This shows that it is not straightforward to create maps by hand in which the most involved route planning leads to a clear victory. It is worth mentioning, however, that the maps designed for the competition were not created purely with the aim of being a challenge, but also with the objective of making them aesthetically pleasing, a feature not considered in this research.

It is also interesting to compare how the evolved maps differ from randomly created maps. In the case of three routes, the fitness of 100 randomly initialized maps (after being repaired) is, using the

Map	$EC(N_{TSP})$	$EC(D_{TSP})$	$EC(P_{TSP})$	$Fitness f_3$
1	1298	1103	1091	-12
2	1010	952	922	-30
3	1032	1001	990	-11
4	1146	1146	1152	6
5	1558	1415	1415	0
6	1197	1269	1121	72
7	1070	1070	1070	0
8	1357	1281	1281	0
9	1463	1415	1337	-48
10	1366	1014	1014	0

Table 6.2: Estimated Costs f_3 on maps of the 2012 PTSP Competition.

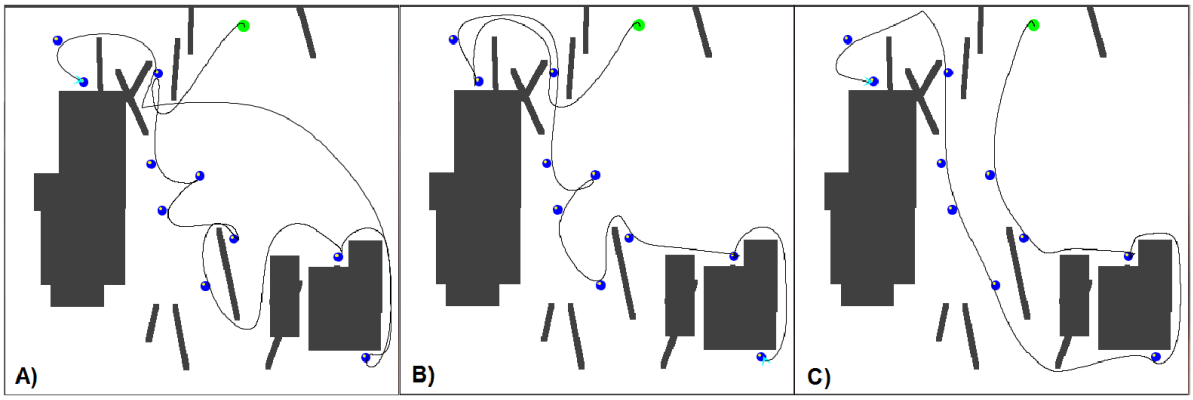


Figure 6.2: *Example of a map and its three routes evolved with CMA-ES. The trajectories, followed by the MCTS driver, are shown in the following order, from left to right: A) N_{TSP} route, with an average of* 2649.75 (78.03) *time steps; B) D_{TSP} route, with an average of* 2037.6 (43.82) *time steps; C) P_{TSP} route, with an average of* 1658.0 (37.93) *time steps.*

estimated cost, −4.24 (1.95); very different from the −427 shown in Figure 6.1.

A representative example:

Figure 6.2 shows the map and three routes obtained in one of the runs explained in this section. The MCTS verification step produced averages of 2649.75 (78.03), 2037.6 (43.82) and 1658.0 (37.93) for each one of the routes N_{TSP}, D_{TSP} and P_{TSP}, respectively. According to the relative performance, D_{TSP} and N_{TSP} routes spend respectively around 22% and 59% more time steps than P_{TSP} to visit all waypoints and complete the game. This picture also backs up a concept previously mentioned in this study: routes that visit waypoints in a straight (or almost straight) line can take advantage of the speed of the ship, even if the distance travelled is higher. In this case, it can be seen that the P_{TSP} route has much fewer changes in direction than the other two and a good balance between travelling a long distance (as in the N_{TSP} route) and the shortest possible distance (given by D_{TSP}).

Figure 6.3 shows the evolution of the fitness f_3 of the best individual of each generation during this

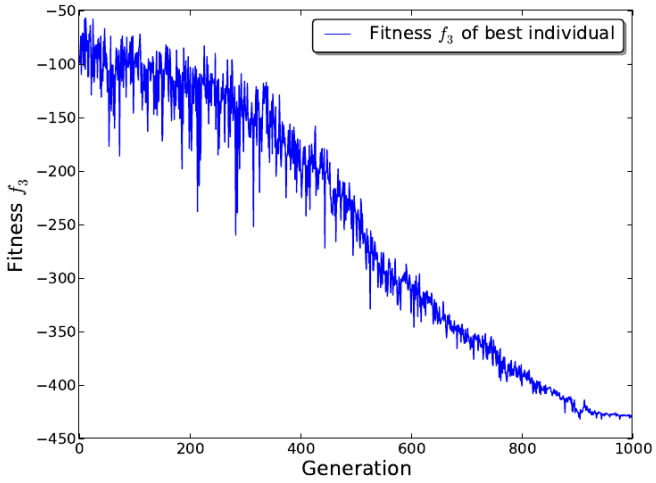


Figure 6.3: *Evolution of the fitness of the best individual during one of the* 40 *runs performed with* 3 *routes.*

particular run. It is easier to observe in Figure 6.3, rather than in Figure 6.1, that during the first half of the generations given, the algorithm explores the search space, obtaining very different values for the f_3 measure. During this time, the best individual fitness decreases slowly. Once half the generations are past, the fitness values become less variable and decrease rapidly, converging towards a solution and reaching a stable fitness value of -430 by the end of the run.

6.4.2 Heuristic Cost Estimation: 5 routes

This section analyzes the results obtained after performing 40 runs of the algorithm to obtain maps that differentiate among 5 routes. The only difference in the setup with respect to the 3 route case is that now the fitness function employed is f_5 (from Equation 6.4). As explained in Section 6.3.3, r_2 is the second best route derived from P_{TSP} using the *2-Opt* and *3-Opt* parameters. Figure 6.4 shows the average of the best individuals on each generation in the 40 runs performed.

As in the 3 routes case, there is a clear improvement of the fitness. In this case, the average fitness of the best individuals of the last population is $f_5 = -369$ (12.08). It is interesting to observe, comparing Figures 6.1 and 6.4, how the fitness obtained in this case is not as good as the ones obtained before for 3 routes. This is logical, owing to the fact that this scenario is more complex than the previous one, as the quality of the route r_2 is, by definition, the second best route obtained with the P_{TSP} solver. Although the progress of fitness is clear and significant, it visibly converges slower than when the P_{TSP}

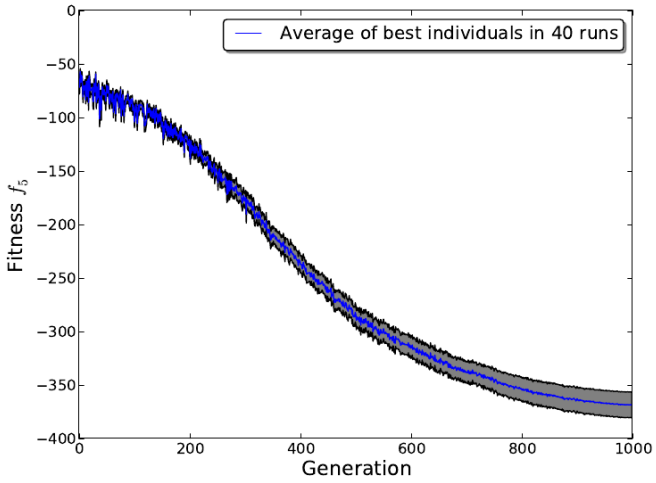


Figure 6.4: *Averaged evolution of the best individuals per generation of the* 40 *runs executed with the heuristic cost estimator for routes, employing* 5 *routes per map. Shadowed area indicates the standard error.*

N_{TSP}	D_{TSP}	$P_{TSP} \equiv r_0$	r_1	r_2
1.53 (0.02)	1.39 (0.02)	0.99 (0.01)	1.01 (0.01)	1.0 (0.0)

Table 6.3: Relative Average Performance in best maps of 40 runs.

route is employed to calculate fitness. In this case, around 60% of the runs converged to a solution before reaching the maximum of 1000 generations.

As before, a verification phase with the MCTS driver has been performed to analyze the resultant maps, employing the fitness function declared in Equation 6.5. In this case, the average fitness obtained by the MCTS driver is $f'_5 = -196.27$ (23.45), which though again smaller than its counterpart for 3 routes, still demonstrates that the maps obtained have the desired properties. Table 6.3 shows the relative average fitness for the results obtained with the MCTS driver.

In this case, this table also shows the results obtained with the other 2 physical routes (P_{TSP} and r_1). It is clear that the proposed algorithm is able to find maps where three different routes (given by the physical TSP planner: P_{TSP}, r_1 and r_2) provide a similar performance, all of them better than the route planned by D_{TSP}, which is still better than N_{TSP}.

A comparison with hand-crafted maps from the 2012 WCCI PTSP Competition has also been made. Table 6.4 shows the results of the heuristic estimated cost and the fitness values in the maps distributed with the competition framework. If all 64 competition maps are to be compared, the average fitness is $f_5 = 39$ (4.89), which is a much worse value than the one obtained in the experiments described here.

Map	$EC(N_{TSP})$	$EC(D_{TSP})$	$EC(r_2)$	$Fitness f_5$
1	1298	1103	1126	23
2	1010	952	1020	68
3	1032	1001	1024	23
4	1146	1146	1183	37
5	1558	1415	1470	55
6	1197	1269	1172	72
7	1070	1070	1142	72
8	1357	1281	1368	87
9	1463	1415	1415	0
10	1366	1014	1135	121

Table 6.4: Estimated Costs f_5 on maps of the 2012 PTSP Competition.

Average	N_{TSP}	D_{TSP}	P_{TSP}	r_1	r_2
Time	2279.0	1972.2	1626.0	1591.75	1650.2
Relative	1.38	1.19	0.98	0.96	1.0

Table 6.5: Performance of the MCTS driver in a run with 5 routes.

Again, the results obtained in this section can be compared with randomly created maps. The fitness of these random maps, according to the fitness function f_5, is 24.0 (3.45). It is straightforward to see that the problem with 5 routes is more complex than that with 3, as there is a clear difference between the fitness of random maps in both scenarios. A positive value like this indicates that the physics-based route r_2 is a worse choice than following the nearest waypoint first approach. This shows that it is not trivial to create maps with the features desired.

These comparisons are examples of one of the main advantages of using the technique described in this research. If map designs are not good enough (as happened in the PTSP Competition), it may be better to follow a naive route (N_{TSP}) and obtain the same results as taking a more complex approach (D_{TSP}), as in map 7, or even better, as in map 6. This can be extrapolated to other games, where simulated based PCG can avoid unfair situations in which simpler players exploit deficiencies in levels to beat better players.

A representative example:

Figure 6.5 shows an example of one of the runs with 5 routes and the map evolved by CMA-ES. After running the MCTS verification step, the average fitness obtained is shown in Table 6.5. As can be seen, the three physical routes provide a similar performance, while both the distant and nearest ones need more time to be completed.

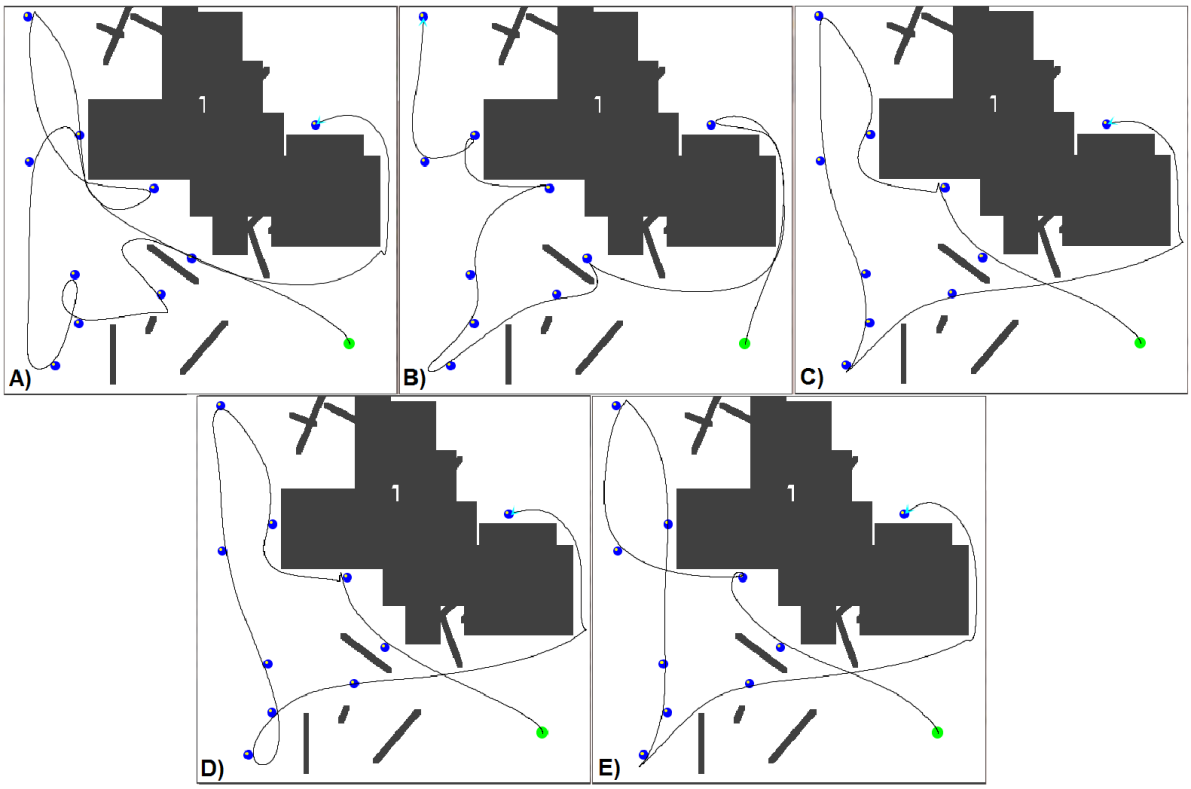


Figure 6.5: *Example of a map and its five routes evolved with CMA-ES. The trajectories, followed by the MCTS driver, are shown in the following order, from left to right, top to bottom: A) N_{TSP} route, with an average of* 2279.0 (50.33) *time steps; B) D_{TSP} route, with an average of* 1972.2 (34.29) *time steps; C) P_{TSP} route, with an average of* 1626.0 (60.96) *time steps; D) r_1 route, with an average of* 1591.75 (60.97) *time steps; E) r_2 route, with an average of* 1650.2 (86.1) *time steps.*

Figure 6.6 shows the average of some of the 114 genes of the individuals during the evaluations performed in the run. It is interesting to see how, during the first half of the experiments, these values have a high variance, which corresponds to the exploratory phase of the algorithm. However, close to the end of the run, these genes stabilize and the standard deviation of each one of them is reduced significantly, showing that the algorithm is converging to a solution.

6.5 Conclusions

The results presented in this research show the success of using adaptive controllers as a tool for creating maps that fulfil the conditions required. In this case, we have seen PTSP maps where naive and simple approaches are easily outperformed by those route planners that take the physics of the game into account when generating the routes. Additionally, the experiments performed in this research show that it is possible to generate maps where several trajectories provide solutions that are far better than the ones obtained by naive routes.

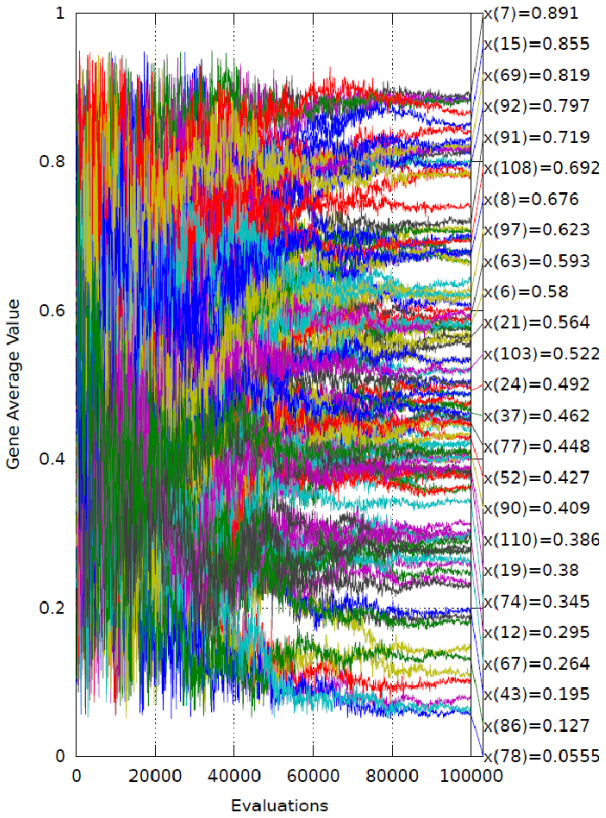


Figure 6.6: *Evolution of the values of some of the* 114 *genes of the individual. The average value of each gene is presented (between* 0.05 *and* 0.95*) against the number of evaluations performed by CMA-ES. The column on the right shows the final values for the mean m of the genes shown here.*

This research can be extended in several ways, and the use of adaptive controllers will still be needed for these. For instance, one possibility is to allow the evolutionary algorithm to modify aspects of the game, such as the number of waypoints or even the rules themselves (for example, the time steps requested to visit each waypoint), allowing the algorithm to adjust the difficulty of the level. Another option would be to enhance the aesthetic aspects of the maps or, since the PTSP game is also available to be played by humans, to create maps that are fun to play and research what features in the maps make a PTSP map entertaining.

The conclusions and algorithms described here can also be applicable to other games and domains. Simulation based PCG allows the creation of maps or levels where a simplistic approach is not good enough to tackle the game, or where other more complex behaviours clearly outperform the simpler ones. A game/level designer would not like to create levels with loopholes that can be easily exploited by naive approaches. Additionally, this research shows that it is possible to employ simulation based PCG techniques to evolve mazes with distinct levels of difficulty, or even to propose maps where different

solutions play with a similar performance, but much better than simpler approaches.

In a scenario where the use of PCG is extended to evolve certain parameters that affect the rules of the game, it would be possible for evolution to discard those rules that make the game unbalanced. A good example here is designing opposing armies for strategy games. If some army units provide an army with a strong advantage, it unbalances the game and detracts from the player experience. This balancing is currently performed by hand through a combination of game design and testing, but one can envisage a procedure where the system tweaks the rules of a game to improve its playability.

"Do you see the absurdity of what I am?
I can't even express these things properly
because I have to - I have to
conceptualize complex ideas in this stupid
limiting spoken language! . . . "
- Brother Cavil, Battlestar Galactica

Chapter 7

Adaptive Controllers for Online Learning

This chapter introduces the concept of macro-actions for online learning, employing the Physical Travelling Salesman Problem (2012). It also compares the performance of MCTS and Rolling Horizon Evolutionary Algorithms. Most of the material in this chapter has been published in the papers (Perez, Samothrakis, Lucas, and Rohlfshagen, 2013b) and (Perez, Powley, Whitehouse, Rohlfshagen, Samothrakis, Cowling, and Lucas, 2013a).

7.1 Introduction

When it comes to planning (or control), evolutionary algorithms are mostly used as follows: an evolutionary algorithm is used in conjunction with an off-line simulator in order to train a controller, for example as in (Gomez and Miikkulainen, 1999). The controller is then used in the real problem. This approach is not just used for control problems, but has been popular in fields such as Artificial Life (Beer and Gallagher, 1992).

The first concept introduced by this chapter proposes a different approach for evolutionary algorithms on planning. Evolution is used in the same manner as MCTS uses roll-outs and the forward model. Thus, an agent will evolve a plan by performing simulations in an imaginary model and, when the time budget is over, will act on the real world by performing the first action of its plan. The next cycle, a new plan is evolved, repeating this process until the game is over.

This type of behaviour is called "Rolling Horizon", "Receding Horizon" or "Model Predictive" control or planning. There have been some efforts in the past to use such approaches (Samothrakis and Lucas, 2010; Mercieca and Fabri, 2011) for the Mountain Car problem, and this chapter explores

two versions of a genetic algorithm for real-time games. The term Rolling Horizon comes from the fact that planning happens until a limited look-ahead in the game (and thus replanning is needed at every time step). Contrary to MCTS, the regret bounds of genetic algorithms in the general case are uncertain (Reeves and Rowe, 2002).

The second concept explored here is introducing *macro-actions*. The idea is simple: instead of considering a move as a single action, a move is composed of several actions in a sequence, and plans are made according to this definition. In essence, this reduces the search space significantly, improving the algorithm performance.

Grouping actions is not a new concept in the literature, and it has been used in different ways. For instance, Childs et al. (Childs, Brodeur, and Kocsis, 2008) (recently revisited by Van Eyck et al. (Van Eyck and Müller, 2012)) applied combined groups of actions when using UCT in the artificial game tree P-Game. A P-Game tree is a minimax tree where the winner is decided by a counting mechanism on the final state of the board. In their study, the authors group several similar actions in a single move to reduce the branching factor, obtaining promising results.

Balla and Fern (Balla and Fern, 2009) applied UCT to the Real Time Strategy game Wargus. This game is based on tactical assault planning, where multiple units are concurrently performing actions in the game and different actions, executed simultaneously, can take a variable duration. The authors discretize this continuous state in an abstraction that simplifies the characteristics of the current game layout: transitions between states are managed by actions that affect a group of units, reducing the size of the tree search and allowing more efficient simulations. Similarly, Powley et al. (Powley, Whitehouse, and Cowling, 2011) grouped similar actions when applying MCTS to the card game of Dou Di Zhu, where a single move choice is sometimes split into two separate and consecutive decisions to reduce branching factor at the expense of tree depth.

In this chapter, macro-actions are used in the PTSP game as a way to reduce the search space. Section 7.2 explains macro-actions and the reduction of search space achieved. Section 7.3 presents the algorithms used in this study, and Section 7.4 details the experiments performed. Section 7.4.5 ends this chapter with some conclusions.

7.2 Adaptive Controllers for the PTSP

This section introduces macro-actions and the value function to score states.

7.2.1 From Single to Macro-Actions

PTSP (see Section 3.2) allows for six different actions that can be applied at each time step. Each action must be decided within the 40ms of time allowed for the controller to choose a move. A good quality solution for a PTSP map with 10 waypoints requires between 1000 and 2000 time steps or action decisions, depending on the complexity of the map. This creates a search space of $(1000, 2000) \rightarrow (6^{1000}, 6^{2000})$, which obviously makes it impossible to calculate the optimal route within the time limitations given.

A reasonable reduction of this search space can be achieved by limiting how far ahead in the game the algorithms can look. As in the previous chapters, the order of waypoints is calculated during the initialization process. The controller has information about how many and which waypoints have been visited already, and it knows which ones are the next N waypoints to be collected. If only the next 2 waypoints are accounted for, and according to the game length estimates, the search space size is reduced to $(6^{2\times100}, 6^{2\times200})$ (assuming that the agent usually takes 100 to 200 actions per waypoint).

More reduction can be achieved by introducing the concept of *macro-actions*. In this case, a macro-action is just the repetition of the same action for L consecutive time steps. The impact of changing a single action in a given trajectory is not necessarily big, so high quality solutions may still be found by not using such fine-grained precision. For instance, choosing a macro-action *length* (defined as the number of repetitions per macro-action) of 10, reduces the search space size to $(6^{20}, 6^{40})$. Different values of L are used during the experiments described in this study in order to show how this parameter affects the performance of each one of the algorithms described.

Macro-actions have a second and important advantage: they allow longer evaluations to execute before taking a decision (through several game steps). In a single action scenario, each algorithm has only 40 ms to decide the next action to execute. However, in the case of a macro-action that is composed by L single actions, each macro-action has $L \times 40$ ms to decide the move, as the previous macro-action needs L time steps to be completed (with the exception of the very first move, where there is no previous action). Algorithm 5 defines a handler for macro-actions.

The algorithm works as follows: the function *GetAction*, called every game step, must return a

Algorithm 5 Algorithm to handle macro-actions.

```
1:  function GetAction(GameState : gs)
2:      if IsGameFirstAction(gs) then
3:          actionToRun ← DecideMacroAction(gs)
4:      else
5:          for i = 0 → remainingActions do
6:              gs.Advance(actionToRun)
7:          if remainingActions > 0 then
8:              if resetAlgorithm then
9:                  Algorithm.Reset(gs)
10:                 resetAlgorithm ← false
11:             Algorithm.NextMove(gs)
12:         else                                    ▷ remainingActions is 0
13:             actionToRun ← DecideMacroAction(gs)
14:     remainingActions = (remainingActions − 1)
15:     return actionToRun
16:
17: function DecideMacroAction(GameState : gs)
18:     actionToRun ← Algorithm.NextMove(gs)
19:     remainingActions ← L
20:     resetAlgorithm ← true
21:     return actionToRun
```

(single) action to execute in the current step. When the function is called for the first time (line 3), or the number of remaining actions is back to 0 (line 13), the function *DecideMacroAction* is called. This function (lines 17 to 21) executes the decision algorithm (which could be tree search, evolution or random search), that is in charge of returning the move to make, and sets a counter (*remainingActions*) to the length of the macro-action (L). Before returning the action to run, it indicates that the next cycle in the algorithm needs to be reset.

In cases where the current step is not the first action of the game, the state is advanced until the end of the macro-action that is currently being executed (line 6). This way, the decision algorithm can start the simulation from the moment in time when the current macro-action will have finished (note that this is possible because the game is completely deterministic). The algorithm is then reset, if previously indicated to do so by *DecideMacroAction*, and finally evaluates the current state to take the next move (line 11).

Note that in this case the value returned by *NextMove* is ignored: the action to be executed is *actionToRun*, decided in a different time step. The objective of this call is to keep building on the knowledge the algorithm is gathering in order to decide the next macro-action (which effectively happens in *DecideMacroAction*) while the ship makes the move from the `previous` macro-action decision. Therefore, *Algorithm.NextMove* is called L times, but only in the last one is a real decision

made.

It is important to note that the algorithms determine a best solution of N macro-actions with L single actions per macro-action. However, only the `first` macro-action of the best solution is finally handled and executed by the controller. Once the best individual has been determined after L iterations, the first macro-action is used and the rest are discarded, as the algorithm is reset at that point. The decision algorithm is in charge of keeping its internal state from one call to the next, replying within the allowed time budget and resetting itself when *Algorithm.Reset* is called.

7.2.2 Score function

Even the usage of macro-actions does not ensure that any succession of N actions reaches the next two waypoints in the (pre-planned) route: it is unlikely that a random sequence of actions visits even one of them. In fact, the vast majority of random trajectories will not visit any waypoints at all. It is thus necessary to improve the granularity of the rewards by defining a function that is able to distinguish the quality of the different trajectories, by scoring the game in a given state. This function is defined independently from the algorithm that is going to use it, so the research is centred on how the search space is explored by the different algorithms.

The score function takes into account the following parameters: distance and state (visited/unvisited) of the next two waypoints in the route, time spent since the beginning of the game and collisions in the current step. The final score is the addition of four different values. The first one is given by the *distance points*, which is defined as follows: being d_w the distance to the waypoint w, the reward for distance r_{dw} is set to $10000 - d_w$. If the first waypoint is not visited, it is set to r_{d1}. However, if the first waypoint is visited, this is set to r_{d1} plus a reward value (10000). The other values for the score function are given by the waypoints visited (number of visited waypoints, out of the next two, multiplied by a reward factor, 1000), time spent (set to 10000 minus the time spent during the game) and collisions (-100 if a collision is happening in this step).

Figure 7.1 shows a scheme of the controllers employed in this study. All four controllers described in upcoming sections share this same framework.

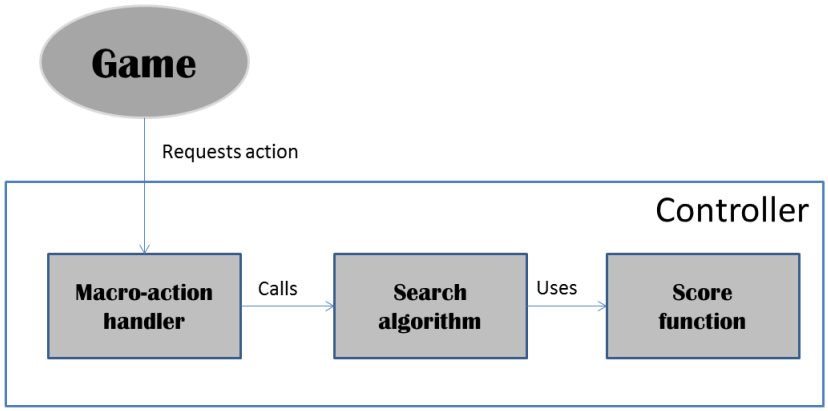


Figure 7.1: *General controller framework.*

7.3 Algorithms for Online Learning

This section presents the three algorithms employed in this study: a Genetic Algorithm, Monte Carlo Tree Search and simple Random Search.

7.3.1 Genetic Algorithm

A Genetic Algorithm (GA), firstly introduced by Holland (Holland, 1992), is a process that, by means of mimicking biological evolution, generates and evaluates a population of solutions to a given problem. The algorithm evolves "individuals" (strings, each one of its elements known as a *gene*), each one of them representing a possible solution that is evaluated by a *fitness function*, in charge of determining how well the given individual solves the problem at stake. Evolution is simulated by applying different genetic operators: *selection*, *crossover* and *mutation*.

Selection determines which individuals from the population must be chosen for crossover. This can be done via tournaments of size N (where N individuals are randomly selected, compared their fitness, and the best one is chosen), fitness proportionate selection (by randomly choosing individuals, biasing the selection by their fitness), reward-based selection (where the probability of choosing an individual is proportional to the sum of its fitness and their parents), or other methods. Once two individuals (parents) have been selected for crossover, their genes are recombined to generate a new individual. Again, different methods can be used, like uniform or one/n point crossover. The resultant individual is then mutated, a process based on randomly varying one or more genes according to certain probability (or *mutation rate*), in order to explore other areas of the search space.

The process continues until a new population is created, when all the individuals are evaluated and assigned a fitness. It is also possible to promote the overall best individual(s) to the next population directly, a strategy called *elitism*. The algorithm ends when a pre-defined stopping criterion is met, such as number of iterations or a fitness-based measure. For more details about GAs, the reader is referred to (Holland, 1992).

The two versions of the GA employed in this study are simple evolutionary algorithms, where each individual encodes a group of macro-actions using integers. Each one of the possible six actions is represented with a number from 0 to 5. Note that each one of these genes represents a macro-action: the action denoted by the gene is repeated for L steps in the game where L is defined in the experimental setup.

All individuals in the population have a fixed length, as defined in the experimental setup section (Section 7.4). As the algorithm is meant to run online (i.e. while the game is actually being played), a small population size of $P = 10$ individuals is used in order to be able to evolve over a large number of generations. Elitism is used to automatically promote the best $E = 2$ individuals of each generation to the next. Evolution is stopped when $(40ms - \epsilon)$ is reached. ϵ is introduced in order not to use more time than allowed to provide an action.

Individuals are initialized uniformly at random such that each gene can take any value between 0 and 5. This initialization happens many times during each game: in the first step of the game and every L moves, as described in Section 7.2.1. Mutation is used with different rates: 0.2, 0.5 and 0.8. Each time a gene is mutated, only one of the two inputs involved in the action is randomly chosen to be changed. If acceleration is changed, the value is flipped from *On* to *Off*, or vice versa. If steering is modified, *left* and *right* mutate to *straight*, while *straight* can mutate to *right* or *left* at 50% chance. This procedure was designed in order to transit smoothly between actions. Tournaments of size 3 are used to select individuals for uniform crossover.

Finally, the evaluation of an individual is performed by the following mechanism: given an individual with N macro-actions and L single actions per macro-action, the game is advanced, using the forward model of the PTSP (mentioned in Section 3.2), applying the $N \times L$ actions defined in the individual. Then, the resultant game state is evaluated with the score function defined in Section 7.2.2.

Two different variants are employed in this study. The first one (referred to as GAC), implements all

features described above. The second one (herein simply referred to as GA) does not employ tournament selection nor crossover. In this case, the individuals are sorted by fitness within the population. Elitism is used to promote the best $E = 2$ individuals to the next generation, and the rest of members are generated by mutating the best $P - E = 8$ individuals.

7.3.2 Monte Carlo Tree Search

In the problem presented in this study, each of the actions in the tree is a macro-action. This means that, when navigating through the tree, L single actions must be performed to move from one state to the next, defined by the action picked. In most games, the depth of the tree is limited by the number of moves required to reach an end game state. In the case of the PTSP, the end of the game is usually not reached, so the maximum depth is determined by the number and length of the macro-actions, defined differently for each experiment, bringing it closer to rolling horizon search then normal MCTS.

The tree is preserved from one game step to the next (i.e. during L moves) and is discarded every time the algorithm is reset. When the controller asks for a move, the action chosen is the one that corresponds to the child of the root with the maximum expected value $Q(s, a)$.

7.3.3 Random Search

Random Search (RS) is a simple algorithm that creates random solutions of a specified length. During the allocated time, new individuals are created uniformly at random and evaluated, keeping the best solution found so far. As in the other solvers, the best solution is forgotten every time the algorithm is told to be reset. The motivation behind including RS in this research is the good results obtained by the GA with high mutation rates. Thus, it is important to investigate how the results achieved by RS compare to those.

7.4 Experimental work

The experiments presented in this study have been carried out on 10 different maps, running 20 matches per maze. Hence, each configuration has been executed during 200 games, in order to get reliable results. The maps are those distributed within the framework of the PTSP competition (see Appendix A).

Five different configurations are used for each algorithm, varying the number of macro-actions (N) and the number of single actions per macro-action (L). These configurations, indicated by the pairs (N,L), are as follows:

$$\{(50, 1), (24, 5), (12, 10), (8, 15), (6, 20)\}$$

The last four configurations share the property of foreseeing the same distance into the future: $N \times L = 120$. The only exception is the first configuration, where $L = 1$. This is considered as a special case as no macro-actions are used. Here, simulation depth is set to 50 actions, a value that gave good results in the past, as shown in Section 5.4.1 and Table 5.1 (although note that the heuristics employed in this research are not the same).

A key aspect for interpreting the results is to understand how games are evaluated. If two single games are to be compared, following the rules of PTSP, it is simple to determine which game is better: the solution that visits more waypoints is the best. If both solutions visit the same number of waypoints, the one that does so in the minimum number of time steps wins.

However, when several matches are played in the same map, determining the winner is not that simple. One initial approach is to calculate the averages of waypoints visited and time spent, following the same criteria used in the single game scenario. The problem with this approach is that the difference between two solutions, taken as the average of waypoints visited, can be quite small (less than one waypoint), while the difference in time steps can be large. For instance, imagine a scenario where two solvers (A and B) obtain an average of waypoints (w_i) of $w_a = 9.51$ and $w_b = 9.47$, and an average of time steps (t_i) of $t_a = 1650$ and $t_b = 1100$. The traditional comparison would say that A wins (because $w_a > w_b$), but actually B is much faster than A ($t_b \ll t_a$) with a very small waypoint difference. Intuitively, B seems to be doing a better job.

A possible solution to this problem is to calculate the ratio $r_i = t_i/w_i$ that approximates the time taken to visit a single waypoint. The drawback to this approach is that it does not provide a reliable comparison when one of the solvers visits a small number of waypoints (or even 0). Therefore, the following two features have been analysed:

- *Efficacy*: number of waypoints visited, on average. The higher this value, the better the solution provided.

- *Efficiency*: ratio $r_i = t_i/w_i$, but only for those matches where w_i equals the number of waypoints on the map. The goal is to minimize this value, as it represents faster drivers.

7.4.1 40 milliseconds

Table 7.1 shows the number of waypoints visited on average, including the standard error. The first notable aspect is that, out of the algorithms presented here, when no macro-actions are provided ($L = 1$), only MCTS is able to solve the problem up to a point. The other three algorithms are not able to complete the game, not getting any waypoint, or visiting only one in some rare cases. When the length of macro-action is increased to 5, on average all algorithms achieve close to all waypoints. Perfect efficacy is obtained when the macro-action length is raised to 15, which seems to be the overall best value for this parameter. In general, the GA (especially with high mutation rates) seems to be the algorithm that obtains the best efficacy.

	Average Waypoints Visited							
L	MCTS	RS	GA 0.2	GA 0.5	GA 0.8	GAC 0.2	GAC 0.5	GAC 0.8
1	7.66 (0.25)	0.03 (0.01)	0.03 (0.01)	0.05 (0.01)	0.01 (0.01)	0.03 (0.01)	0.02 (0.01)	0.01 (0.01)
5	9.62 (0.11)	9.76 (0.08)	9.87 (0.06)	9.89 (0.06)	9.86 (0.06)	9.91 (0.05)	9.9 (0.05)	9.69 (0.09)
10	9.62 (0.12)	9.95 (0.03)	9.96 (0.03)	10.0 (0.0)	10.0 (0.0)	9.96 (0.04)	9.91 (0.06)	9.96 (0.03)
15	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)
20	9.97 (0.03)	9.95 (0.04)	10.0 (0.0)	10.0 (0.0)	10.0 (0.0)	9.97 (0.02)	9.98 (0.02)	9.92 (0.04)

Table 7.1: Waypoint visits with 40ms per time step.

Regarding efficiency, Figure 7.2 shows the ratio average with standard error obtained with the GA (very similar results are obtained with GAC). One of the first things to note is that the mutation rate that obtains the best solutions is 0.5, for both evolutionary algorithms, especially when the macro-action length is 10 or lower. It is interesting to see how the distinct mutation rates make the algorithm behave differently: it is clear that a higher mutation rate works better with longer macro-actions, obtaining similar results to 0.5. On the other hand, although rate 0.5 is still the best, 0.2 provides better results than 0.8 for smaller values of L. This is confirmed by a Mann-Whitney-Wilcoxon statistical test (MW-test) run on this data: GA-0.5 is better than GA-0.8 for $L = 5$ and 10 (p-values of 0.006 and 0.041,

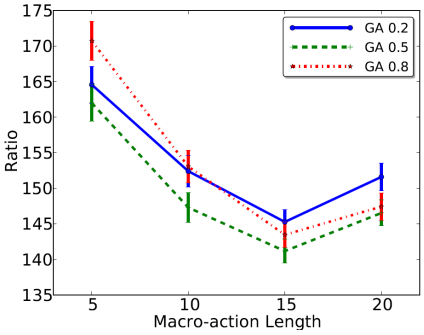


Figure 7.2: *Ratio (time/waypoints) results, only for GA.*

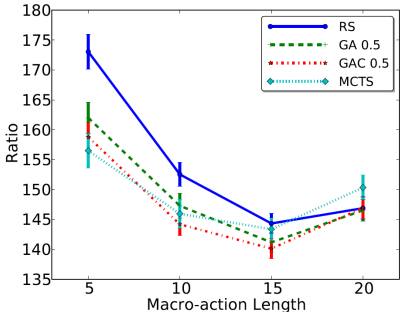


Figure 7.3: *Ratio results.*

respectively), while GA-0.5 is faster than GA-0.2 for higher values of L (p-values of 0.046 and 0.022 for $L = 15, 20$).

Figure 7.3 shows the ratio average and standard error of RS, MCTS, GA and GAC (both with mutation rate 0.5). One initial result to note is that the 2012 PTSP winner algorithm (MCTS) is no better than GA or GAC in any of the macro-action lengths tested (with the exception, as mentioned earlier, of $L = 1$, which obtains a ratio of 168.37 (7.7)). Actually, although overall there is no statistical difference, GA and GAC get better results than MCTS in some of the maps tested (see Section 7.4.3). It is also worthwhile mentioning that $L = 15$ provides the best results for all algorithms. In fact, these results are compatible with previous studies: Powley et al. (Powley, Whitehouse, and Cowling, 2012) used $L = 15$ in their MCTS entry that won the 2012 WCCI PTSP competition.

Finally, we observe an interesting phenomenon: while RS is clearly worse than the other algorithms in lower macro action-lengths ($L = 5, 10$), the performance of all algorithms converges to the same performance when the maximum value of L is reached.

A possible explanation for this effect is as follows: for the maximum macro-action length ($L = 20$), the length of the solution (or individual) that is being evaluated is $N = 6$, providing $6^6 \approx 3 \times 10^5$ combinations. This evaluation performs single actions during $L \times N = 120$ time steps, but remember that only the first macro-action is finally performed in the actual game, discarding the rest of the individual actions (as explained in Section 7.3). Clearly, not all the actions in the sequence of 120 moves are equally important; actions chosen at the beginning of this sequence produce a higher impact in the overall trajectory followed by the ship. Therefore, it is fair to assume that, for instance, the first half of the individual (60 actions) is more relevant for the search than the second. In the case of

$L = 20$, this is encoded in the first 3 genes, which provides $6^3 = 216$ different possibilities. This is a very reduced search space where even random search can find a good solution (especially considering the amount of evaluations performed per cycle, as shown in Section 7.4.4).

In the cases where $L = 5$ or $L = 10$, the same amount of look ahead (60 single actions) is obtained with 12 and 6 genes, which provides a search space of $6^{12} \approx 2 \times 10^9$, and $6^6 \approx 3 \times 10^5$ respectively. In these more complex scenarios, the search space is better explored by both GAs (and MCTS) than by RS.

7.4.2 80 milliseconds

All the experiments described above were repeated with the decision time doubled to 80 ms. For the sake of space, these results are not detailed here, but they are consistent with the results obtained from spending 40 ms per cycle.

All the algorithms perform slightly better when more time is permitted, as would be expected. For instance, GA-0.5 obtains the best performance when $L = 15$, with a ratio of 138.17 (1.56), about three units faster than the results achieved in 40ms. Again, the relationship between the different algorithms is maintained: all algorithms perform better with $L = 15$ and their performances get closer as the size of macro-action approaches 20.

7.4.3 Results by map

These results can also be analysed on a map by map basis. Figure 7.4 represents the ratio achieved by each algorithm in every map of this setup (remember lower is better). First of all, it is interesting to see how the overall performance of the algorithms changes depending on the map presented. For instance, map 9 happens to be especially complex. All algorithms obtain worse results in this map than, for instance, in map 10, that contains no obstacles except the boundaries of the map. Regarding averages, GAC-0.5 obtains the best average of ratio per map in 7 out of the 10 different maps, whilst MCTS leads the rest.

It is also possible to take the best results obtained in each map and compare them with the ones obtained in the WCCI PTSP competition. For instance, both GA and GAC obtain better solutions than the winner of the WCCI competition in 4 out of the 10 maps from the ones distributed with the

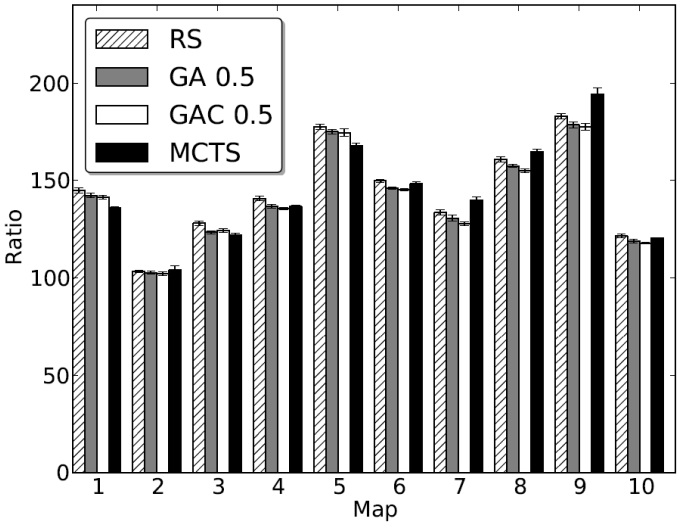


Figure 7.4: *Results per map,* $L = 15$.

competition framework. Similarly, MCTS outperforms the best results in two different maps.

The PTSP competition is quite strict regarding controllers that overspend the 40 ms of time allowed per cycle, executing the action 0 (no acceleration, no steering) when this occurs. In the experiments presented in this chapter, however, and for the sake of simplicity, this penalisation is ignored. In other words, controllers do use 40ms, but should this time limit be violated (which may happen if a single evaluation takes more time than expected) the desired action is executed anyway. In order to verify if the results obtained are comparable with those from the competition, statistics relating to the number of times the controllers overspent the real-time limit were extracted. This phenomenon occurs in only between 0.1% and 0.03% of the total game cycles in a game, which suggests that, even if the results cannot be fairly considered as new records for those maps, the performance achieved by the algorithms presented in this chapter is significant and is comparable to some extent.

7.4.4 Number of evaluations

Another aspect worth mentioning is the number of evaluations (or calls to the score function) performed every game cycle. MCTS is able to perform more evaluations per cycle, going from about 950 when $L = 5$ to 3000 if $L = 20$, while the other algorithms range between 330 and 370. The main reason for this being that MCTS stores intermediate states of the game in the tree, and is able to reuse this information during the simulations performed in a game cycle, whereas the other three algorithms

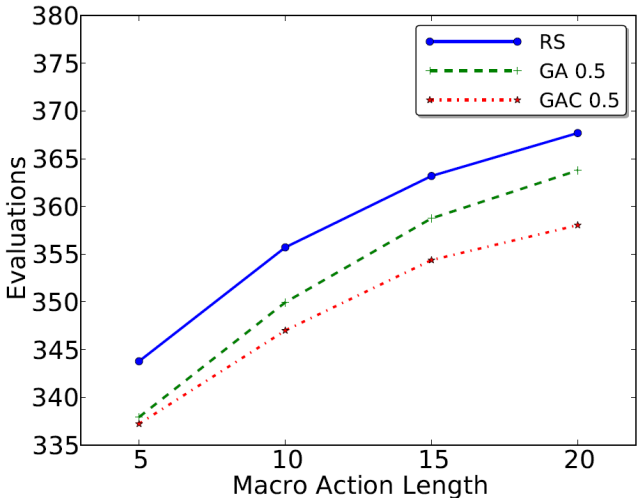


Figure 7.5: *Evaluations per algorithm and L.*

execute all the actions encoded in each individual, from the first one to the last, producing a more expensive and hence slower evaluation.

The difference between both *GAs* and *RS* depends on the overhead of the algorithm (and not on the evaluation itself, which takes the same time for all of them): as shown in Figure 7.5, *RS* is the more efficient because of its simplicity, followed by *GA* and then *GAC*, which is the slowest due to the cost of using selection and crossover. *MCTS* is not included in this picture for the sake of clarity.

7.4.5 Conclusions

This study compared the performance of several different search techniques for the PTSP, a real-time game where an action must be supplied every 40 ms, introducing the concept of macro-actions and a rolling horizon version for evolutionary algorithms.

MCTS is the only algorithm able to produce relatively good solutions when no macro-actions were used. The fact that this algorithm stores information in the tree allows for a more efficient search when the search space is vast. However, the best results were obtained using macro-actions, more specifically where a macro-action length of 15 was used. In this case, GAC and MCTS obtained better solutions on each one of the maps, in some cases achieving better results than those from the PTSP competition. The concept of macro actions is in no way restricted to the PTSP and may equally well be applied to the majority of real-time video games. Examples are high level commands in strategy games of first

person shooters, where the use of macro-actions instead of low level orders enhances the performance of search algorithms due to a reduction of the search space.

Another important contribution of this research is understanding how different discretizations of the search space affect the performance of the algorithms in this game. On the one hand, if the granularity is fine (small macro-action lengths, such as $L = 5, 10$), the search algorithm can benefit from artefacts such as the genetic operators. On the other hand, if the granularity is coarse the fitness landscape is rough, and the rolling horizon effect enables random search to perform as well as the other algorithms. Finally, it is worth mentioning that both GAs obtained very good solutions while performing fewer evaluations than the other techniques. This would suggest that the search for solutions is more efficient in these two algorithms. This is significant, suggesting that evolutionary algorithms offer an interesting alternative to MCTS for general video game agent controllers when applied in this on-line rolling horizon mode.

It is also worth mentioning that a different set of experiments were conducted in parallel to these, in collaboration with the winners of the PTSP Competition, Edward Powley and Daniel Whitehouse. For the sake of space, these experiments have not been included in this thesis, but they can be consulted in the published work (Perez et al., 2013a). This paper compared different algorithms (Depth First Search and Monte Carlo planning, instead of GA), using different score functions and TSP solvers for maps with 30, 40 and 50 waypoints. Results are similar, showing the strength of MCTS, the importance of creating an appropriate long-term plan, and the use of macro-actions.

Finally, it is worth highlighting that the implementation of macro-actions employed in this research is a rather simple one, as it consists of the repetition of the same single action during L time steps. The use of more complex macro-actions is a matter of further research. For instance, it would be possible to create macro-actions that combine single actions to describe specific curves or trajectories. Another interesting approach would be to learn which macro-actions are more appropriate for a specific game, or automatically adapt the type or length of the macro-actions to the features of the current state.

"But I know I want to reach out with something other than these prehensile paws! And feel the wind of a supernova flowing over me! . . . "
- Brother Cavil, Battlestar Galactica

Chapter 8

Multi-Objective MCTS

This chapter explores Multi-Objective MCTS for real-time games. Most of the material in this chapter has been published in the papers (Perez, Samothrakis, and Lucas, 2013c) and (Perez, Mostaghim, Samothrakis, and Lucas, 2014a).

8.1 Introduction

Viewed simplistically, most competitive games can be thought of as having a single objective: to win. While this is easily stated, achieving it is usually complex, otherwise the game would be of limited interest. This chapter proposes a change in perspective, considering games as systems with several, simultaneous, objectives to accomplish.

Multi-objective optimization has been a field of study in manufacturing, engineering (Marler and Arora, 2004) and finance (Coello, 2006), while having little impact on games research. The goal of this chapter is to show that multi-objective optimization has much to offer in developing game strategies that allow for a fine-grained control of alternative policies. The application of such approaches to this field can provide interesting results, especially in games that are long or complex enough that long-term planning is not trivial, and achieving a good level of play requires balancing strategies.

In real-time strategy games (for instance, Starcraft) where the objective is clearly defined (being the only one alive at the end of the match), the workarounds to achieve the victory may take several factors into account simultaneously, such as balancing attack power, defensive structures and resource gathering. When building up a team in a Role Playing Game, the different members need to be balanced to be competitive (strength, dexterity, and healing capabilities, for instance). Multi-objective

approaches may help in these scenarios, where different goals have to be attended to at the same time. Furthermore, even simpler games can benefit from these approaches. For instance, in Othello, apart from maximizing the stone difference between the players, an agent could try to avoid (or capture) as many corners of the board as possible, as this has proven to be a good strategy.

This chapter presents a Multi-Objective version of Monte Carlo Tree Search, testing it in three different benchmarks: the Deep Sea Treasure (DST), Puddle Driver (PD) and Multi-Objective PTSP (MO-PTSP), as defined in Sections 3.3, 3.4 and 3.5 respectively. This algorithm is also compared with a Rolling Horizon version of a popular, state of the art, Multi-Objective optimization algorithm: NSGA-II, described in detail in Section 2.4, along with general concepts of Multi-Objective optimization.

An initial attempt at Multi-Objective MCTS was addressed by Weijia Wang and Michele Sebag (Weijia and Sebag, 2012, 2013). In their work, the authors employ a mechanism, based on the HV calculation, to replace the UCB1 equation (Equation 2.1). The algorithm keeps a Pareto archive (P) with the best solutions found in end game states. Every node in the tree defines $\overline{r}_{sa}$ as a vector of UCB1 values, in which each $\overline{r}_{sa,i}$ is the result of calculating UCB1 for each particular objective i.

The next step is to define the value for each state and action pair, $V(s,a)$, as in Equation 8.1. $\overline{r}^{p}_{sa}$ is the projection of $\overline{r}_{sa}$ into the piecewise linear surface defined by the Pareto archive P (see Figure 8.1). Then, $HV(P \cup \overline{r}_{sa})$ is declared as the HV of P plus the point $\overline{r}_{sa}$. If $\overline{r}_{sa}$ is dominated by P, the distance between $\overline{r}_{sa}$ and $\overline{r}^{p}_{sa}$ is subtracted from the HV calculation. The tree policy selects actions based on a maximization of the value of $V(s,a)$.

$$V(s,a) = \begin{cases} HV(P \cup \overline{r}_{sa}) - dist(\overline{r}^{p}_{sa}, \overline{r}_{sa}) & Otherwise \\ HV(P \cup \overline{r}_{sa}) & if \quad \overline{r}_{sa} \preceq P \end{cases} \tag{8.1}$$

The proposed algorithm was employed successfully in two domains: the DST and the Grid Scheduling problem. In both cases, the results matched the state-of-the-art, albeit at the expense of a high computational cost. This, however, makes this algorithm unsuitable for real-time games. Although the work discussed here is influenced by Wang's approach, some modifications need to be made in order to overcome the high computational cost involved in their approach.

This chapter is structured as follows. First, Section 8.2 details the necessary heuristics for the games in this study. Section 8.3 presents the proposed algorithm, and Section 8.4 describes the experimental

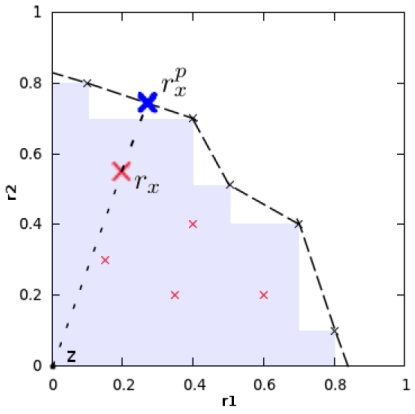


Figure 8.1: *r_x^p is the projection of the r_x value on the piecewise linear surface (discontinuous line). The shadowed surface represents $HV(P)$. From (Weijia and Sebag, 2012).*

work and results. Finally, Section 8.5 ends this chapter with some conclusions.

8.2 Heuristics

This section explains how different heuristics are employed for each game, in order to reduce the search space the algorithms must face, and to determine how the states found during the search will be scored. It is very important to note that all the algorithms compared here share the same heuristics (with the exception of transposition tables, which only make sense in tree search algorithms).

8.2.1 Transposition Tables in DST

The DST is a problem especially suited for the use of Transposition Tables (TT) within MCTS, an enhancement introduced by Childs et al. (Childs et al., 2008). TT is a technique used to optimize tree search-based algorithms when two or more states can be found at different locations in the tree. It consists of sharing information between these equivalent states in a centralized manner, in order to avoid managing these positions as completely different states. Examples of transpositions can be found in the work by Kozlek (Kozelek, 2009) for the game Arimaa, and by Méhat et al. (Méhat and Cazenave, 2010) and Saffidine (Saffidine, 2010) for Single-Player General Game Playing. In both cases, the use of transposition tables significantly improved the playing strength of the MCTS algorithm employed to play the games.

Figure 8.2 shows an example of this situation in the DST. In this example, the submarine starts in the initial position ($P1$, root of the tree) and makes a sequence of moves that places it in $P2$. From this

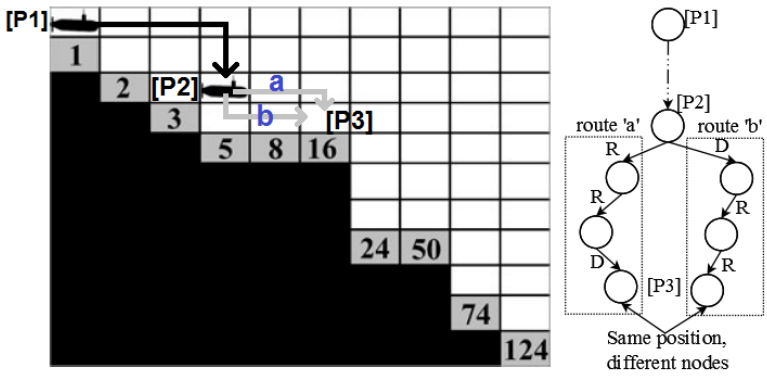


Figure 8.2: *Example of two different sequences of actions (R: Right, D: Down) that lie in the same position in the map, but at a different node in the tree.*

location, two optimal trajectories to move to $P3$ would be route a and b. In a tree that does not use TT, there would be two different nodes to represent $P3$, although the location and number of moves performed up to this point are the same (thus, the states are equivalent). It is worthwhile highlighting that the coordinates of the vessel are not enough to identify two equivalent states, the number of moves is also needed: imagine a third route from $P2$ to $P3$ with the moves: *Up*, *Right*, *Right*, *Down*, *Down*. As the submarine performs 5 moves, the states are not the same, and the node where the ship is in $P3$ now would be two levels deeper in the tree.

TT tables are implemented in the MCTS algorithms tested in this benchmark by using hash tables that store a *representative* node for each pair $(position, moves)$ found. The key of the hash map then needs to be obtained from three values: position coordinates x and y of the ship in the board, and number of moves, indicated by the depth of the node in the tree. Hence, transpositions can only happen at the same depth within the tree, a successful approach tried before in the literature (Kozelek, 2009).

8.2.2 Macro-actions for Puddle Driver and MO-PTSP

Both PD and MO-PTSP have several features that make them interesting and more challenging than the DST. First of all, they takes place in a continuous world, so discretizations obtained with techniques such as the transposition tables used here for the DST are not directly applicable to this game. Secondly, these games take longer than the DST to be finished (approximately 300 and 1500 steps for PD and MO-PTSP, respectively). Considering the 40 ms of time allowed per game step, creating plans for the whole route becomes complex. For this reason, macro-actions (as described in Chapter 7) are used for

both the algorithm proposed in this chapter and also in Rolling Horizon NSGA-II.

As before, a macro-action of length L is defined as a repetition of a given action during L consecutive time steps. In both PD and MO-PTSP, the macro-action size is $L = 15$, a value that has shown its proficiency before in PTSP, as results show in Chapter 7.

8.2.3 Evaluating the state

DST

DST has two different objectives: the number of moves and the value of the treasure collected. Hence, the quality of a state can be assessed by two rewards, ρ_p and ρ_v, for each objective respectively. ρ_p is adjusted to be maximized using the maximum number of moves in the game, while ρ_v is simply the value of the cell that holds the treasure. Therefore, the reward vector to maximize is defined as $\overline{r} = \{\rho_p, \rho_v\}$. Equation 8.2 summarizes these rewards:

$$\begin{aligned} \rho_p &= 100 - moves \\ \rho_v &= treasureValue \end{aligned} \tag{8.2}$$

PD

The reward scheme provided for the Puddle Driver consists of multiplying each one of the two objectives (*total time* and *damage*) by the distance from the current position of the ship to the goal's location. As, given the shape and size of the map, the maximum distance between two points is the length of the diagonal ($MD \simeq 725$), a factor f in the range [0,1] is obtained by calculating $f = 1 - \frac{dist}{MD}$, where $dist$ is the current distance between ship and goal. Thus, the two rewards can be defined as $\rho_t = (1000 - T) \times f$ and $\rho_d = (1000 - D) \times f$ where T (*total time*) and D (*damage*) are the values given by the game. The rewards vector $\overline{r} = \{\rho_t, \rho_d\}$ is meant to be maximized.

MO-PTSP

MO-PTSP provides an increase in difficulty as three different objectives need to be optimized. In order to evaluate a game state in MO-PTSP, the following measures are taken: ρ_t, ρ_f, ρ_d, one for each one of the objectives: time, fuel and damage, respectively. All these rewards are defined so they have to

be maximized. The first reward uses a measure of distance for the time objective, as indicated in Equation 8.3:

$$\rho_t = 1 - d_t/d_M \tag{8.3}$$

MO-PTSP, as PTSP, includes a path-finding library. d_t indicates the distance from the current position until the last waypoint, following the desired route, and d_M is the distance of the whole route from the starting position. Minimizing the distance to the last waypoint will lead to the end of the game.

Equation 8.4 shows how the value of the fuel objective, ρ_f, is obtained:

$$\rho_f = (1 - (\lambda_t/\lambda_0)) \times \alpha + \rho_t \times (1 - \alpha) \tag{8.4}$$

λ_t is the fuel consumed so far, and λ_0 is the initial fuel at the start of the game. α is a value that balances between the fuel component and the time objective from Equation 8.3. Note that, as waypoints need to keep being visited, it is necessary to include a distance measure for this reward. Otherwise, an approach that prioritizes this objective would not minimize distance to waypoints at all (the ship could just stand still: no fuel consumed is optimal), and therefore would not complete the game. The value of α has been determined empirically, in order to provide a good balance between these two components, and is set to 0.66.

Finally, Equation 8.5 gives the method used to calculate the damage objective, ρ_d:

$$\rho_d = \begin{cases} (1 - (g_t/g_M)) \times \beta_1 + \rho_t \times (1 - \beta_1), & sp > \gamma \\ (1 - (g_t/g_M)) \times \beta_2 + \rho_t \times (1 - \beta_2), & sp \leq \gamma \end{cases} \tag{8.5}$$

g_t is the damage suffered so far in the game, and g_M is the maximum damage the ship can take. In this case, three different variables are used to regulate the behaviour of this objective: γ, β_1 and β_2. Both β_1 and β_2 have the same role as α in Equation 8.4: they balance between the time objective and the damage measure. The difference is that β_1 is used in high speeds, while β_2 is employed with low velocities. This is distinguished by the parameter γ, that can be seen as a threshold value for the ship's speed (sp). This differentiation is made in order to avoid low speeds in lava lakes, as this significantly

increases the damage suffered. The values for these variables have also been determined empirically, and they are set to $\gamma = 0.8$, $\beta_1 = 0.75$ and $\beta_2 = 0.25$.

The final reward vector to be maximized is therefore $\overline{r} = \{\rho_t, \rho_f, \rho_d\}$. It is important to highlight again that these rewards are the same for all the algorithms tested in the experiments.

8.3 Multi-Objective MCTS for Real-Time Games

It is worthwhile mentioning that in most cases found in the literature, MCTS techniques have been used with some kind of heuristic that guides the Monte Carlo simulations or the tree selection policy. In the algorithm proposed in this study, simulations are purely random, as the objective is to compare the search abilities of the different algorithms. The intention is therefore to keep the heuristics to a minimum, and the existing pieces of domain knowledge are shared by all the algorithms presented (as in the case of the score function for MO-PTSP, described above).

Adapting MCTS into Multi-Objective Monte Carlo Tree Search (MO-MCTS) requires the obvious modification of dealing with multiple rewards instead of just one. As these are collected at the end of a Monte Carlo simulation, the reward value r now becomes a vector $R = r_0, r_1, \ldots, r_m$, where m is the number of objectives to optimize. Derived from this change, the average value $Q(s, a)$ becomes a vector that stores the average reward of each objective. Note that the other statistics ($N(s, a)$ and $N(s)$) do not need to change, as these are just node and action counters. The important question to answer next is how to adapt the vector $Q(s, a)$ to use it in the UCB1 formula (Equation 2.1).

In the algorithm proposed in this study, the reward vector $\overline{r}$ that was obtained at the end of a Monte Carlo simulation is back-propagated through the nodes visited in the last iteration until the root is reached. In the vanilla algorithm, each node would use this vector $\overline{r}$ to update its own accumulated reward vector $\overline{R}$. In the algorithm proposed here, each node in the MO-MCTS algorithm also keeps a local Pareto front approximation (P), updated at each iteration with the reward vector $\overline{r}$ obtained at the end of the simulation. Algorithm 6 describes how the node statistics are updated in MO-MCTS.

Here, if $\overline{r}$ is not dominated by the local Pareto front, it is added to the front and $\overline{r}$ is propagated to its parent. It may also happen that $\overline{r}$ dominates all or some solutions of the local front, in which case the new solution is included and the dominated solutions are removed from the Pareto front approximation. If $\overline{r}$ is dominated by the node's Pareto front, it does not change and there is no need to maintain this

Algorithm 6 MO-MCTS node update.

```
1: function UPDATE(node, r̄, dominated = false)
2:     node.Visits = node.Visits + 1
3:     node.R̄ = node.R̄ + r̄
4:     if !dominated then
5:         if node.P ⪯ r̄ then
6:             dominated = true
7:         else
8:             node.P = node.P ∪ r̄
9:     UPDATE(node.parent, r̄, dominated)
```

propagation up the tree. Three observations can be made about the mechanism described here:

- Each node in the tree has an estimate of the quality of the solutions reachable from there, both as an average (as in the baseline MCTS) and as the best case scenario (by keeping the non-dominated front P).
- By construction, if a reward $\overline{r}$ is dominated by the local front of a node, it is a given that it will be dominated by the nodes above in the tree, so there is no need to update the fronts of the upper nodes, producing little impact on the computational cost of the algorithm.
- It is easy to infer, from the last point, that the Pareto front of a node cannot be worse than the front of its children (in other words, the front of a child will never dominate that of its parent). Therefore, the root of the tree contains the best non-dominated front ever found during the search. This means that the algorithm tries to approximate the optimal Pareto front of the problem, which aligns it with the multiple-policy category of Multi-Objective approaches, as described in Section 2.4.

This last detail is important for two main reasons. First of all, the algorithm allows the root to store information as to which action to take in order to converge to any specific solution in the front discovered. This information can be used, when all iterations have been performed, to select the move to perform next. If weights for the different objectives are provided, these weights can be used to select the desired solution in the Pareto front approximation of the root node, and hence select the action that leads to that point. Secondly, the root's Pareto front can be used to measure the global quality of the search using the hypervolume (HV, see Section 2.4.2) calculation.

Finally, the information stored at each node regarding the local Pareto front can be used to substitute $Q(s, a)$ in the UCB1 equation. The quality of a given pair (s, a) can be obtained by measuring the HV

of the Pareto front stored in the node reached from state s after applying action a. This can be defined as $Q(s,a) = HV(P)/N(s)$, and the Upper Confidence Bound equation, referred to here as $MO-UCB$, is described as in Equation 8.6:

$$a^* = \text{argmax}_{a \in A(s)} \left\{ HV(P)/N(s) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{8.6}$$

This algorithm, similar to NSGA-II due to their multi-objective nature, provides a non-dominated front as a solution. However, in planning and control scenarios like the games analyzed in this research, an action must be provided to perform a move in the next step. The question that arises is how to choose the move to make based on the information available.

As shown before, it is straightforward to discover which actions lead to what points in the front given as a solution: the first gene in the best NSGA-II individual, a root's child in MO-MCTS. Hence, by identifying the point in the Pareto front that the algorithm should converge to, it is possible to execute the action that leads to that point.

In order to select a point in the Pareto front, a weight vector W can be defined, with a dimension m equal to the number of objectives ($W = (w_1, w_2, \ldots, w_m); \sum_i^m w_i = 1$). Two different mechanisms are proposed, for reasons that will be explained in the experiments section (8.4):

- Weighted Sum: the action chosen is the one that maximizes the weighted sum of the reward vector multiplied by W, for each point in the front.
- Euclidean distance: the euclidean distance from each point in the Pareto front (normalized in $[0,1]$) to the vector W is calculated. The action to choose would be the one that leads to the point in the Pareto front with the shortest distance to W.

Note that, in the vanilla MCTS, there is no Pareto front obtained as a solution. Typically, in this case, rewards are calculated as a weighted sum of the objectives and a weight vector W. The action is then chosen following any of the mechanisms usually employed in the literature: the action taken more often from the root; the one that leads to the best reward found; the move with the highest expected reward; or the action that maximizes the UCB1 Equation 2.1 in the root.

8.4 Experimentation

The experiments performed in this research compare three different algorithms in the three benchmarks presented before: a single objective MCTS (referred to here simply as *MCTS*), the Multi-Objective MCTS (*MO-MCTS*) and a rolling horizon version of the NSGA-II algorithm described in Section 2.4 (*NSGA-II*). This NSGA-II version evolves a population where the individuals are sequences of actions (macro-actions in the PD and MO-PTSP case), obtaining the fitness from the state of the game after applying the sequence. The population sizes, determined empirically, were set to 20, 50 and 50 individuals for DST, PD and MO-PTSP respectively. The value of C in Equations 2.1 and 8.6 is set to $\sqrt{2}$.

All algorithms have a limited number of evaluations before providing an action to perform. In order for these games to be real-time, the time budget allowed is close to 40 milliseconds. With the objective of avoiding congestion peaks at the machine where the experiments are run, the average number of evaluations possible in 40 ms is calculated and employed in the tests. This leads to 4500 evaluations in the DST, and 500 evaluations for MO-PTSP.

8.4.1 Results in DST

As the optimal Pareto front of the DST is known, a measure of performance can be obtained by observing the percentage of times these solutions are found by the players. As the solution that the algorithms converge to depends on the weights vector employed during the search, the approach taken here is to provide different weight vectors W and analyze them separately.

The weight vector for DST has two dimensions. This vector is here referred to as $W = (w_p, w_v)$, where w_p weights moves; and w_v weights the treasure value ($w_v = 1 - w_p$). w_p takes values between 0 and 1, with a precision of 0.01, and 100 runs have been performed for each pair (w_p, w_v). Hence, the game has been played a total of 10000 times, for each algorithm.

Figure 8.3 shows the results obtained after these experiments were performed. The first point to note is that MCTS only converges (mostly) to the two optimal points located at the edges of the optimal Pareto front (*OPT-1* and *OPT-10*, see also Figure 3.5). This is an expected result, as K. Deb. suggested in (Deb, 2001) (and was also discussed before in Section 2.4): linear scalarization approaches only converge to the edges of the optimal Pareto front if its shape is non-convex.

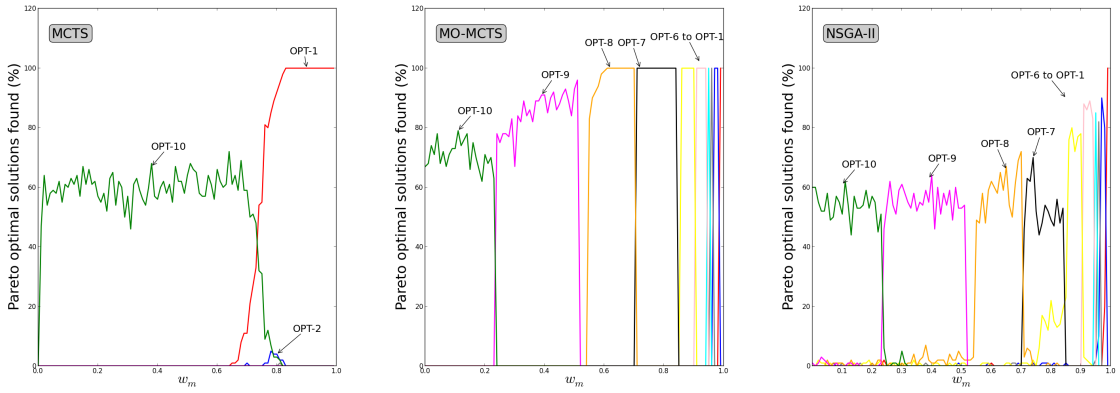


Figure 8.3: *Results in DST: percentages of each optima found during* 100 *games played with different weight vectors. Scalarization approaches converge to the edges of the optimal front, whereas Pareto approaches are able to find all optimal solutions. The proposed algorithm, MO-MCTS, finds these solutions significantly more often than NSGA-II.*

The results show clearly how approximating the optimal Pareto front allows for finding all possible solutions. Both the NSGA-II and MO-MCTS approaches are able to converge to any solution in the front given the appropriate weight vector. It is important to highlight that these two algorithms employed the Euclidean distance action selection. Other experiments, not included in this study, showed that weighted sum action selection provides similar results to MCTS (that is, convergence to the edges of the front). The crucial distinction to make here is that both algorithms, NSGA-II and MO-MCTS, allow for better action selection mechanisms, which are able to overcome this problem by approximating a global Pareto front.

Finally, in the comparison between NSGA-II and MO-MCTS, the latter algorithm obtains higher percentages for each one of the points in the front. This result suggests that, with a limited number of iterations/evaluations, the proposed algorithm is able to explore the search space more efficiently than a rolling horizon version of a state-of-the-art NSGA-II.

8.4.2 Results in Puddle Driver

The optimal Pareto front for this problem is unknown, so the focus in these experiments is on how different weights provide different solutions, and the comparison between the results obtained by each one of the algorithms presented here.

In order to evaluate the performance of these algorithms in the Puddle Driver, 10 different maps are created with 25 puddle squares, of 25×25 pixels, uniformly randomly distributed in them. Each algorithm is executed 10 times on each map, providing a total of 100 executions for each one of the

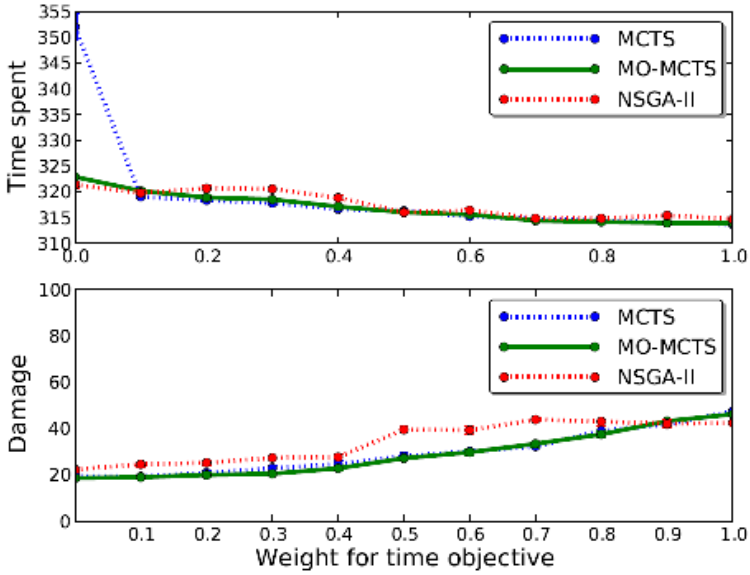


Figure 8.4: *Average rewards with standard error against weights for first objective. Smaller values are better.*

weights employed. For this problem, the weights range from 0 to 1 in increments of 0.1.

Both algorithms execute $N = 8$ macro-actions per evaluation (i.e. in NSGA-II, the length of the individual's genome is 8), where the macro-action length is $L = 15$, leading to $8 \times 15 = 120$ single actions applied to evaluate the individual.

Figure 8.4 shows the average rewards (with standard error) obtained by the algorithms for each one of the tested weight values. As can be seen, the total time spent (upper sub-figure from Figure 8.4) decreases as long as this objective obtains a higher reward, while the opposite happens with the damage taken in puddle regions (lower plot). In most cases, MCTS and MO-MCTS obtain better rewards than NSGA-II, although both MCTS algorithms obtain very similar results (with the exception of MCTS expending more time when this objective is given a weight of 0). This similarity may be due to the simplicity of the fitness landscape, suggesting that in some cases a weighted sum approach can provide a performance on a par with the MO-MCTS algorithm proposed here.

8.4.3 Results in MO-PTSP

The same three algorithms have been tested in the MO-PTSP domain. The experiments are performed in the 10 maps available within the framework of the competition (see Appendix A). In this case, the optimal Pareto front is not known in advance, and normally differs from one map to another.

The performance of the algorithms is compared as follows. First of all, 4 different weight vectors are

tested: $W_1 = (0.33, 0.33, 0.33)$, $W_2 = (0.1, 0.3, 0.6)$, $W_3 = (0.1, 0.6, 0.3)$ and $W_4 = (0.6, 0.1, 0.3)$, where each w_i corresponds to the weight for an objective (w_t for time, w_f for fuel and w_d for damage, in this order). W_1 treats all objectives as having the same weight, while the other three give more relevance to different objectives. These vectors then provide a wide spectrum for weights in this benchmark. In this case, MO-MCTS uses these weights to select an action based on a weighted sum, which in preliminary experiments has shown better performance than the Euclidean distance mechanism.

The first point to check is if the different weight vectors affect the solutions obtained by MO-MCTS. Table 8.1 shows the results of executing the MO-MCTS controller during 30 games in each one of the 10 maps of the MO-PTSP, for every weight vector. This table includes the averages and standard errors obtained. Results in bold are those with an independent t-test p-value smaller than 0.01, comparing the results of the objective with the highest weight with the result from the default weight vector $W_1 = (0.33, 0.33, 0.33)$ in the same map. It can be seen that the highest weight in W leads, in most of the cases, to the best solution in that objective in the map.

Some exceptions can be explained by analysing the specific maps: in map 1, a fuel canister is always collected near the end, restoring the fuel level to its maximum. This leaves too few cycles to make a difference in the controller (note also that this is the map with the smallest fuel consumption). Also, in map 10, there is no difference in the damage objective, however map 10 is a map with no obstacles to damage the ship (thus only lava lakes deal damage). This also results in this map being the one with the lowest overall damage. It can also be seen that, in those maps where time has priority, the results for this objective are not as dominant as the other two. This can be explained by the fact that the time heuristic is actually part of the fuel and damage heuristics (term ρ_t from Equation 8.3, present in Equations 8.4 and 8.5).

These results suggest that the weights effectively work on making the algorithm converge to different points in the Pareto front discovered by the algorithm. Now, it is time to compare these results with the other algorithms. In order to do this, the same number of runs is performed for NSGA-II and MCTS in the 10 maps of the benchmark.

In order to compare the several algorithms, the results are examined in pairs, in terms of dominance. The procedure is as follows: once all games on a single map have been run, the Mann-Whitney-Wilcoxon non-parametric test with 95% confidence is calculated on the three objectives. If the measures on all

Map	$W : (w_t, w_f, w_d)$	Time	Fuel	Damage
Map 1	$W_1 : (0.33, 0.33, 0.33)$	1654 (7)	131 (2)	846 (13)
	$W_2 : (0.1, 0.3, 0.6)$	1657 (8)	130 (2)	**773(11)**
	$W_3 : (0.1, 0.6, 0.3)$	1681 (11)	131 (2)	837 (15)
	$W_4 : (0.6, 0.1, 0.3)$	1649 (8)	132 (2)	833 (13)
Map 2	$W_1 : (0.33, 0.33, 0.33)$	1409 (7)	235 (4)	364 (3)
	$W_2 : (0.1, 0.3, 0.6)$	1402 (6)	236 (5)	**354(2)**
	$W_3 : (0.1, 0.6, 0.3)$	1416 (8)	**219(4)**	360 (3)
	$W_4 : (0.6, 0.1, 0.3)$	1396 (8)	245 (5)	361 (2)
Map 3	$W_1 : (0.33, 0.33, 0.33)$	1373 (6)	221 (3)	301 (7)
	$W_2 : (0.1, 0.3, 0.6)$	1378 (5)	211 (3)	**268(5)**
	$W_3 : (0.1, 0.6, 0.3)$	1385 (6)	**203(4)**	291 (7)
	$W_4 : (0.6, 0.1, 0.3)$	1363 (4)	229 (4)	285 (7)
Map 4	$W_1 : (0.33, 0.33, 0.33)$	1383 (6)	291 (5)	565 (5)
	$W_2 : (0.1, 0.3, 0.6)$	1385 (7)	304 (4)	**542(4)**
	$W_3 : (0.1, 0.6, 0.3)$	1423 (6)	**273(4)**	583 (5)
	$W_4 : (0.6, 0.1, 0.3)$	1388 (6)	309 (4)	559 (5)
Map 5	$W_1 : (0.33, 0.33, 0.33)$	1405 (7)	467 (4)	559 (4)
	$W_2 : (0.1, 0.3, 0.6)$	1431 (9)	447 (4)	**541(4)**
	$W_3 : (0.1, 0.6, 0.3)$	1467 (9)	**411(5)**	567 (5)
	$W_4 : (0.6, 0.1, 0.3)$	1399 (9)	469 (4)	547 (3)
Map 6	$W_1 : (0.33, 0.33, 0.33)$	1575 (7)	549 (5)	303 (4)
	$W_2 : (0.1, 0.3, 0.6)$	1626 (9)	540 (6)	286 (5)
	$W_3 : (0.1, 0.6, 0.3)$	1703 (11)	**499(4)**	316 (7)
	$W_4 : (0.6, 0.1, 0.3)$	1571 (7)	559 (5)	294 (4)
Map 7	$W_1 : (0.33, 0.33, 0.33)$	1434 (5)	599 (6)	284 (6)
	$W_2 : (0.1, 0.3, 0.6)$	1475 (10)	602 (5)	243 (6)
	$W_3 : (0.1, 0.6, 0.3)$	1489 (12)	**549(3)**	264 (6)
	$W_4 : (0.6, 0.1, 0.3)$	**1407(8)**	618 (5)	270 (6)
Map 8	$W_1 : (0.33, 0.33, 0.33)$	1761 (9)	254 (5)	382 (3)
	$W_2 : (0.1, 0.3, 0.6)$	1804 (10)	269 (4)	**357(4)**
	$W_3 : (0.1, 0.6, 0.3)$	1826 (10)	230 (3)	392 (7)
	$W_4 : (0.6, 0.1, 0.3)$	1732 (9)	311 (8)	379 (6)
Map 9	$W_1 : (0.33, 0.33, 0.33)$	2501 (14)	926 (6)	574 (9)
	$W_2 : (0.1, 0.3, 0.6)$	2503 (10)	921 (10)	**524(8)**
	$W_3 : (0.1, 0.6, 0.3)$	2641 (14)	**833(5)**	574 (14)
	$W_4 : (0.6, 0.1, 0.3)$	2470 (9)	956 (5)	573 (8)
Map 10	$W_1 : (0.33, 0.33, 0.33)$	1430 (8)	630 (4)	205 (2)
	$W_2 : (0.1, 0.3, 0.6)$	1493 (13)	615 (4)	209 (2)
	$W_3 : (0.1, 0.6, 0.3)$	1542 (10)	**554(4)**	229 (5)
	$W_4 : (0.6, 0.1, 0.3)$	**1378(5)**	663 (6)	202 (4)

Table 8.1: MO-PTSP averages (plus standard error) with different weight vectors. Results in bold obtained an independent t-test p-value < 0.01.

objectives are assumed to be drawn from different distributions, their averages are compared for a dominance test (if not, then no dominance relationship can be derived from the results). If one result dominates another, then the algorithm dominates the other in that particular map.

Extending this comparison to all maps, each pair of algorithms ends with a triplet $(D, \emptyset, d)$. D is the number of maps where the first algorithm dominates the second. $\emptyset$ is the amount of maps where no dominance can be established, either because the Mann-Whitney-Wilcoxon non-parametric

	$W : (w_t, w_f, w_d)$	**MO-MCTS** $(D, \emptyset, d)$	**MCTS** $(D, \emptyset, d)$	**NSGA-II** $(D, \emptyset, d)$	**PurofMovio** $(D, \emptyset, d)$
MO-MCTS	$W_1 : (0.33, 0.33, 0.33)$	–	$(8, 2, 0)$	$(8, 2, 0)$	$(0, 5, 5)$
	$W_2 : (0.1, 0.3, 0.6)$		$(10, 0, 0)$	$(4, 6, 0)$	$(0, 6, 4)$
	$W_3 : (0.1, 0.6, 0.3)$		$(8, 2, 0)$	$(7, 3, 0)$	$(0, 5, 5)$
	$W_4 : (0.6, 0.1, 0.3)$		$(10, 0, 0)$	$(3, 7, 0)$	$(0, 3, 7)$
MCTS	$W_1 : (0.33, 0.33, 0.33)$	$(0, 8, 2)$	–	$(0, 2, 8)$	$(0, 2, 8)$
	$W_2 : (0.1, 0.3, 0.6)$	$(0, 0, 10)$		$(4, 2, 4)$	$(0, 3, 7)$
	$W_3 : (0.1, 0.6, 0.3)$	$(0, 2, 8)$		$(0, 1, 9)$	$(0, 6, 4)$
	$W_4 : (0.6, 0.1, 0.3)$	$(0, 0, 10)$		$(3, 3, 4)$	$(0, 1, 9)$
NSGA-II	$W_1 : (0.33, 0.33, 0.33)$	$(0, 2, 8)$	$(8, 2, 0)$	–	$(0, 4, 6)$
	$W_2 : (0.1, 0.3, 0.6)$	$(0, 6, 4)$	$(4, 2, 4)$		$(0, 4, 6)$
	$W_3 : (0.1, 0.6, 0.3)$	$(0, 3, 7)$	$(9, 1, 0)$		$(0, 5, 5)$
	$W_4 : (0.6, 0.1, 0.3)$	$(0, 7, 3)$	$(4, 3, 3)$		$(0, 4, 6)$
PurofMovio	$W_1 : (0.33, 0.33, 0.33)$	$(5, 5, 0)$	$(8, 2, 0)$	$(6, 4, 0)$	–
	$W_2 : (0.1, 0.6, 0.3)$	$(4, 6, 0)$	$(7, 3, 0)$	$(6, 4, 0)$	
	$W_3 : (0.1, 0.3, 0.6)$	$(5, 5, 0)$	$(6, 4, 0)$	$(5, 5, 0)$	
	$W_4 : (0.6, 0.1, 0.3)$	$(7, 3, 0)$	$(9, 1, 0)$	$(6, 4, 0)$	

Table 8.2: Results in MO-PTSP: Each cell indicates the triplet $(D, \emptyset, d)$, where D is the number of maps where the row algorithm dominates the column one, $\emptyset$ is the amount of maps where no dominance can be established, and d states the number of maps where the row algorithm is dominated by the column one. All algorithms followed the same route (order of waypoints and fuel canisters) in every map tested.

test returned no significant difference, or because there is no dominance according to the dominance rules described in Section 2.4. Finally, d states the number of maps where the first is dominated by the second. For example, a triplet $(D, \emptyset, d) = (8, 2, 0)$ comparing algorithms A and B would mean that the results obtained by A dominate those from B in 8 of the 10 maps, and that it is not possible to derive any dominance in the other 2. Table 8.2 summarizes these results for all algorithms tested.

One of the first things to notice is that MO-MCTS dominates MCTS and NSGA-II in most of the maps, and it is never dominated in any. In particular, the dominance of MO-MCTS over MCTS is outstanding, even dominating in all 10 maps for two of the weight vectors. MO-MCTS also dominates NSGA-II in more maps than in those where there is no dominance, and is never dominated by NSGA-II in any map.

It is also interesting to see that NSGA-II dominates the weighted sum version of MCTS, although for the vector W_2 there is a technical draw, as they dominate each other in 4 different maps each and there is no dominance in the other 2.

Additionally, Table 8.2 contains a fourth entry, `PurofMovio`, that the algorithms are compared against. `PurofMovio` was the winning entry of the 2013 MO-PTSP competition, a controller based on a weighted-sum MCTS approach (see (Powley, Whitehouse, and Cowling, 2013) for details of its

implementation). As can be seen, `PurofMovio` obtains better results than the algorithm proposed here.

However, it is very important to highlight that `PurofMovio` is not using the same heuristics as the ones presented in this research. Hence, nothing can be concluded from making pairwise comparisons directly with the winning entry of the competition. It is likely that `PurofMovio`'s heuristics are more efficient than the ones presented here, but the goal of this study is not to develop the best possible heuristics for the MO-PTSP, but to provide an insight into how a multi-objective version of MCTS compares to other algorithms using the same heuristics.

Nonetheless, the inclusion of `PurofMovio` in this comparison is not pointless: it is possible to assess the quality of the three algorithms tested here by comparing their performance relatively, against this high quality entry. Attending to this criteria, it can be seen how MO-MCTS is the algorithm that is dominated less often by `PurofMovio`, producing similar results on an average of 4.75 out of the 10 maps, and being dominated in 5.25 maps. MCTS and NSGA-II are dominated more often than MO-MCTS, being dominated in an average of 7.5 and 5.75 of the maps, respectively. This comparative result shows again that MO-MCTS is achieving the best results among the three algorithms compared here.

8.4.4 A step further in MO-PTSP: segments and weights

There is another aspect that can be further improved in the MO-PTSP benchmark, and is also applicable to other domains. It is naive to think that a unique weight vector will be the ideal one for the whole game. Specifically in the MO-PTSP, there are regions of the map where there are more obstacles or lava lakes, hence the ship is most likely to suffer higher damage there. Also, the route followed during the game affects the relative ideal speed between waypoints, or perhaps a fuel canister may be picked up, which will affect how the fuel objective will be managed. In general, many real-time games go through different phases, with different objectives and priorities.

A way to provide different weights at different times in MO-PTSP is straightforward. Given the route of waypoints (and fuel canisters) being followed, one can divide it into *segments*, where each segment starts and ends with a waypoint (or fuel canister). Then, each segment can be assigned a particular weight vector W.

The question is then how to assign these weight vectors. Three different ways can be devised:

- Manually set the weight vectors. This was attempted and it proved to be a non trivial task.

- Setting the appropriate weight for each segment dynamically, based on the segment's characteristics. This involves the creation or discovery of features and some kind of function approximation to assign the values.

- Learn, for each specific map, the combination of weight vectors that produces better results.

This section shows some initial results obtained when testing the third variant, using a stochastic hill climbing algorithm on each map. The goal is to check if, by varying the weight vectors between segments, better solutions can be achieved.

An individual is identified by a string of integers, where each integer refers to one of the weight vectors utilized in the previous sections ($1 = W_1 = (0.33, 0.33, 0.33)$, $2 = W_2 = (0.1, 0.3, 0.6)$ and $3 = W_3 = (0.1, 0.6, 0.3)$. W_4 has been left out of this experiment, as it has proven to be the least influential weight vector). The solution is evaluated playing a particular map 10 times, and its fitness is obtained by calculating the average of those runs. A population of 10 individuals is kept, and the solutions of the initial population are created either randomly, or mutated from base individuals. These base individuals all have segments with the same weight vector W_1, W_2, or W_3.

The best solution, determined by dominance, is kept and promoted to the next generation, where it is mutated to generate other individuals of the population. Also, a portion of the individuals of the next population is created uniformly at random at every generation, until the end of the algorithm, which is established at 50 generations.

Table 8.3 shows the results obtained on each run, one per map. Each row corresponds to a run in the associated map, and it provides different results depending on the weights vector. The top three are the base individuals, taken from Table 8.2 for comparison. The fourth result on each row is the best one after the run. Each genome is a string of the form $abc \dots z$, that represents $W_a W_b W_c \dots W_z$, where each element is a weight vector used in that particular segment. The last column indicates if the evolved individual dominates ($\preceq$) or not ($\emptyset$) each one of the base genomes for that particular map.

The results suggest some interesting ideas. First of all, it is indeed possible to obtain better results by varying the weights of the objectives along the route: in 22 out of the 30 comparisons made, the evolved solution dominates the base individuals and, in the other 8, both solutions in the comparison would be in the same Pareto front of solutions.

Map	Weight genome	Time	Fuel	Damage	D
Map 1	11111111111111	1654 (7)	131 (2)	846 (13)	$\preceq$
	22222222222222	1657 (8)	130 (2)	773 (11)	$\preceq$
	33333333333333	1681 (11)	131 (2)	837 (15)	$\preceq$
	32312212331112	1619 (12)	130 (2)	744 (15)	
Map 2	11111111111111	1409 (7)	235 (4)	364 (3)	$\preceq$
	22222222222222	1402 (6)	236 (5)	354 (2)	$\preceq$
	33333333333333	1416 (8)	219 (4)	360 (3)	$\preceq$
	23131312323213	1390 (10)	210 (3)	353 (3)	
Map 3	11111111111111	1373 (6)	221 (3)	301 (7)	$\preceq$
	22222222222222	1378 (5)	211 (3)	268 (5)	$\preceq$
	33333333333333	1385 (6)	203 (4)	291 (7)	Ø
	11122222112231	1358 (9)	219 (7)	263 (12)	
Map 4	11111111111111	1383 (6)	291 (5)	565 (5)	$\preceq$
	22222222222222	1385 (7)	304 (4)	542 (4)	$\preceq$
	33333333333333	1423 (6)	273 (4)	583 (5)	Ø
	11121131212112	1360 (4)	282 (5)	540 (4)	
Map 5	11111111111111	1405 (7)	467 (4)	559 (4)	$\preceq$
	22222222222222	1431 (9)	447 (4)	541 (4)	$\preceq$
	33333333333333	1467 (9)	411 (5)	567 (5)	Ø
	21311213111211	1397 (11)	448 (10)	535 (5)	
Map 6	11111111111111	1575 (7)	549 (5)	303 (4)	$\preceq$
	22222222222222	1626 (9)	540 (6)	286 (5)	$\preceq$
	33333333333333	1703 (11)	499 (4)	316 (7)	Ø
	31121312111111	1570 (16)	535 (10)	266 (6)	
Map 7	11111111111111	1434 (5)	599 (6)	284 (6)	$\preceq$
	22222222222222	1475 (10)	602 (5)	243 (6)	$\preceq$
	33333333333333	1489 (12)	549 (3)	264 (6)	Ø
	11332211321332	1401 (12)	563 (4)	230 (12)	
Map 8	11111111111111	1761 (9)	254 (5)	382 (3)	$\preceq$
	22222222222222	1804 (10)	269 (4)	357 (4)	$\preceq$
	33333333333333	1826 (10)	230 (3)	392 (7)	Ø
	23221131313323	1747 (11)	247 (9)	363 (9)	
Map 9	11111111111111	2501 (14)	926 (6)	574 (9)	$\preceq$
	22222222222222	2503 (10)	921 (10)	524 (8)	$\preceq$
	33333333333333	2641 (14)	833 (5)	574 (14)	Ø
	21132331333223	2463 (19)	891 (8)	523 (9)	
Map 10	11111111111111	1430 (8)	630 (4)	205 (2)	$\preceq$
	22222222222222	1493 (13)	615 (4)	209 (2)	$\preceq$
	33333333333333	1542 (10)	554 (4)	229 (5)	Ø
	11311111322231	1418 (9)	623 (9)	197 (2)	

Table 8.3: MO-PTSP Results with different weights. The last column indicates if the evolved individual dominates ($\preceq$) or not (Ø) each one of the base genomes for that particular map.

If the focus is set on what particular weights are more successful, it is interesting to see that the best configurations found are better, in all maps, than the base individuals with all weights equal to W_1 ($w_t = 0.33, w_f = 0.33, w_d = 0.33$) and W_2 ($w_t = 0.1, w_f = 0.3, w_d = 0.6$). In other words, it was always possible to find a combination of weight vectors that performed better than approaches that gave the same weight to all objectives, and also better than the ones that prioritized low damage.

It is also worthwhile mentioning that in those cases where dominance over the base individual was

not achieved, the base was the one that prioritized fuel. Actually, it can be seen that in these cases, the result in the fuel objective is the one that prevents the dominance from happening, obtaining a better value than the evolved solution in this particular objective.

8.5 Conclusions

The results obtained in this study show the strength of the algorithm proposed, as it provides better solutions than the ones obtained by the other algorithms in comparison. Only in the case of PD do MCTS and MO-MCTS obtain similar results, suggesting that in some simple cases a weighted sum approach can also obtain positive results. The MO-MCTS approach samples successfully across the different solutions in the Pareto front (optimal front in the DST, the best found in PD and MO-PTSP), depending on the weights provided for each objective. Finally, some initial results are obtained applying the idea of using distinct weight vectors for the objectives in different situations within the game, showing that it is possible to improve the performance of the algorithms.

The stochastic hill climbing algorithm used for creating strings of weight vectors for the MO-PTSP is relatively simple, although it was able to obtain very good results in few iterations. However, it could be desirable to model behaviours that dynamically change the weight vectors according to measurements of the game state, instead of evolving a different solution for each particular map. A possible extension for this work is to analyse and discover features in the game state that allow the establishment of relationships between game situations and the weight vectors to use for each objective. This mechanism would be general enough to be applicable to many different maps without requiring specific learning in any particular level.

Finally, the results obtained in the experiments described in this chapter allows us to assume that multi-objective approaches can provide a high quality level of play in real-time games. Multi-objective problems pop up in different settings and by using MCTS we can balance not only between exploration and exploitation but between multiple objectives as well.

"I'm a machine! And I can know much
more! I can experience so much more.
But I'm trapped in this absurd body! . . . "
- Brother Cavil, Battlestar Galactica

Chapter 9

MCTS for General Video Game Playing

This chapter extends the research to the field of General Video Game playing. Most of the material in this chapter has been accepted for publication in the paper (Perez, Samothrakis, and Lucas, 2014c).

9.1 Introduction

In most games used by researchers, there exists the possibility of adding a significant amount of domain knowledge that helps the algorithms tested to achieve the game goals. Examples can be found in previous chapters of this thesis, where heuristics were used to guide the Monte Carlo simulations and/or score the states at the end of a roll-out.

Whilst not considered 'incorrect', the use of domain-dependent heuristics can raise doubts as to whether the type of knowledge introduced is better suited to only certain of the algorithms under comparison, or even cast doubt on how much of the success of an algorithm can be attributed to the algorithm itself or to the heuristics employed. It could even be possible to argue that, in spite of applying the same heuristics in all algorithms, the comparison could still be unbalanced in certain cases, as some heuristics could benefit certain algorithms more than others. When comparing algorithms, it may possibly be fairer to allow each algorithm to use its best possible heuristic. But should we not then compare the quality of each heuristic separately as well?

The goal of General Video-Game Playing (GVGP, see Section 2.5) is to by-pass the addition of game specific knowledge, especially if the algorithm is tested in games that have not been played before. The objective of this chapter is to study how vanilla MCTS struggles in this scenario (i.e. in the absence of domain-dependent knowledge) and to propose certain game-agnostic heuristics that allow the agent

to discover and gain knowledge about its environment. For this, the experiments conducted in this chapter use the General Video Game AI (GVGAI) games and framework, as described in Section 3.6.

In this scenario, it is therefore essential to learn as much as possible from the roll-outs performed by the algorithm. Several approaches have been tried to achieve this in the literature, as in (Robles, Rohlfshagen, and Lucas, 2011), where the authors employed Temporal Difference Learning to learn a linear function approximator. This was used as an heuristic to bias the move selection during the Monte Carlo simulations in the game Othello. The authors found that the proposed approach obtained better results than using uniformly random roll-outs.

Another interesting example is the Move-Average Sampling Technique (MAST), introduced by Finnsson and Björnsson (Finnsson and Björnsson, 2008) for General Game Playing, where the average reward of applying certain actions are computed independently from the state from which these are played. Later, Finnsson (Finnsson and Björnsson, 2010) also proposed the Feature-Average Sampling Technique (FAST), employing TD(λ) to learn the relative importance of the features extracted from the game definition, used subsequently to modify the value of $Q(s, a)$. His work showed that learning features considerably improved the performance of the algorithm over a random simulation policy.

Several authors (Benbassat and Sipper, 2013; Alhejali and Lucas, 2013) have evolved heuristics, in an offline manner, to bias roll-outs in MCTS. More recently, Lucas et al. (Lucas, Samothrakis, and Perez, 2014) proposed Fast Evolutionary MCTS, a method to guide roll-outs via online evolution. An important element of the algorithm proposed in this chapter is based on Fast Evolutionary MCTS, using evolution to adapt to the environment and increase performance. In this approach, a vector of weights w is evolved online to bias the Monte Carlo simulations, using a fixed set of features extracted for the current game state. Experiments performed in the Mountain Car problem and in the game Space Invaders showed an increase in performance of the algorithm.

The idea behind Fast Evolutionary MCTS is to embed the algorithm roll-outs within evolution. Every roll-out evaluates a single individual of the evolutionary algorithm, providing as fitness the reward calculated at the end of the roll-out. Its pseudocode can be seen in Algorithm 7.

The call in line 4 retrieves the next individual, or weight vector w, to evaluate, while its fitness is set in line 9. The vector w is used to bias the roll-out (line 7), following the next process: mapping from state space S to feature space F. A number of N features are extracted on each state found during the

Algorithm 7 Fast Evolutionary MCTS Algorithm, from (Lucas et al., 2014), assuming one roll-out per fitness evaluation.

Input: v_0 root state.
Output: weight vector w, action a
while within computational budget **do**
    w = Evo.GetNext()
    Initialize Statistics Object S
    v_l = TreePolicy(v_0)
    δ = DefaultPolicy($s(v_l), D(w)$)
    UpdateStats(S, δ)
    Evo.SetFitness(w, S)
return w = Evo.getBest()
return a = recommend(v_0)

roll-out. Given a set of A available actions, the relative strength of each action (a_i) is calculated as a weighted sum of feature values, as shown in Equation 9.1.

$$a_i = \sum_{j=1}^{N} w_{ij} \times f_j; \tag{9.1}$$

The weights, initialized at every game step, are stored in a matrix W, where each entry w_{ij} is the weighting of feature j for action i. These relative action strengths are introduced into a softmax function in order to calculate the probability of selecting each action (see Equation 9.2). For more details about this algorithm, the reader is referred to (Lucas et al., 2014).

$$P(a_i) = \frac{e^{-a_i}}{\sum_{j=1}^{A} e^{-a_j}} \tag{9.2}$$

This chapter proposes an extension of this work, by first employing any number of features, and then dynamically creating a knowledge base that is used to better calculate the reward of a given state. Section 9.2 analyzes the problems found when applying vanilla MCTS to GVGP. Later, Sections 9.3 and 9.4 present the algorithm proposed, tested in the experiments shown in Section 9.5. Finally, Section 9.6 draws some conclusions on this research.

9.2 Vanilla MCTS Controller: analysis

The vanilla MCTS algorithm features in the GVGAI framework as a sample controller. For this controller, the simulation (or play-out) depth of the tree is 10 actions, $C = \sqrt{2}$ and the score of each state is calculated as a function of the game score, normalized between the minimum and maximum

scores ever seen during the play-outs. In case the game is won or lost in a given state, the reward is a large positive or negative number, respectively.

Although detailed results of the sample controller are reported later in Section 9.5, it is important to analyze first why this controller only achieves a 21.6% victory rate.

Initially, the obvious reason for this low rate of success is that the algorithm has no game specific information to bias the roll-outs, and the rewards depend only on the score and outcome of the game. Additionally, due to the real-time nature of the framework, roll-outs are limited in depth and therefore the vast majority of the play-outs do not reach an end game state, preventing the algorithm from finding winning states. This is, however, a problem that is not possible to avoid: play-out depth is always going to be limited, and the Monte Carlo simulations cannot use any game specific information to bias the actions taken in this domain.

There is another important problem that MCTS faces continuously when playing general video-games: in its vanilla form, MCTS has no way of reusing past information from events that provided some score gain in the past, even during the same game. Imagine that in the game *Butterflies* (see Table 3.1), the avatar has successfully captured some butterflies in any area of the level. Here, MCTS learned that certain actions (that caused the avatar to collide with a butterfly) provided a boost in the score, indirectly learning that colliding with butterflies was a good thing to do. However, these learnt facts are not used again to drive the agent to capture other butterflies that are beyond the horizon reachable by MCTS during the roll-outs.

Furthermore, there is a second important problem: some sprites in the game are never reached by the algorithm's simulations. Imagine the game of *Zelda*, where the avatar must collect a key before exiting the level to win the game. If the key is beyond the simulation horizon (in this case, if it is more than 10 actions away), there is no incentive for the agent to collect it. Note that, if we were programming an agent that would only play this game, an obvious heuristic would be to reduce the distance to the key. But in general video-game playing there is no way to know (*a priori*) what sprites should be targeted by the avatar in the first place.

The next section focuses on a proposed solution to these problems, by providing the algorithm with a knowledge base, and biasing the Monte Carlo simulations to maximize the knowledge gain obtained during the play-outs.

9.3 Fast Evolutionary MCTS

The first step on the new algorithm consists of integrating MCTS with evolution to guide the roll-outs, as in (Lucas et al., 2014). In order to do this, it is important to first identify the features of the game state that will be used by Fast Evolutionary MCTS. In this research, the features extracted from each game state are the Euclidean distances to the closest NPC, resource, non-static object and portal (as described in Section 3.6.2).

Note that each one of these features may be composed of more than one distance, as there might be more than one type of NPC, resource, portal, etc. For instance, in the game *Chase*, the first feature would return two distances: to the closest scared and to the closest angry goat. The same amount of features will not be available in every game step of the same game, as there could be sprites that do not exist at some point of the game: such as enemy NPCs that have not been spawned yet, or depleted resources.

Therefore, the number N of features not only varies from game to game, but also varies from game step to game step. Fast Evolutionary MCTS must therefore be able to adapt the weight vector according to the present number of features at each step. While in the original Fast Evolutionary MCTS algorithm the number of features was fixed, here the evolutionary algorithm maps each feature to a particular position in the genome, increasing the length of the individual every time a new feature is discovered.

In this research, as in the original Fast Evolutionary MCTS paper, the evolutionary algorithm used is a $(1 + 1)$ Evolution Strategy (ES). Albeit a simple choice, it produces good results.

9.4 Knowledge-based Fast Evolutionary MCTS

Now that there is a procedure in place to guide the roll-outs, the next step is to define a score function that provides solutions to the problems analyzed at the end of Section 9.2. Note that this score function is also the one that defines the fitness for the evolved weight vector w, and ultimately will affect how the roll-outs are to be biased.

In this domain, we define the concept *knowledge base* as the combination of two factors: *curiosity* plus *experience*. In this context, the former term refers to discovering the effects of colliding with other

sprites. The concept of *artificial curiosity* has been explored intensively by J. Schmidhuber (Schmidhuber, 2006) in the domain of beauty, creativity and music. In his work, the author employs a reinforcement learning controller to improve the predictive model of the agent in the absence of an external reward. More recently, Susanne Still and Doina Precup (Still and Precup, 2012) explored the concept of information gain and discovery in reinforcement learning algorithms.

The second concept, re-use of past *experience*, allows the agent to reward those events that provided a score gain in the past. This is an event-driven heuristic used in the state evaluation. It somewhat resembles certain MCTS enhancements, such as Rapid Action Value Estimation (RAVE, (Gelly and Silver, 2007)), where the estimated value of an action taken at any level of depth in the tree is updated with the final play-out reward. In the case presented here, however, it is the estimated value of each event, occurring at any point during a roll-out, that is updated and subsequently used to evaluate the final state.

In the research presented in this chapter, each piece of knowledge (*knowledge item*) corresponds to one event that, as defined in Section 3.6.2, represents the collision of the avatar - or a sprite produced by the avatar - with another sprite. Specifically, the knowledge base contemplates only the types of sprites used to extract the features (NPC, resource, non-static object and portal). Each one of these items maintains the following statistics:

- Z_i: number of occurrences of the event i.
- $\overline{x_i}$: average of the score change, calculated as the difference between the game score before and after the event took place. It is important to note that an event does not contain information about the proper score change, this needs to be inferred by the controller. As multiple simultaneous events can trigger a score change, the larger the value of Z_i, the more reliable $\overline{x_i}$ will be.

These statistics are updated every time MCTS makes a move in a roll-out. When each roll-out finishes, the following three values are calculated in the final state:

- Score change ΔR: the difference of the game score between the score value at the beginning and at the end of the play-out.
- Knowledge change $\Delta Z = \sum_{i=1}^{N} \Delta(K_i)$: a measure of *curiosity* that values the change of all Z_i in the knowledge base, for each knowledge item i. $\Delta(K_i)$ is calculated as shown in Equation 9.3,

where Z_{i0} is the value of Z_i at the beginning of the play-out and Z_{iF} is the value of Z_i at the end of the roll-out.

$$\Delta(K_i) = \begin{cases} Z_{iF} & : Z_{i0} = 0 \\ \frac{Z_{iF}}{Z_{i0}} - 1 & : \text{Otherwise} \end{cases} \tag{9.3}$$

Essentially, ΔZ will be higher when the roll-outs produce more events. Events that have been rarely seen before will provide higher values, rewarding knowledge gathering from events less triggered in the past.

- Distance change $\Delta D = \sum_{i=1}^{N} \Delta(D_i)$: a measure of change in distance to each sprite of type i. Equation 9.4 defines the value of $\Delta(D_i)$, where D_{i0} is the distance to the closest sprite of type i at the beginning of the play-out, and D_{iF} is the same distance at the end of the roll-out.

$$\Delta(D_i) = \begin{cases} 1 - \frac{D_{iF}}{D_{i0}} & : Z_{i0} = 0 \text{ OR} \\ & \quad D_{i0} > 0 \text{ and } \overline{x_i} > 0 \\ 0 & : \text{Otherwise} \end{cases} \tag{9.4}$$

Here, ΔD will be higher if the avatar, in the course of the roll-out, has reduced the distance from unknown sprites (again, measuring *curiosity*), or from those that provided a score boost in the past (*experience*).

Once these three values have been calculated, the final score for the game state reached at the end of the roll-out is obtained as in Equation 9.5. Essentially, the reward will be the score difference ΔR, unless $\Delta R = 0$. If this happens, none of the actions during the roll-out were able to change the score of the game, and the reward refers to the other two components. The values of $\alpha = 0.66$ and $\beta = 0.33$ have been determined empirically for this research.

$$Reward = \begin{cases} \Delta R & : \Delta R \neq 0 \\ \alpha \times \Delta Z + \beta \times \Delta D & : \text{Otherwise} \end{cases} \tag{9.5}$$

To summarize, the new score function prioritizes the actions that lead to a score gain in the MCTS iterations. However, if there is no score gain, more reward will be given to the actions that provide

more information to the knowledge base, or that will get the avatar closer to sprites that, by colliding with them in the past, seemed to produce a positive score change.

9.5 Experimental work

The experimental work of this study has been performed on the 10 games explained in Table 3.1. There are five different levels for each one of the games, with variations on the location of the sprites and, sometimes, with slightly different behaviours of the NPC sprites. The complete set of games and levels can be downloaded from the competition website (Perez et al., 2014d). Each one of the levels is played 5 times, giving a total of 250 games played for each configuration tested. Four different algorithms have been explored in the experiments:

- **Vanilla MCTS**: the sample MCTS controller from the competition, as explained in Section 9.2.
- **Fast-Evo MCTS**: Fast Evolutionary MCTS, as per Lucas et al. (Lucas et al., 2014), using dynamic adaptation of the number of features, as explained in Section 9.1.
- **KB MCTS**: Knowledge-based (KB) MCTS as explained in Section 9.4, but employing uniformly random roll-outs in the Simulation phase of MCTS (i.e., no Fast Evolution is used to guide the Monte Carlo simulations).
- **KB Fast-Evo MCTS**: Knowledge-based Fast Evolutionary MCTS, as explained in Section 9.4, using both knowledge base and evolution to bias the roll-outs.

The experiments can be analyzed by considering two measures: the percentage of victories achieved and the score earned. As in the competition (see Appendix A), it is considered that the former value takes precedence over the rankings: it is more relevant to win the game than to lose it with a higher score. Also, comparing the percentage of victories across all games is more representative than comparing scores, as each game has a completely different score system. However, it is interesting to compare scores on a game by game basis.

The average victories and scores obtained in every game are shown in Tables 9.1 and 9.2, respectively. In most games, *KB Fast-Evo MCTS* shows a better performance than *Vanilla MCTS* in both percentage of victories and scores achieved.

	Percentage Victories			
Game	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**
Aliens	8.0 (5.4)	4.0 (3.9)	4.0 (3.9)	**100.0 (0.0)**
Boulderdash	0.0 (0.0)	4.0 (3.9)	**28.0 (9.0)**	**16.0 (7.3)**
Butterflies	88.0 (6.5)	**96.0 (3.9)**	80.0 (8.0)	**100.0 (0.0)**
Chase	12.0 (6.5)	12.0 (6.5)	0.0 (0.0)	**92.0 (5.4)**
Frogs	24.0 (8.5)	16.0 (7.3)	8.0 (5.4)	20.0 (8.0)
Missile Command	20.0 (8.0)	20.0 (8.0)	20.0 (8.0)	**56.0 (9.9)**
Portals	12.0 (6.5)	**28.0 (9.0)**	16.0 (7.3)	**28.0 (9.0)**
Sokoban	0.0 (0.0)	0.0 (0.0)	**4.0 (3.9)**	**8.0 (5.4)**
Survive Zombies	44.0 (9.9)	36.0 (9.6)	52.0 (10.0)	44.0 (9.9)
Zelda	8.0 (5.4)	**20.0 (8.0)**	8.0 (5.4)	**28.0 (9.0)**
Overall	21.6 (2.6)	23.6 (2.7)	22.0 (2.6)	**49.2 (3.2)**

Table 9.1: Percentage of victories from each game. In bold, those results that are the best ones from each game. Each value corresponds to the average obtained by playing that particular game 25 times.

According to the total average of victories, *KB Fast-Evo MCTS* leads the comparison with 49.2% (3.2) of games won. The other MCTS versions all obtained similar victory rates in the range 20% to 25%. This difference shows that adding both the knowledge base and evolution to bias the roll-outs provides a strong advantage to MCTS, but adding each one of these features separately does not impact the vanilla MCTS algorithm significantly.

Regarding the scores, *KB Fast-Evo MCTS* also leads on average points achieved, with 13.5 (1.2) points, against the other algorithms (with results all ranging between 9 and 11 points). Nevertheless, as mentioned before, this particular result must be treated with care as different games vary in their score system. It is therefore more relevant to compare scores on a game by game basis.

In some cases, like in *Boulderdash*, the increase in victory percentage is obtained when adding the knowledge base system, while in others, as in *Zelda*, it is the evolution feature that gives this boost. On average in most cases (and exceptionally in *Missile Commmand* and *Chase*), it is both evolution and the knowledge base that cause the improvement. Special mention must be made of *Aliens*, *Butterflies* and *Chase*, in which the *KB Fast-Evo MCTS* algorithm achieved a very high rate of victories. Similarly, *Aliens*, *Chase* and *Missile Command* show a relevant improvement in average score.

KB Fast-Evo MCTS fails, however, to provide good results in certain games, where little (as in *Sokoban* and *Boulderdash*) or no improvement at all (like in *Survive Zombies* and *Frogs*) is observed compared with *Vanilla MCTS*. The reasons are varied as to why this algorithm does not achieve the results it did in other games. Clearly, one is the fact that the distances between sprites are Euclidean,

	Average Score			
Game	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**
Aliens	36.72 (0.9)	38.4 (0.8)	37.56 (1.0)	**54.92 (1.6)**
Boulderdash	9.96 (1.0)	12.16 (1.2)	**17.28 (1.7)**	**16.44 (1.8)**
Butterflies	27.84 (2.8)	31.36 (3.4)	31.04 (3.4)	28.96 (2.8)
Chase	4.04 (0.6)	4.8 (0.6)	3.56 (0.7)	**9.28 (0.5)**
Frogs	-0.88 (0.3)	-1.04 (0.2)	-1.2 (0.2)	-0.68 (0.2)
Missile Command	-1.44 (0.3)	-1.44 (0.3)	-1.28 (0.3)	**3.24 (1.3)**
Portals	0.12 (0.06)	**0.28 (0.09)**	0.16 (0.07)	**0.28 (0.09)**
Sokoban	0.16 (0.1)	0.32 (0.1)	**0.7 (0.2)**	**0.6 (0.1)**
Survive Zombies	13.28 (2.3)	14.32 (2.4)	18.56 (3.1)	21.36 (3.3)
Zelda	0.08 (0.3)	0.6 (0.3)	0.8 (0.3)	0.6 (0.3)
Overall	9.0 (0.9)	10.0 (1.0)	10.7 (1.0)	**13.5 (1.2)**

Table 9.2: Percentage of scores from each game. The results in bold are the best from each game. Each value corresponds to the average result obtained by playing that particular game 25 times.

not considering obstacles. Shortest distances (i.e. using A*) would positively affect the performance of the algorithm. This is, however, not trivial: path-finding requires the definition of a navigable space. In the GVGAI framework this can be inferred as empty spaces in most games, but in others (particularly in *Boulderdash*), the avatar moves by digging through *dirt*, creating new paths with each movement (but *dirt* itself is not an obstacle).

Nevertheless, not all problems can be attributed to how the distances are calculated. For instance, in *Sokoban*, where boxes are to be pushed, it is extremely important to consider where the box is pushed from. However, the algorithm considers collisions as non-directional events. It could be possible to include this information (which collisions happened from what direction) in the model, but this would be an unnecessary division for other games, where it is not relevant. Actually, it would increase the number of features considered, causing a higher computational cost for their calculation and a larger search space for the evolutionary algorithm.

Another interesting case to analyze is *Frogs*. In this game, the avatar usually struggles with crossing the road. This road is typically composed of three lanes with many trucks that kill the agent when contacting with it. Therefore, the road needs to be crossed quickly, and most of the roll-outs are unable to achieve this without colliding with a truck. The consequence of this is that most of the feedback retrieved suggests that the action that moves the avatar into the first lane will most likely cause the player to lose the game. A typical behaviour observed in this game is the agent moving parallel to the road without ever crossing it, trying to find gaps to cross, but too "scared" to actually try it.

9.6 Conclusions

This chapter explored the performance and problems of a vanilla Monte Carlo Tree Search (MCTS) algorithm in the field of General Video Game Playing (GVGP). Several modifications to the algorithm have been implemented in order to overcome the problems, such as rewarding the discovery of new sprites, augmenting the knowledge of other elements in the game, and using past experience to ultimately guide the MCTS roll-outs. Results show a significant improvement in performance, both in percentage of victories and scores achieved. These improvements work better in some games than in others, and reasons for this have also been suggested.

This work presages multiple future extensions, such as the implementation of a general path-finding algorithm for better distance measurements, or experimenting with different algorithms to bias roll-outs. This study features a $(1+1)$ Evolution Strategy to guide the Monte Carlo simulations, but more involved evolutionary techniques will be explored, as well as other approaches like gradient descent methods.

Another possibility is to employ a larger set of features, which could help to overcome some of the problems found in this research. This option would probably increase the computational cost of each iteration, but some works in the literature can help to deal with this issue. For instance, Gaudel and Sebag (Gaudel and Sebag, 2010) introduced Feature UCT Selection (FUSE) for single player games where the combinatorial problem of choosing among several features was tackled. Alternatively, there exists the possibility of learning the relative importance of these features, as described by (Finnsson and Björnsson, 2010), using TD(λ) or other learning algorithms.

Finally, another line of future work is to apply Mutli-Objective MCTS (as discussed in Chapter 8) to tackle this problem, considering *score*, *curiosity* and *experience* as three different objectives.

GVGP, with the absence of game-dependent heuristics, has proven to be a challenging and fascinating problem. It reproduces current open challenges in Reinforcement Learning, such as the absence of meaningful rewards (as explained for *Frogs*, in Section 9.5).

ƹ

"And why? Because my creators thought

that God wanted it that way!"
- Brother Cavil, Battlestar Galactica

Chapter 10

Postmortem

This chapter draws the final conclusions of the thesis and outlines directions for future work.

10.1 Conclusions

This thesis has explored the main problems and hazards that real-time games pose to adaptive controllers, owing to the limited time budget available for deciding the next move. This affects effective planning, preventing the algorithms from reaching end-game states during simulations, and creates the need for long-term plans to foresee the consequences of the actions taken to reach the game's end. The conducted research focuses on providing solutions to these problems, proposing new algorithms or modifying existing ones.

Not being able to reach end-game states forces the algorithm to evaluate non-terminal states, giving higher rewards to those states that seem to be closer to a victory in the game. Non-terminal state evaluation has always been a part of the different algorithms explored in this thesis. Usually, score functions are programmed by applying game-specific knowledge: heuristics that can also be used to bias the simulations performed in the model of the game. These heuristics help the algorithms to focus on those regions of the search space that are more promising, discarding or penalizing those actions that clearly do not lead to a victory. Even when the time budget is very limited (as in Chapter 4 with only 10 ms of thinking time), experiments show that the use of heuristics clearly improves the performance of the algorithms tested. Domain knowledge can be successfully applied to guide the roll-outs of Monte Carlo Tree Search, determine the reward of non-terminal states, or even to prune branches of the tree that seem to lead to poor states in the search space (as shown in Chapter 4).

Interestingly, game-specific knowledge can be deceptive when analyzing the performance of algorithms. If the heuristics designed to help a given search technique are so good that they are able to simplify the reward landscape of the problem to the extreme, it is sensible to think that many algorithms embedded with these heuristics will be successful in solving the problem. Obviously, this is not technically wrong (after all, the problem is being solved), but this situation may make the choice of algorithm almost irrelevant. In other words, what would be the benefit to improving any algorithm if we could design the perfect heuristic for any given game? Of course, this heuristic may not exist, or may have not been discovered yet, so investing in improving algorithms is not worthless.

General Video-Game Playing tries to tackle this issue by seeking controllers capable of playing in completely different games, even in hitherto unseen games. Game-specific heuristics are no longer an option in this scenario and the search, discovery and memory capabilities of the algorithms compared become more relevant. Vanilla algorithms that are tested in this domain struggle to produce good results when deprived of domain-dependant heuristics, reproducing most of the more relevant problems in reinforcement learning (as seen in Chapter 9). The difficulties faced become more complex as this research gets closer to the open problem of General Artificial Intelligence, and this provides an interesting and prolific base for future work.

The research described in this thesis also tackles the problem of devising plans that affect the entire line of play of an agent, that can be integrated with a short-term action decision mechanism. It is possible to significantly improve the performance of the algorithms by solving the long-term and short-term aspects of the game separately. Chapter 5 shows that these two different ways of planning should not be completely independent, as one affects the other and vice versa. Exemplified in the Physical Travelling Salesman Problem (PTSP), it is not possible to obtain very good results if the order of waypoints (long-term plan) does not consider the way the agent drives through the maze avoiding obstacles (short-term plan). Similarly, the way the controller drives to approach a waypoint is influenced by the overall route the plan has devised.

Long-term planning is, in fact, an important part of how an agent plays any (moderately long) game. It is clearly influenced by the design of the game, and creating these plans effectively should reward the player with higher chances of victory (or better results). It is this concept that motivated the exploration in this thesis of another use of adaptive controllers, over and above simply behaving

intelligently in the game. It is possible to generate maps or levels for a game in which a good long-term plan achieves generally better results than a poor one, by measuring the quality of a map as the difference between the performance of these different approaches. Chapter 6 shows that it is possible to automatically generate maps for the PTSP that fulfil this requirement. Furthermore, by employing agents able to devise different, good, long-term plans, experimental work shows that there exist several different high quality strategies to solve any problem. In other words, there is not only one way of winning the game. Clearly, as levels of any kind can be automatically created, the importance of a reliable adaptive controller that can deal with a wide range of situations is essential. Procedural content generation for games is therefore an interesting application for adaptive controllers.

Another problem, a consequence of the very few milliseconds controllers have to decide the next move, is that simulations executed by the algorithm are not able to reach states far in the future. Chapter 7 of this thesis shows that, in those cases where the effect of a single action does not heavily change the state of the game, it is possible to treat sequences of actions (macro-actions) as a single move. Essentially, this benefits the algorithm in two ways. First, it allows the algorithm to reach states farther in the future, permitting it to analyze the consequences of the actions chosen at a later state. Secondly, it indirectly allows for longer time budgets in subsequent game cycles, as the algorithm has as many game steps as the number of moves in the macro-action to decide the next move. Experiments showed that this coarsening of the action space clearly improves the performance of the algorithms.

This thesis also proposes a new use of an evolutionary algorithm for real-time games, following a rolling (or receding) horizon procedure. Traditionally, evolutionary algorithms are employed offline: a solution is evolved in a first stage, and it is used afterwards to solve a given problem. Chapter 7 describes a rolling horizon version of an evolutionary algorithm, and compares it with MCTS, obtaining very promising results. The concept is simple: instead of evolving a solution offline, small populations are evolved on each game cycle, where each individual is a sequence of actions to execute in the game. Each individual is evaluated according to the score function of the state reached at the end of the executed sequence of actions. When the time budget runs out, the first action of the best individual is returned to be executed. This algorithm, in conjunction with the macro-actions described above, produces results comparable to MCTS, even better in some scenarios.

Finally, this thesis suggests looking at games from a different perspective. It may be possible to

decompose the objective of the game (in the broader sense, to win) into multiple tasks that must be tackled simultaneously, and to apply multi-objective principles to solve them. Chapter 8 proposes a novel Multi-Objective MCTS (MO-MCTS) algorithm for real-time games, testing it in different multi-objective games. The main idea is to build Pareto front approximations on each node of the tree during the back-propagation phase. The algorithm proposed is computationally inexpensive, approximates the optimal Pareto front of the problem at the root of the tree, and maintains a record of which action at the root led to what point in the Pareto front. As action selection is made considering the statistics from the root node, the behaviour of the agent can be focused to any solution of the approximated Pareto front. Experiments show that MO-MCTS obtains better results than an MCTS algorithm that uses a weighted-sum on the different rewards in the most complicated problems tested.

10.2 Future Work

The research conducted for this thesis leads to many possible lines for future work. Essentially, the improvements and algorithms proposed are focused on real-time single player games. In these scenarios, the player is the only agent that actively pursues victory in the game, without any other opposition than gameplay features or random events. Therefore, a possible continuation of this work could be to analyze the performance of adaptive controllers in two-player (or more) adversarial games.

Additionally, this research is centred on games where the whole state is observable to the agent and the model of the game is deterministic (with the exception of some games from the General Video Game Playing framework). Multiple games intentionally hide information from the player, such as Fog Of War in real-time strategy games, or are driven by stochastic events, such as incoming pieces in the game of Tetris or letters in Scrabble. These types of games (partially observable and non-deterministic, respectively) are an interesting field of study, and the addition of real-time constraints makes them more challenging.

However, from the point of view of the author of this thesis, General Video-Game Playing (GVGP) offers probably the most interesting line of future work suggested in this research. The real-time aspect of video-games makes behaving appropriately a difficult task for the agent, in the absence of game specific information. The experiments described in Chapter 9 of this thesis show that the variety of situations faced in this context are a significant hazard, leaving some problems unresolved. Knowledge

discovery and re-use of past experiences have proven to be important concepts on GVGP, but there is still room for other improvements in this context. One example is the concept of contingency awareness introduced by Bellemare et al. (Gendron-Bellemare et al., 2012) for the Atari 2600 games, a notion that tries to identify which events in the future are a consequence of the agent's actions, and not part of the environment. Another possibility is the usage of multi-objective approaches, considering *score*, *curiosity* and *experience* as three objectives that must be optimized simultaneously.

Furthermore, adaptive controllers can also be put to the test with partially observable and non-deterministic games within the GVGAI framework. In the long term, research in this field leads to General Artificial Intelligence, where an agent would be able to solve any task without domain specific information.

Finally, the use of adaptive controllers to automatically generate content for games (as seen in Chapter 6 of this thesis) suggests another line of future work for GVGP. In principle, it would be possible to employ real-time agents to create complete games. Analogously to the case explored in this research, adaptive controllers that are able to play any real-time (automatically created) game could be used to evolve new game rules or descriptions. In principle, it should be possible to use different controllers, that employ distinct strategies and are of a different strength, to generate games with some desired specific features.

ƹ

Appendix A

Video-Game Competitions

This appendix provides an overview on the competitions organized by the author of this thesis. Most of the contents of this chapter has been published in (Perez et al., 2012c) and (Perez, Powley, Whitehouse, Samothrakis, Lucas, and Cowling, 2014b).

A.1 Organized Competitions

The research conducted and described in this thesis is complemented with the organization of real-time game competitions between 2012 and 2014, using most of the games described in Chapter 3. All competitions organized share a similar infrastructure, following what intuitively seems to be a set of good practices:

- **Information:** By using a website, instructions of the competition, documentation, getting started guides, software and competition rules are detailed for the participants of the competition.
- **Testing sets:** Each competition uses three sets of maps (for the PTSP, games for GVGAI): *training*, *validation* and *test*. The first set is provided with the framework, and it is the only set visible to the participants. The second one is sited on the web so participants can submit their controllers to be tested there at any time, while the third one is only used after the competition deadline is over to obtain the final rankings of the contest.
- **Immediate feedback:** Participants can submit their controllers at any time to be evaluated in the *training* and *validation* sets. The server compiles and executes the controller automatically, reporting compilation and runtime errors in case these occur.

- **Online rankings:** Results from the participant submissions form preliminary rankings that are shown in the website. This way, the participants can foresee the quality of their controllers in relation to other contestants.

- **Game replays:** Games executed in the *training* set can be replayed in an applet in the website. This allows the participants to visualize the execution of their (and others') controllers, in order to pinpoint potential mistakes and aspects to improve.

- **Discussion group:** A discussion group (via an email list) is provided so participants and organizers can discuss aspects of the competition, bugs, rules and suggestions.

- **Human play:** An applet on the website allows participants to play the game as humans. This not only engages the participants in the contest, but also permits the gathering of human results and comparison of these with the final results of the competition submissions.

The rest of this section details the different competitions organized and provides a brief discussion on the results.

A.1.1 WCCI 2012 PTSP Competition

Organizers: Diego Perez, Philipp Rohlfshagen, David Robles and Simon Lucas.

Website: www.ptsp-game.net

PTSP was employed for the first time in a contest in the World Congress on Computational Intelligence (WCCI) 2012 PTSP Competition, Brisbane, Australia. The game is described in Section 3.2 of this thesis, details about the competition infrastructure can be found at (Perez et al., 2012c) and the complete results can be found at the competition website[1].

This competition ran three different tracks in parallel: *bots*, *human* and *human vs. bots*. The competition received 28 submissions and 306 humans played the maps in the *training* set. In the comparison between human and bots, it was the user *slash* who, as a human, was declared winner. All maps from the competition contained 10 waypoints, suggesting that a reasonable order of waypoints can be devised by a human that plays the game.

[1] www.ptsp-game.net/final_rankings.php

Rank	Bot	Controller	Total
1	Purofvio	TSP solver for order of waypoints, MCTS for steering using macro-actions, (Powley et al., 2012).	**196**
2	st3f1	Heuristic based controller.	**124**
3	shavlir	Depth First Search (DFS).	**121**
4	ICE_DE	Differential Evolution algorithm to find the order of waypoints and Fuzzy Control to drive the ship.	**88**
5	hiten_sonpal	A* for waypoint order and DFS	**84**

Table A.1: Bot Track Competition Results, PTSP, WCCI 2012.

The five top controllers from the *bots* track are shown in Table A.2. It is interesting to note that the winner, *Purofvio* (by Edward Powley and Daniel Whitehouse), is an MCTS implementation that uses macro-actions, a very similar approach to the one described in this thesis, in Chapter 7. This controller also ranked second in the *human vs. bots* track of the competition.

The second best submitted bot was created using no computational intelligence technique, but based only on rules and heuristics determined by hand. Special mention must be made of the use of Depth First Search to decide the next move to make, and the use of differential evolution and fuzzy logic by the third best bot of the competition.

The rankings of the competition were computed as follows. Following a Formula 1 (F1) scoring scheme, each entry on each map received points according to its rank ($10, 8, 6, 5, 4, 3, 2$ and 1 points from the first to the eighth entry), determined by the number of waypoints visited and the time spent, in this order. The final score was computed as the sum of all points across the maps in the *test* set.

A.1.2 CIG 2012 PTSP Competition

Organizers: Diego Perez, Philipp Rohlfshagen, David Robles and Simon Lucas.

Website: www.ptsp-game.net

A second edition of the competition was held, a few months later, at the Conference on Computational Intelligence and Games (CIG) 2012, Granada, Spain. The main difference between this and the previous contest was the number of waypoints per map: 30, 40 and 50. This competition received fewer bot submissions (only 4) and human players (12). The winner of the last *human* and *human vs. bots* track participated in this contest as well, ranking first in the *human* track, and third in the *human vs. bot* track, behind two bots. This result suggests that the difficulty of the game was increased when adding more waypoints, and therefore calculating the optimal route became a task too complex for a

Rank	Bot	Controller	Total
1	Purofvio	TSP solver for order of waypoints, MCTS for steering using macro-actions, (Powley et al., 2012).	**500**
2	shavlir	Depth First Search (DFS).	**309**
3	hiten_sonpal	A* and DFS.	**290**
4	ICE_SOM_FZ	Self Organizing Map (SOM) algorithm to find an order of cities to visit and fuzzy logic for driving the ship.	**285**
5	greedy	Greedy Sample Controller	**202**

Table A.2: Bot Track Competition Results, PTSP, CIG 2012.

human.

Table A.2 shows the five top controllers from the *bots* track. Again, the controller submitted by *Purofvio* is at the top of the rankings, with a very similar controller to the one submitted to the WCCI competition (with some differences to adjust to the higher number of waypoints). The other participants at the top of the rankings repeated the same approach (or modified it slightly) attempted in the previous competition. The complete rankings can be found on the competition website[2].

Rankings were computed similarly to the ones in the previous contest, but using the newest F1 score system: $25, 18, 15, 12, 10, 8, 6, 4, 2$ and 1 points from the first to the tenth entry, which awards more value to higher positions in the ranking.

A.1.3 CIG 2013 MO-PTSP Competition

Organizers: Diego Perez and Simon Lucas.

Website: www.ptsp-game.net

MO-PTSP, as described in section 3.5, featured in the MO-PTSP CIG 2012 Competition, Niagara Falls, Canada. The competition received 6 bot submissions and 13 humans played the game for the *human* track. Table A.1 shows the top five submitted controllers. The winner of the *bot* track, *Purofmovio*, was submitted by the same group that won the two editions of the PTSP competition. The winning approach was again an MCTS controller, providing a weighted sum on the different objectives of the game. This controller is significantly more complex than its predecessors, with a larger set of parameters to tune. Although the parameters where set by hand in the PTSP competitions, CMA-ES was used to tune them in this contest. The controller is fully described in (Powley et al., 2013).

The second controller, submitted by Weijia, is a modification of the algorithm described in their pa-

[2]http://www.ptsp-game.net/final_rankings.php?c=cig12

Rank	Bot	Controller	Total
1	PurofMovio	Weighed-sum MCTS, parameteres evolved with CMA-ES (Powley et al., 2013)	**305**
2	Weijia	Multi-Objective MCTS	**31**
3	MacroRSController	Macro-Action Sample Controller	**8**
4	GreedyController	Greedy Sample Controller	**3**
5	LordOkami	Rule-based system	**0**

Table A.3: Bot Track Competition Results, MO-PTSP, CIG 2013.

per (Weijia and Sebag, 2013), and also introduced in this thesis in Section 8.1. The other 4 submissions ranked behind two sample controllers distributed with the framework, that used a rule-base system, UCT, MCTS and a simple genetic algorithm, respectively. In the *human vs. bots* track, *Purofmovio* was also declared winner and the first human player, *mmorenosi*, ranked second in this track. Full rankings can be found at the competition website[3].

One of the main contributions of this competition was the mechanism used to rank the entries. By using the concept of dominance (see Section 2.4), the controllers were ranked in Pareto fronts per map, and only those entries in the optimal front were awarded with points. Additionally, the best results in each particular objective (within the optimal front), received extra points. A full description of this procedure, rules of the competition and final rankings can be found at (Perez et al., 2014b).

A.1.4 CIG 2014 GVGAI Competition

Organizers: Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul and Simon Lucas.

Website: www.gvgai.net

General Video Game Playing is the topic of the GVGAI Competition, held at CIG in August 2014, Dortmund, Germany. Participants were able to train their approaches in ten games distributed with the framework (the ones described in Section 3.6) and submit their controllers to the competition server to be executed in a validation game set, which games were unknown to the contestants. The final rankings were computed after the deadline of the competition, in a final (and unseen) set of ten new games, also unknown to the participants.

Rankings use the same scheme as in the CIG 2012 PTSP Competition, and they are determined as follows for each game. There are five different levels for each one of the 10 games from the *test* set. All levels were played ten times, recording the number of victories, total scores achieved and time steps

[3]http://www.ptsp-game.net/final_mo_bot_rankings.php

Rank	Bot	Controller	Total
1	adrienctx	Open-loop tree search. States are not saved in the nodes of the tree and UCB is used to balance exploration and exploitation.	**158**
2	JinJerry	Multi-step look forward with heuristic for game state.	**148**
3	SampleMCTS	Sample MCTS Controller provided with the framework	**99**
4	Shmokin	MCTS with hill climbing heuristic	**77**
5	Normal_MCTS	MCTS and pathfinding	**68**

Table A.4: GVGAI Competition Results, CIG 2014.

employed. Rankings in a particular game were determined by the controller that obtained the highest number of victories, using highest score and lowest time spent as the first and second tie breakers, respectively. Table A.4 shows the top five controllers in the final rankings of the competition. Detailed information about the games played and full rankings can be found on the website[4].

This competition received 14 working entries. An important percentage of these were MCTS variants, although other approaches were also attempted by the participants, such as genetic algorithms, rule based controllers, Q-Learning and Pathfinding techniques. The winner of the competition implemented an open-loop tree search algorithm: a tree is built that represents a sequence of actions, without storing the states in the nodes of the tree. The UCB1 formula (see Equation 2.1) is used to select actions during the tree policy, modified with a *taboo bias* that prevents the controller from selecting actions that would lead the avatar to previously visited positions.

It is interesting to mention that there were no controllers able to obtain more than 50% of victories in both validation and test game sets (this is, in the games that were unknown to the participants). This shows the complexity of this domain, and reveals that further research is needed to tackle this problem. Similarly, it is interesting to highlight that most variations of Sample MCTS finished in a position in the rankings below the original algorithm, suggesting the difficulty of adapting a simple approach to account for unknown games.

[4]http://www.gvgai.net/gvg_rankings_test.php

A.2 Related Competitions

Competitions in the field of game artificial intelligence have been organized regularly during the last decade, where developers and researchers can test and compare their algorithms in an established benchmark. A relevant subset of these competitions involves real-time games. In these competitions the controller must respond with an action within a time limit and must balance the desire to make an optimal move with the necessity to respond quickly.

One of the longest-running real-time competitions is the *Ms Pac Man* screen-capture competition, based on the famous Ms Pac Man arcade game, organised by Simon M. Lucas (Lucas, 2007) since 2007. The objective is to control Ms Pac Man and obtain the highest possible score receiving input from a screen capture of the original game. A recent modification of this competition, the *Ms Pac-Man vs Ghosts* competition (by P. Rohlfshagen, D. Robles and S. Lucas (Rohlfshagen and Lucas, 2011)), allows the participants to control both Ms Pac-Man and the ghosts.

Another well known competition is the *Simulated Car Racing* competition, by D. Loiacono (Loiacono, Lanzi, Togelius, Onieva, Pelta, Butz, Lönneker, Cardamone, Perez, Sáez, Preuss, and Quadflieg, 2010). It has been running since 2007 and is based on the TORCS racing simulator. With different formats, the goal of this game is to find the fastest driver, in either single- or multi-player modes. Again, the controller has to respond within a given amount of time. A more recent competition is the *Mario AI* Competition, by S. Karakovskiy and J. Togelius (Togelius, Karakovskiy, and Baumgarten, 2010). It is based on the famous video game Super Mario Bros., using a modified version of it called *Infinite Mario Bros.*. Its gameplay track combines path planning and reactiveness to enemy movements.

Two more recent real-time game competitions are the *2K BotPrize* (by P. Hingston (Hingston, 2010)) and the *Starcraft* Competition (by B. Weber (Weber et al., 2011)). These games cover two genres of games that have not been covered before: First Person Shooter and Real-Time Strategy games, respectively. In both scenarios, navigation through the world plays an important part in the final success of the controllers. In fact, in the case of the 2K BotPrize, the objective is to make the participant's controller behave like a real human (as in the Turing Test), so navigation needs to be not only effective, but also performed in a human-like way.

ﻩ

Bibliography

A. Alhejali and S. M. Lucas. Using Genetic Programming to Evolve Heuristics for a Monte Carlo Tree Search Ms Pac-Man Agent. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 65–72, 2013.

B. Arneson, R. B. Hayward, and P. Henderson. Monte Carlo Tree Search in Hex. *IEEE Trans. Comp. Intell. AI Games*, 2(4):251–258, 2010.

D. Ashlock, C. Lee, and C. McGuinness. Search-Based Procedural Generation of Maze-Like Levels. *IEEE Transactions in Computational Intelligence and AI in Games*, 3:260–273, 2011.

D. Ashlock, C. McGuinness, and W. Ashlock. Representation in Evolutionary Computation. In *IEEE Congress on Evolutionary Computation*, pages 77–97, 2012.

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2):235–256, 2002.

A. Auger and N. Hansen. A Restart CMA Evolution Strategy With Increasing Population Size. In *Proceedings of Conference on Evolutionary Computation*, pages 1769–1776, 2005.

R.-K. Balla and A. Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence.*, pages 40–45, 2009.

L. Barrett and S. Narayanan. Learning All Optimal Policies with Multiple Criteria. In *Proceedings of the International Conference on Machine Learning*, pages 41–47, 2008.

R. Beer and J. Gallagher. Evolving Dynamical Neural Networks for Adaptive Behavior. *Adaptive Behavior*, 1(1):91–122, 1992.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

A. Benbassat and M. Sipper. EvoMCTS: Enhancing MCTS-based Players Through Genetic Programming. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 57–64, 2013.

N. Beume, B. Naujoks, and M. Emmerich. SMS-EMOA: Multiobjective Selection Based on Dominated Hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

Z. Bouzarkouna, D. Y. Ding, and A. Auger. Using Evolution Strategy with Meta-models for Well Placement Optimization. In *Proceedings of the 12th European Conference on the Mathematics of Oil Recovery ECMOR 2010*, pages 1–15, 2010.

D. Brockhoff. Tutorial on evolutionary multiobjective optimization. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 307–334, 2013.

C. Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, School of Software Engineering and Data Communications, Queensland University of Technology, 2008.

C. Browne, E. J. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 2012.

G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proceedings of the Artificial Intelligence for Interactive Digital Entertainment Conference*, pages 216–217, 2006.

G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Math. Nat. Comput.*, 4(3):343–357, 2007.

B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 389–395, 2008.

M. Chung, M. Buro, and J. Schaeffer. Monte Carlo Planning in RTS Games. In *the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 117–124, 2005.

C. Coello. An Updated Survey of Evolutionary Multiobjective Optimization Techniques: State of the Art and Future Trends. In *Proc. of the Congress on Evolutionary Computation*, pages 3–13, 1999.

C. Coello. *Handbook of Research on Nature Inspired Computing for Economy and Management*, chapter Evolutionary Multi-Objective Optimization and its Use in Finance, pages 74–88. Idea Group Publishing, 2006.

P.-A. Coquelin and R. Munos. Bandit Algorithms for Tree Search. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 67–74, 2007.

R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computer Games*, pages 72–83. Springer-Verlag, 2006.

K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2002.

H. Finnsson and Y. Björnsson. Simulation-based Approach to General Game Playing. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 259–264, 2008.

H. Finnsson and Y. Björnsson. CADIA-Player: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1:1–12, 2009.

H. Finnsson and Y. Björnsson. Learning Simulation Control in General Game-Playing Agents. In *Proc. 24th AAAI Conf. Artif. Intell.*, pages 954–959, Atlanta, Georgia, 2010.

Z. Gabor, Z. Kalmar, and C. Szepesvari. Multi-criteria Reinforcement Learning. In *The Fifteenth International Conference on Machine Learning*, pages 197–205, 1998.

R. Gaudel and M. Sebag. Feature Selection as a One-Player Game. In *Proc. 27th Int. Conf. Mach. Learn.*, pages 359–366, Haifa, Israel, 2010.

S. Gelly and D. Silver. Combining Online and Offline Knowledge in UCT. In *Proc. 24th Annu. Int. Conf. Mach. Learn.*, pages 273–280, Corvalis, Oregon, 2007. ACM.

S. Gelly and D. Silver. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, 2006.

M. Gendron-Bellemare, J. Veness, and M. Bowling. Investigating Contingency Awareness using Atari 2600 Games. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI)*, pages 864–871, 2012.

M. Genesereth, N. Love, and B. Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26:62–72, 2005.

F. Gomez and R. Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1356–1361. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

A. Gut. *An Intermediate Course in Probability*. Springer-Verlag, 1995.

N. Hansen. The CMA Evolution Strategy: A Comparing Review. In J. Lozano, P. Larranaga, I. Inza, and Bengoetxea, editors, *Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms*, pages 75–102. Springer, 2006.

N. Hansen. Benchmarking a BI-Population CMA-ES on the BBOB-2009 Function Testbed. In *Workshop Proceedings of the GECCO Genetic and Evolutionary Computation Conference*, pages 2389–2395, 2009.

E. Hastings, R. Guha, and K. O. Stanley. Evolving Content in the Galactic Arms Race Video Game. In *IEEE Symposium on Computational Intelligence and Games*, pages 241–248, 2009.

M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, DOI:10.1109/TCIAIG.2013.2294713:1–18, 2013.

P. Hingston. A new design for a turing test for bots. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 345–350, 2010.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992.

A. Homaifar, C. X. Qi, and S. H. Lai. Constrained Optimization Via Genetic Algorithms. *SIMULATION*, 62(4):242–253, 1994.

J. Huang, Z. Liu, B. Lu, and F. Xiao. Pruning in UCT Algorithm. In *Proc. Int. Conf. Tech. Applicat. Artif. Intell.*, pages 177–181, Hsinchu, Taiwan, 2010.

I. D. V. I. (IDV). Speedtree. http://www.speedtree.com/, 2012.

N. Ikehata and T. Ito. Monte-Carlo Tree Search in Ms. Pac-Man. In *Proceedings of IEEE Conference on Computational Intelligence and Games*, pages 39–46, 2011.

H. Jain and K. Deb. An Improved Adaptive Approach for Elitist Nondominated Sorting Genetic Algorithm for Many-Objective Optimization. In R. Purshouse, P. Fleming, C. Fonseca, S. Greco, and J. Shaw, editors, *Evolutionary Multi-Criterion Optimization*, volume 7811 of *Lecture Notes in Computer Science*, pages 307–321. Springer Berlin Heidelberg, 2013.

D. S. Johnson and L. A. McGeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*. John Wiley and Sons, Ltd., 1997.

J. Joines and C. Houck. On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems with GA's. In *Proceedings of the Evolutionary Computation Conference, Poster Sessions*, pages 579–584, 1994.

S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a Feasible-Infeasible Two-Population FI-2Pop genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 192(2):310–327, 2008.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. *Machine Learning: ECML 2006*, 4212:282–293, 2006.

T. Kozelek. *Methods of MCTS and the game Arimaa*. Master's thesis, Charles Univ., Prague, 2009.

A. Land and A. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960.

C.-S. Lee, M.-H. Wang, G. M. J.-B. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73–89, 2009.

J. Levine, C. B. Congdon, M. Bída, M. Ebner, G. Kendall, S. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson. General Video Game Playing. *Dagstuhl Follow-up*, 6:1–7, 2013.

A. Liapis, G. N. Yannakakis, and J. Togelius. Neuroevolutionary Constrained Optimization for Content Creation. In *Proc. of the IEEE Conference on Computational Intelligence and Games (CIG)*, pages 71–78, 2011.

A. Liapis, G. N. Yannakakis, and J. Togelius. Adapting Models of Visual Aesthetics for Personalized Content Creation. *IEEE Transactions on Computational Intelligence and AI in Games*, Special Issue on Computational Aesthetics in Games 2012:213–228, 2012.

D. Loiacono, P. Lanzi, J. Togelius, E. Onieva, D. Pelta, M. Butz, T. Lönneker, L. Cardamone, D. Perez, Y. Sáez, M. Preuss, and J. Quadflieg. The 2009 Simulated Car Racing Championship. *IEEE Transactions on Computational Intelligence and Games*, 3(2):131–147, 2010.

S. M. Lucas. Ms Pac-Man competition. *SIGEVOlution*, 2:37–38, December 2007.

S. Lucas, S. Samothrakis, and D. Perez. Fast Evolutionary Adaptation for Monte Carlo Tree Search. In *Proceedings of EvoGames*, page to appear, 2014.

R. T. Marler and J. S. Arora. Survey of Multi-objective Optimization Methods for Engineering. *Structural and Multidisciplinary Optimization*, 26:369–395, 2004.

S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo, and H. Futahashi. Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem. In *Proceedings of the International Multi Conference of Engineers and Computer Scientists*, volume 3, pages 2086–2091, 2010.

J. Méhat and T. Cazenave. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions in Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

J. Méhat and T. Cazenave. A Parallel General Game Player. *KI - Künstliche Intelligenz*, 25:43–47, 2011.

J. Mercieca and S. Fabri. Particle Swarm Optimization for Nonlinear Model Predictive Control. In *ADVCOMP 2011, The Fifth International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 88–93, 2011.

Z. Michalewicz. A Survey of Constraint Handling Techniques in Evolutionary Computation Methods. In *Evolutionary Programming*, pages 135–155, 1995.

M. Möller, M. T. Schneider, M. M. Wegner, and T. Schaub. Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI - Künstliche Intelligenz*, 25:17–24, 2011.

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

S. Natarajan and P. Tadepalli. Dynamic Preferences in Multi-Criteria Reinforcement Learning. In *In Proceedings of the International Conference of Machine Learning*, pages 601–608, 2005.

M. Naveed, D. Kitchin, and A. Crampton. Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games. In *Proceedings of PlanSIG 2010. 28th Workshop of the UK Special Interest Group on Planning and Scheduling*, 2010.

S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5:4:293–311, 2013.

D. Perez and S. Lucas. The Multi-Objective PTSP Competition, August 2013. www.ptsp-game.net.

D. Perez, S. Mostaghim, S. Samothrakis, and S. Lucas. Multi-Objective Monte Carlo Tree Search for Real-Time Games. *IEEE Transactions on Computational Intelligence and AI in Games (submitted)*, pp:1–13, 2014a.

D. Perez, E. Powley, D. Whitehouse, S. Samothrakis, S. Lucas, and P. I. Cowling. The 2013 Multi-Objective Physical Travelling Salesman Problem Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2014b.

D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. Lucas. Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions. *IEEE Transactions on Computational Intelligence and AI in Games*, 6:1:31–45, 2013a.

D. Perez, P. Rohlfshagen, and S. Lucas. Monte-Carlo Tree Search for the Physical Travelling Salesman Problem. In *Proceedings of EvoApplications*, pages 255–264, 2012a.

D. Perez, P. Rohlfshagen, and S. Lucas. Monte Carlo Tree Search: Long Term versus Short Term Planning. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 219–226, 2012b.

D. Perez, P. Rohlfshagen, and S. Lucas. The Physical Travelling Salesman Problem: WCCI 2012 Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1–8, 2012c.

D. Perez, P. Rohlfshagen, D. Robles, and S. Lucas. The PTSP Competition, June-August 2012d. www.ptsp-game.net.

D. Perez, S. Samothrakis, and S. Lucas. Knowledge-based Fast Evolutionary MCTS for General Video Game Playing. In *Proceedings of the Conference on Computational Intelligence and Games (CIG) (submitted)*, 2014c.

D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen. Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games. In *Proc. of the Conference on Genetic and Evolutionary Computation (GECCO)*, pages 351–358, 2013b.

D. Perez, S. Samothrakis, and S. M. Lucas. Online and Offline Learning in Multi-Objective Monte Carlo Tree Search. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 121–128, 2013c.

D. Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas. The General Video Game AI Competition, August 2014d. www.gvgai.net.

D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen, and S. Lucas. Automated Map Generation for the Physical Travelling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, PP, doi: 10.1109/TEVC.2013.2281508:1–14, 2013d.

D. Powell and M. M. Skolnick. Using Genetic Algorithms in Engineering Design Optimization with Non-Linear Constraints. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 424–431, 1993.

E. Powley, D. Whitehouse, and P. I. Cowling. Monte Carlo Tree Search with Macro-Actions and Heuristic Route Planning for the Physical Travelling Salesman Problem. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 234–241, 2012.

E. J. Powley, D. Whitehouse, and P. I. Cowling. Determinization in Monte-Carlo Tree Search for the card game Dou Di Zhu. In *Proceedings Artificial Intelligence and the Simulation of Behaviour*, pages 17–24, York, United Kingdom, 2011.

E. J. Powley, D. Whitehouse, and P. I. Cowling. Monte Carlo Tree Search with Macro-Actions and Heuristic Route Planning for the Multiobjective Physical Travelling Salesman Problem. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 73–80, 2013.

C. R. Reeves and J. E. Rowe. *Genetic Algorithms-Principles and Perspectives: a Guide to GA Theory*, volume 20. Kluwer Academic Pub, 2002.

D. Robles and S. M. Lucas. A Simple Tree Search Method for Playing Ms. Pac-Man. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 249–255, 2009.

D. Robles, P. Rohlfshagen, and S. M. Lucas. Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 305–312, 2011.

P. Rohlfshagen and S. M. Lucas. Ms Pac-Man Versus Ghost Team CEC 2011 Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, page to appear, 2011.

S. J. Russell. Rationality and Intelligence. *Artificial Intelligence*, 1:57–77, 1997.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

A. Saffidine. *Some Improvements for Monte-Carlo Tree Search, Game Description Language Compilation, Score Bounds and Transpositions*. Master's. thesis, Paris-Dauphine Lamsade, France, 2010.

K. Salen and E. Zimmerman. *Rules of Play : Game Design Fundamentals*. The MIT Press, 2003.

S. Samothrakis and S. Lucas. Planning Using Online Evolutionary Overfitting. In *UK Workshop on Computational Intelligence*, pages 1–6. IEEE, 2010.

S. Samothrakis, D. Robles, and S. M. Lucas. A UCT Agent for Tron: Initial Investigations. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 365–371, 2010.

S. Samothrakis, D. Robles, and S. M. Lucas. Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

T. Schaul. A Video Game Description Language for Model-based or Interactive Learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 193–200, 2013.

J. Schmidhuber. Developmental Robotics, Optimal Artificial Curiosity, Creativity, Music, and the Fine Arts. *Connection Science*, 18:173–187, 2006.

M. Schoenauer and Z. Michalewicz. Evolutionary computation at the edge of feasibility. *Lecture Notes in Computer Science*, 1141:245–254, 1996.

M. Schoenauer and S. Xanthakis. Constrained GA optimization. In *In Proc. of 5th Int'l Conf. on Genetic Algorithms*, pages 573–580, 1993.

N. Shaker, J. Togelius, G. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten. The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:4: 332–347, 2011.

S. Sharma, Z. Kobti, and S. Goodwin. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pages 49–55, 2008.

S. E. Siwek. Video Games in the 21st Century: The 2010 Report. Technical report, Entertainment Software Association., 2010.

N. Sorenson, P. Pasquier, and S. DiPaola. A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:229–244, 2011.

K. O. Stanley and R. Miikkulainen. A Taxonomy for Artificial Embryogeny. *Artificial Life*, 9(2):93–130, 2003.

S. Still and D. Precup. An Information-Theoretic Approach to Curiosity-Driven Reinforcement Learning. *Theory in Biosciences*, 131(3):139–148, 2012.

R. Sutton and A. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.

C. Szepesvári. *Algorithms for Reinforcement Learning*. Morgan & Claypool Publishers, 2010.

N. G. P. D. Teuling and M. H. M. Winands. Monte-Carlo Tree Search for the Simultaneous Move Game Tron. In *Computer Games Workshop at ECAI*, pages 126–141, Montpellier, France, 2012.

J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 Mario AI Competition. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 3:172–186, 2011.

P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker. Empirical Evaluation Methods for Multiobjective Reinforcement Learning Algorithms. *Machine Learning*, 84:51–80, 2010.

G. Van Eyck and M. Müller. Revisiting Move Groups in Monte-Carlo Tree Search. *Advances in Computer Games*, 7168:13–23, 2012.

B. Weber, M. Mateas, and A. Jhala. Building Human-Level AI for Real-Time Strategy Games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, 2011.

W. Weijia and M. Sebag. Multi-objective Monte Carlo Tree Search. In *Proceedings of the Asian Conference on Machine Learning*, pages 507–522, 2012.

W. Weijia and M. Sebag. Hypervolume Indicator and Dominance Reward Based Multi-Objective Monte-Carlo Tree Search. *Machine Learning*, 92:2–3:403–429, 2013.

J. Whitehead. Toward Procedural Decorative Ornamentation in Games. In *Proceedings of the Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

M. H. M. Winands and Y. Björnsson. Evaluation Function Based Monte-Carlo LOA. In *Proc. Adv. Comput. Games, LNCS 6048*, pages 33–44, Pamplona, Spain, 2010.

G. N. Yannakakis and J. Togelius. Experience-driven Procedural Content Generation. *IEEE Transactions on Affective Computing*, 2:147–161, 2011.

Q. Zhang and H. Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective Evolutionary Algorithms: A Survey of the State of the Art. *Swarm and Evolutionary Computation*, 1:32–49, 2011.

E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. Shaker Verlag, Germany, TIK-Schriftenreihe Nr. 30, Diss ETH No. 13398, Swiss Federal Institute of Technology (ETH) Zurich, 1999.

List of Figures

List of Tables