The 18th European Conference on Artificial Intelligence

Proceedings

Workshop on Artificial Intelligence in Games

Tuesday July 22, 2008
Patras, Greece

Adi Botea and Carlos Linares López
# Table of Contents

A Minimax Tutor for Learning to Play a Board Game  
*Dimitris Kalles and Panagiotis Kanellopoulos* .......................... 10

Agents Making Moral Decisions  
*Jaspreet Shaheed and Jim Cunningham* ................................. 15

Connecting PDDL-based Off-the-shelf Planners to an Arcade Game  
*Olivier Barthe and Éric Jacopin* ........................................ 20

Using Abstraction in Two-Player Games  
*Mehdi Samadi, Jonathan Schaeffer, Fatemeh Torabi Asr, Majid Samar and Zohreh Azimifar* ...................................................... 25

Bayesian Iteration: Online Learning in Timed Zero-Sum Games with Unknown Enemy  
*Hirotaka Moriguchi, Fuyuki Ishikawa and Shinichi Honiden* ............ 30

A Heuristic for Scrabble Based in Probability  
*Alejandro González Romero, Francisco González Acuña, Arturo Ramírez Flores, Amador Roldán Aguilar, Rene Alquézar and Enric Hernández* ............. 35

Framework for Evaluating Believability of Non-player Characters in Games  
*Tero Hinkkanen, Jaakko Kurhila and Tomi A. Pasanen* .................. 40

LPI: Approximating Shortest Paths using Landmarks  
*Kevin Grant and David Mould* ............................................. 45

Modelling a RTS Planning Domain with Cost Conversion and Rewards  
*Vidal Alcázar, Daniel Borrajo and Carlos Linares López* ............... 50

Towards Engaging MDPs  
*Olana Missura, Kristian Kersting and Thomas Gärtner* ................. 55
Programme Committee

Yngvi Björnsson, Reykjavik University
Adi Botea, NICTA and Australian National University
Stefan Edelkamp, Dortmund University
Ian Frank, Future University, Hakodate, Japan
Pedro Antonio González Calero, Universidad Complutense de Madrid
Jaap van den Herik, Maastricht University
Andreas Junghanns, QTronic GmbH
Akihiro Kishimoto, Future University, Hakodate, Japan
Peter Kissmann, Dortmund University
Carlos Linares López, Universidad Carlos III de Madrid
Héctor Muñoz-Ávila, Lehigh University
Alexander Nareyek, National University of Singapore
Dana Nau, University of Maryland
Jeff Orkin, MIT
Scott Sanner, NICTA and Australian National University
Preface

From an academic perspective, games are an excellent testbed for many AI fields. Puzzles and board games are typical applications of areas such as heuristic search and learning. Video games contain a large diversity of domains for both single-agent and multi-agent problems. They may or may not include features such as uncertainty, dynamic environments, and hidden information. New ideas in fields such as planning, search and learning can be developed and tested using a game environment as a testbed.

On the practical side, games are a booming, multi-billion dollar industry. Recent years have seen an increasing focus on making the inhabitants of a game world smarter. The quality of a game is directly impacted by character skills such as navigating on a map, acting according to a meaningful plan, and learning from previous experience, which need to be implemented using AI algorithms.

The Workshop on AI in Games AIG-08 was originally targeted to both academic researchers and practitioners that actively try to use new technologies in game development. Submissions from all over the world were received including: Finland, Greece, United Kingdom, Spain, France and Germany. However, the workshop has also attracted the attention of researchers outside Europe and submissions were made also from Japan, Canada, Malaysia and Iran, for a total of 14 papers. Unfortunately, not all of them could be accepted, and 10 papers have been finally selected for publication, what stands for an acceptance ratio of 71%.

From a scientific point of view, the workshop covers a wide range of technologies, currently employed in the design and development of any sort of games. Most remarkably, papers on planning, learning, heuristic search and agents can be found herein. Regarding the nature of games under consideration, it became obvious that the scientific community is currently giving a broad attention to a large number of different games including but not limited to: board games, real-time strategy games or even arcade games.

Very remarkably also, Jonathan Schaeffer, gave an invited talk, exactly a year after he and his colleagues from the University of Alberta (Canada) made public they solved the game of checkers.

The event continues previous efforts to bridge the gap between the two communities, and it is expected to be continued by future events, hopefully co-located with the European Conference on Artificial Intelligence.

Adi Botea and Carlos Linares López
Canberra, Australia and Madrid, Spain
June, 2008
Acknowledgements

The workshop organizers do acknowledge, first of all, the work done by all authors, even if their work was not finally selected for publication. We also feel in debt with Jonathan Schaeffer, from the University of Alberta, who gently accepted to give an invited talk at the workshop.

Obviously, this workshop could not have been held without the invaluable help of all the programme committee members, who carefully reviewed the papers and provided significant feedback to both the authors and the workshop organizers. Thanks for the hard work!

At last, but not least, we would also like to publicly acknowledge the cooperation provided by the ECAI’08 conference organizers, very especially to Pavlos Peppas.
A Minimax Tutor for Learning to Play a Board Game

Dimitris Kalles\textsuperscript{1} and Panagiotis Kanellopoulos \textsuperscript{2}

Abstract. We investigate systematically the impact of a minimax tutor in the training of computer players in a strategy board game. In that game, computer players utilise reinforcement learning with neural networks for evolving their playing strategies. A traditionally slow learning speed during conventional self-play is substantially improved upon when minimax is employed; moreover, the ability of minimax itself to win games serves as a very powerful tutor for its opponents who must develop fast effective defensive strategies. Such strategies are eventually shown to be quite good when deployed against a player that cannot draw on minimax and must play utilising whatever it has learnt.

1 Introduction

Several machine learning concepts have been tested in game domains, since strategic games offer ample opportunities to automatically explore, develop and test winning strategies. The most widely publicised results occurred during the 1990s with the development of Deep Thought and Deep Blue by IBM but the seeds were planted as early as 1950 by Shannon \cite{1} who studied value functions for chess playing by computers. This was followed by Samuel \cite{2} who created a checkers program and, more recently, by Sutton \cite{3} who formulated the TD(\(\lambda\)) method for temporal difference reinforcement learning. TD-Gammon \cite{4, 5, 6} was the most successful early application of TD(\(\lambda\)) for the game of backgammon. Using reinforcement learning techniques and after training with 1.5 million self-playing games, a performance comparable to that demonstrated by backgammon world champions was achieved. Very recently, Schaeffer et al. \cite{7} proved that the game of checkers is a draw with perfect play from both players, while the game of Go \cite{8} has also been studied from an AI point of view.

Implementing a computer’s strategy is the key point in strategy games. By the term strategy we broadly mean the selection of the computer’s next move considering its current situation, the opponent’s situation, consequences of that move and possible next moves of the opponent. In our research, we use a strategy game to gain insight into how we can develop (in the sense of evolving) game playing capabilities, as opposed to programming such capabilities (using mini-max, for example). Although the operational goal of achieving improvement (measured in a variety of ways) is usually achieved in several experimental settings \cite{9, 10}, the actual question of which training actions help realize this improvement is central if we attempt to devise an optimized training plan. The term optimize reflects the need to expend judiciously the training resources, be it computer power or human guidance.

Previous work \cite{11, 12, 13, 14} has shown that the strategy game backgammon under consideration in this paper is amenable to basic design verification using reinforcement learning and neural networks. The problem that we aim to highlight in this paper is that, even with the help of a sophisticated tutor, as implemented by a minimax algorithm, learning cannot be straightforward to automate without careful experimental design.

For this reason we have designed, carried out and analyzed several experimental sessions comprising in total about 25,000 simple computer-vs.-computer games and about 500 complex computer-vs.-computer games. In complex games, one of the players was following the recommendations of a minimax algorithm, deployed at increasing levels of look-ahead (and incurring, accordingly, significantly increasing computational costs). We believe that the results are of interest as they indicate that increasing the look-ahead does not necessarily lead to increasing the quality of the learned behaviour and that a pendulum effect is present when two players compete and one of them is temporarily aided by a knowledgeable tutor.

The rest of this paper is organised in four subsequent sections. The next section presents the details of the game, including rules for legal pawn movements, a review of the machine learning context, which includes some reinforcement learning and neural network aspects, and a brief review of the to-date experimental findings. We then describe the experimental setup, presented in distinct sessions, each of which asks a specific question and presents data toward answering that question. We then discuss the impact and the limitations of our approach and identify recommended directions for future development. The concluding section summarises the work.

2 A board game in a nutshell

The game is played on a square board of size \(n\), by two players. Two square bases of size \(\alpha\) are located on opposite board corners. The lower left base belongs to the white player and the upper right base belongs to the black player. At game kick-off each player possesses \(3\) pawns. The goal is to move a pawn into the opponent’s base.

The base is considered as a single square, therefore every pawn of the base can move at one step to any of the adjacent to the base free squares. A pawn can move to an empty square that is vertically or horizontally adjacent, provided that the maximum distance from its base is not decreased (so, backward moves are not allowed). Note that the distance from the base is measured as the maximum of the horizontal and the vertical distance from the base (and not as a sum of these quantities). A pawn that cannot move is lost (more than one pawn may be lost in one round). If some player runs out of pawns he loses.

In Figure 1 some examples and counterexamples of moves are presented. The upper board demonstrates a legal and an illegal move (for the pawn pointed to by the arrow - the illegal move is due to the rule

---

\textsuperscript{1} Hellenic Open University, Sachtouri 23, 26222 Patras, Greece. Email: dkalles@acm.org

\textsuperscript{2} Research Academic Computer Technology Institute & Department of Computer Engineering and Informatics, University of Patras, 26500 Rio, Greece. Email: kanellop@ceid.upatras.gr
that does not allow decreasing the distance from the home base). The lower boards demonstrate the loss of pawns (with arrows showing pawn casualties), where a “trapped” pawn automatically draws away from the game. As a by-product of this rule, when there is no free square next to a base, the rest of the pawns of the base are lost.

The game is a discrete Markov procedure, since there are finite states and moves and each action depends only on the current configuration on the board and not on how this configuration was obtained; therefore, the Markov property (see for example [15], Section 3.5) is satisfied. The a a priori knowledge of the system consists of the rules only.

Reinforcement learning is quite good at helping explore the state space of such games when it comes to learning how to play (as opposed to being instructed), for example by observing a tutor or an opponent. In theory, the advantage of reinforcement learning to other learning methods is that the target system itself detects which actions to take via trial and error, with limited need for direct human involvement. The goal is to learn an optimal policy that will maximize the expected sum of rewards in a specific time, determining which action should be taken next given the current state of the environment.

Before moving on, we reiterate some basic nomenclature. By state \( s \) we mean the condition of a physical system as specified by a set of appropriate variables. A policy determines which action should be performed in each state; a policy is a mapping from states to actions. Reward \( r \) is a scalar variable that communicates the change in the environment to the reinforcement learning system. For example, in a missile controller, the reinforcement signal might be the distance between the missile and the target (in which case, the RL system should learn to minimize reinforcement). The value \( V(s) \) of a state is defined as the sum of the rewards received when starting in that state and following some fixed policy to a terminal state. The optimal policy would therefore be the mapping from states to actions that maximizes the sum of the rewards when starting in an arbitrary state and performing actions until a terminal state is reached. The value function is a mapping from states to state values and can be approximated using any type of function approximation (e.g., multi-layered perceptron, radial basis functions, look-up table, etc.).

Temporal difference (TD) learning is an approach to RL, based on Monte Carlo and dynamic programming. TD methods update estimates based on Monte Carlo and dynamic programming, without waiting for a final outcome (bootstrapping). Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to \( V(s) \) (only then is the reward known), TD methods need only wait until the next time step. Eligibility traces are one of the basic mechanisms of reinforcement learning. They can be seen as a temporary record of the occurrence of an event, such as the visiting of a state. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error. Values are backed up according to the following equation:

\[
V(s)_{\text{new}} = V(s)_{\text{old}} + \alpha e(s) [r + V(s') - V(s)],
\]

where \( s \) is the state-position, \( V(s') \) its value, \( e(s) \) the eligibility trace, \( r \) the reward from the transition, \( \alpha \) the learning rate and \( s' \) the resulting state-position.

We now proceed to discuss some implementation issues concerning the actual learning procedure. Each player approximates the value function on its state space with a neural network. The input layer nodes are the board positions for the next possible move plus some flags depending on the number of surviving pawns and on the adjacency to an enemy base. The hidden layer consists of half as many hidden nodes and there is just one output node; it serves as the degree to which we would like to make a specific move. At the beginning all states have the same value except for the final states. After each move the values are updated through TD(\( \lambda \)) [15]. The algorithm used for the training was “vanilla” backpropagation, with \( \gamma = 0.95 \) and \( \lambda = 0.5 \). By using \( \gamma \neq 1 \), we favour quick victories, as the reward decreases over time. Network weights constitute a vector \( \theta \) where updates occur according to \( \theta_{t+1} = \theta_t + \alpha \delta_t \bar{e}_t \), where \( \delta_t \) is the TD error, \( \delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \) and \( \bar{e}_t \) is a vector of eligibility traces, one for each component of \( \theta_t \), updated by \( \bar{e}_t = \gamma \lambda \bar{e}_{t-1} + \gamma \bar{e}_{t} V(s_t) \) with \( \bar{e}_0 = \bar{e} \). All nodes use the nonlinear sigmoid function

\[
h(j) = \frac{1}{1 + e^{-\sum_{i=1}^{d} \theta_i o(i)}}
\]

of which the values range from 0 to 1. In order to avoid local minima and encourage exploration of the state space, a commonly used starting \( \epsilon \)-greedy policy with \( \epsilon = 0.9 \) was adopted, i.e., the system chooses the best-valued action with a probability of 0.9 and a random action with a probability of 0.1.

Note that, drawing on the above and the game description, we may conclude that we cannot effectively learn a deterministic optimal policy. Such a policy does exist for the game [16], however the use of an approximation effectively rules out such learning. Of course, even if that was not the case, it does not follow that converging to such a policy is computationally tractable [17].

Earlier experimentation [11] initially demonstrated that, when trained with self-playing games, both players had nearly equal opportunities to win and neither player enjoyed a pole position advantage. Follow-up research [12] furnished preliminary results that suggested a computer playing against itself would achieve weaker performance when compared to a computer playing against a human player. More recently that line of research focused on the measurable detection of improvement in automatic game playing, by constraining the moves of the human (training player), while experimenting with different options in the reward policies [13] and with varying game workflows [14].

3 The experimental setup

To investigate the effect of employing a tutor at several sophistication levels, we devised a set of experiments along the following stages and
associated objectives:

1. One session of 1,000 computer-vs.-computer (CC) games. The objective was to generate a baseline reference.
2. Five sessions of 100 computer-vs.-computer (MC) games each, where the white player used minimax to determine its next move. Each session involved a different look-ahead; we experimented with look-aheads 1, 3, 5, 7 and 9 (note that a look-ahead of $2n + 1$, denoted by MC$_{2n+1}$, indicates $n + 1$ moves for the white player and $n$ moves for the black player). The objective was to train the white players by tutors of increased sophistication.
3. Five sessions of 1,000 CC games each, where each session was based on one of the previous stage (for example, the MC$_3$ session was followed by a 1,000 CC session). The objective was to examine how the white player did when the tutor was absent, as well as how the black player reacted when its opponent lost expert support.
4. A tournament among all MC variants. A comparison between variants $X$ and $Y$ is done in two steps of 1,000 CC games each where, in the first step the white player of the $X$th batch plays against the black player of the $Y$th batch, and in the second step, the white player of the $Y$th batch plays against the black player of the $X$th batch. The objective was to measure the quality of deep look-ahead, which is an expensive undertaking.

All experiments were made on 8 × 8 boards with 2 × 2 bases, with 10 pawns for each player.

### 3.1 The tabula rasa case

The first stage delivered 490 games won by the white player and 510 games won by the black player. However, the white player needed an average of 630 moves to win each game, whereas the black player only required 438 moves on average.

On closer inspection of intermediate steps (by examining the first 1/10-th and then the fifth 1/10-th of experiments), we observed that the balance of games won was never really disturbed, whereas the average number of moves per game won fluctuated widely.

It is reasonable to declare that session a draw - and a good reference point. It also confirmed earlier similar findings [11].

### 3.2 The minimax case: early and standard

When deploying a minimax tutor, a certain level of look-ahead is required.

Note that the white player’s network is always updated. This reflects that the white player attempts to build a playing model based on its (minimax) tutor. That model is also used whenever minimax examines a leaf state (in the minimax tree) that is not also a final game; we use that state’s value as a surrogate for the minimax value which we cannot compute.

When we examine MC experiments, we expect that the white player’s performance will be improved the longer we allow experimentation to carry on. There is a simple reason for that: the white player is better situated to observe winning states and then update its neural network (with credit, however, only marginally being able to flow back to states that correspond to game openings).

### 3.3 Judging a tutor by the impact of absence

When the minimax tutor is absent, the black player has a learned behaviour that can be effectively deployed. The white player, however, may be able to better deal with end-games; therein look-ahead is easier to deliver a win and subsequently update the neural network (with credit, however, only marginally being able to flow back to states that correspond to game openings).

There are two easily identifiable alternatives for game development given that the two players have roughly similar learning capabilities (see section 3.1).

One option is that the relative distribution of wins will not change much from the numbers reported in the relevant columns (MC 100) of Table 1.

Another option, however, is that the black player, which has had to sustain a battering by an unusually effective opponent (the minimax player), has also had to improve itself as much as possible due to such harsh circumstances. In that case, we would expect that the black player should have developed quite an effective defense. Since both players are now not really adept at offense, it should be that both could converge to an equal number of games being won by each side in the long run and, as a side effect, games should also take longer to complete.

The results are shown in Table 2.

We observe, that with the notable exception of MC$_1$ experiments, where the black player actually becomes faster at winning when the CC session takes longer, actually allowing the CC session to evolve makes both players slower.

It is very interesting that the first part of the CC session, which consists of the first 100 games, invariably shows a dramatic increase in the games won by black. That increase is dramatic throughout, and not just for MC$_1$ experiments, where one might be tempted to say that no real minimax is actually employed.

It is also very interesting that a deeper look-ahead is associated

<table>
<thead>
<tr>
<th>Look-ahead</th>
<th>Games Won</th>
<th>Average # of Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 MC</td>
<td>100 MC</td>
</tr>
<tr>
<td></td>
<td>W B W B W B W B</td>
<td>10 MC 100 MC</td>
</tr>
<tr>
<td>1</td>
<td>1 7 3 93 7 30 67 34</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>6 4 93 7 19 31 17 45</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>4 9 4 91 9 21 19 17</td>
<td>74</td>
</tr>
<tr>
<td>7</td>
<td>3 7 82 18 62 102 54 181</td>
<td>181</td>
</tr>
<tr>
<td>9</td>
<td>10 0 89 11 17 21 14</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1. The evolution of minimax tutoring.
3.4 Judging a tutor by a student tournament

Judging a tutor by a student tournament really affects the percentage of games won by each player. Experiments, where the increase in the length of the CC session does not exceed the number actually used in this paper. That may be particularly true for the MC experiments; we believe that this is a signal that a learning stalemate is being reached. That may be particularly true for the MC experiments, where the increase in the length of the CC session does not really affect the percentage of games won by each player.

### Table 2. The evolution of post-minimax self play.

<table>
<thead>
<tr>
<th>Look-ahead</th>
<th>Games Won</th>
<th>Average # of Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100 1 CC</td>
<td>1000 1 CC</td>
</tr>
<tr>
<td>W</td>
<td>B</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>72</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>23</td>
<td>77</td>
</tr>
</tbody>
</table>

The evolution of post-minimax self play.

with a more dramatic decrease in how white does; however, we note that for a look-ahead value of more than 5, that trend is eventually reversed. Maybe, that signifies that the white player initially pays the price for the absence of its tutor, yet loses so many games that it is also forced to update that part of its learning structure that deals with effective defence and eventually manages to counter the attack. Where this eventually is not possible (see MC3 and MC9 experiments), games take the longest to conclude among the observed experiments; we believe that this is a signal that a learning stalemate is being reached. That may be particularly true for the MC experiments, where the increase in the length of the CC session does not really affect the percentage of games won by each player.

### Table 3. Comparing learning batches X and Y.

<table>
<thead>
<tr>
<th>Games Won</th>
<th>Average # of Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
</tr>
<tr>
<td>White X</td>
<td>Black Y</td>
</tr>
<tr>
<td>White Y</td>
<td>Black X</td>
</tr>
</tbody>
</table>

### Table 4. Tournament results (on 1, 000 games).

<table>
<thead>
<tr>
<th>Black</th>
<th>White</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>-S-</th>
<th>-R-</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-S-</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-R-</td>
<td></td>
<td>-24</td>
<td>-84</td>
<td>-54</td>
<td>-82</td>
<td>-202</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5. Tournament results (on 100 games).

The results are quite interesting. The MC3 session did surprisingly well, both for the white player (132 more wins) and the black player (744 more wins; note the negative sign for black players). Moreover, its white player is just a close second to the white player of MC7; MC3 is plain bad.

It is instructing to compare the above results to their snapshot at only 100 CC games, as shown in Table 5.

### Discussion

We definitely need more experiments if we are to train (and not program) computer players to a level comparable to that of a human player. The options considered to-date [13, 14] range from experimentation with a wealth of parameters of the reinforcement learning and neural network parameters, with the input-output representation of the neural network, or with alternative reward types or expert playing policies.

We have developed our paper along the last recommendation, but we believe that this may affect a decision on how to deal with the other options as well.

In particular, we note that in the experiments detailed in earlier studies of this game [11, 12, 13, 14], no indication was observed of the pendulum effect; indeed, any interesting patterns of behaviour eventually surfaced after the number of self-play games greatly exceeded the number actually used in this paper.
A possible explanation for this behaviour is that, indeed, the parameters of both the reinforcement learning and neural network infrastructure are inadequately specified to capture training on behalf of the white player. However, even observing the pendulum effect and relating it to an explanation that is not outside the realm of human-to-human tutoring (namely, that one learns to do well when facing a competent player), is a finding that, at the very least, justifies the term “artificial intelligence”.

When viewed from the pendulum effect viewpoint, the finding that increasing the look-ahead brings about quite a disturbance in the winning patterns of CC games is less surprising. To be able to home on a more precise explanation, we must scale experimentum up to at least MC₁₀ experiments, since we need at least 10 moves to move a pawn out of its home base and into the enemy base (we say at least, because an attacking pawn may have to move around a defending one and such a manoeuvre could increase the overall path to the target base).

The above development directions notwithstanding, there is a key technical development that merits close attention. This is the implementation of the actual minimax algorithm; a brute force approach, as is currently employed, is quite expensive and its scaling leaves a lot to be desired.

Overall, we believe that an interesting direction for future work is determining whether the pendulum effect is due to the introduction of the minimax tutor or if it relates to the experimental setup (i.e., number of games, board size, learning parameters, etc.). Having said that, the recommendations set out in previous treatments of this game [13, 14] are still valid. A meta-experimentation engine [18] that would attempt to calculate (most probably, via evolution) good reinforcement learning and neural network parameters, as well as design a series of minimax-based training games, seems quite promising. Yet again, however, it is interactive evolution that seems to hold the most potential. While intuitively appealing, earlier experimental workflows [14] had to be customized for the requirements of this paper. This was a very human-intensive effort, costing well beyond what the actual experiments cost. Yet, it is exactly the design of experimental sessions that could help uncover interesting learning patterns, such as the pendulum effect. While a meta-experimentation game could in theory deliver an excellent computer player, it would obviously subtract from our research effort some of the drive that is associated with the discovery of interesting learning phenomena.

5 Conclusion

This paper focused on the presentation of carefully designed experiments, at a large scale, to support the claim that expert tutoring can measurably improve the performance of computer players in a board game. In our context, the expert role is assumed by a minimax player.

After elaborating on the experimental setup, we presented the results which are centered on two key statistics: number of games won at the beginning and at the end of a session.

The computation of these statistics is a trivial task, but the key challenge is how to associate them with the actual depth of the tutoring expertise. As minimax offers a straightforward way to control such depth via its look-ahead parameter, it is tempting to consider such task as an easy one, however we have found that the quality of the training did not necessarily increase with increasing look-ahead.

The AI toolbox is full of techniques that can be applied to the problem of co-evolutionary gaming and beyond [19]. Still, however, streamlining the experimentation process in game analysis is more of an engineering issue. As such, it calls for productivity enhancing tools, especially so if we also attempt to shed some light into the dynamics of intelligent systems and how they relate to identifiable traits of human cognition.

ACKNOWLEDGEMENTS

This paper and the results reported herein have not been submitted elsewhere. All previous related work by the same authors has been referenced and properly acknowledged. The code is available on demand for personal academic research purposes, as well as the complete design of the experimental sequences and the related results. Finally, the authors wish to thank the reviewers for useful comments and suggestions.

REFERENCES

Abstract. As the environments that intelligent agents operate in become more reflective of the real world, agents’ decision-making processes must become more nuanced. In this paper, we present a decision-making model for an intentional agent that has been inspired by Kohlberg’s theory of moral development and the appraisal theory of emotion. Agents utilising this model anticipate how undertaking actions will make both themselves and other agents feel, with the agents’ sense of right and wrong helping to determine which emotions are evoked in which circumstances. We proceed to present some initial findings from runs of our agent implementation over situations from well known children’s stories.

1 Introduction

As the environments that intelligent agents operate in become more reflective of the real world and our expectations of agents become greater, agents’ decision-making processes must become more nuanced. In the case of computer games, we are likely to feel much more empathy for characters in game worlds whose decision-making we intuitively understand. In this paper we present a decision-making model for intentional agents that has been inspired by Kohlberg’s theory of moral development and the appraisal theory of emotion. We describe the theory briefly in section 2. We follow this in section 3 with an introduction to appraisal theory and the OCC model that we use to ‘ground’ our use of Kohlberg’s theory by taking morality to be feeling the right emotions in the right circumstances. Section 4 represents the main contribution of this paper—a description of our agent, equipped with the decision model. A system that utilises our agents has been implemented in Qu-Prolog, a multi-threaded extension of Prolog that provides high-level communication between threads, processes and machines. Two scenarios representing the stories of “The three little pigs” and “The pied piper of Hamelin” have been created and we present some initial findings from these scenarios. Next in section 5, we take a brief look at some related work before finally presenting some conclusions and ideas for future work in section 6.

1 Intentional agents are guided by their plans and not just reacting to events.

2 Kohlberg’s theory has attracted some criticism for seeming to be biased towards certain types of societies, but we will not explore these issues here.

2 Kohlberg’s theory of moral development

Kohlberg [7] interviewed people of different ages, telling them stories and posing them moral dilemmas based upon them. He found that whilst interviewees in the same age bracket might differ on the course of action they might suggest that characters of a story should take, the factors they took into account and the way they reached decisions were often similar. He classified responses, in so doing, identifying six distinct stages (or levels) of moral reasoning. The first two he termed ‘pre-conventional’, levels 3 and 4 ‘conventional’ and levels 5 and 6 ‘post-conventional’ [2]. He found that whilst respondents at higher levels would understand the reasoning of lower levels they would find them inadequate for responding to certain moral dilemmas and prefer the reasoning of the level they had reached.

The pre-conventional levels of moral reasoning (stages 1 and 2) are especially common in the youngest children although adults too can sometimes exhibit this kind of reasoning.

Stage 1: Individuals focus on the direct consequences that their actions will have for themselves. An action is perceived as right/wrong if the person who undertakes it is rewarded/punished and the better/worse the reward/punishment the better/worse the act must have been.

Stage 2: Right behaviour is defined by what is in one’s own best interest. Concern for others is not based on loyalty or intrinsic respect but only to a point where it might further one’s own interests. Less significance is attached to rewards and punishments. Punishments, for instance, are now regarded as an occupational hazard.

The conventional levels of moral reasoning are typical of adolescents and adults. Conventional reasoners judge the morality of their actions in comparison to societal views and expectations:

Stage 3: Individuals are receptive of approval or disapproval from other people and try to be a ‘good boy’ and ‘good girl’ and live up to others’ expectations having learnt that there is an inherent value in doing so. Level 3 reasoners now take into account relationships (and their maintenance) when judging the morality of their actions.
Stage 4: Individuals now begin to take account of laws, dictums and other social conventions not for the approval of others (as in stage 3) but because of a belief in their importance in maintaining a functioning society (including, the belief that society’s needs may often transcend one’s own).

In the post-conventional levels of moral reasoning, an individual’s sense of justice may lead them to hold critical views of laws or norms.

Stage 5: Individuals are regarded as having different values. Laws are no longer regarded as rigid dictums and where they do not promote the general welfare should be changed so as to meet the greatest good for the greatest number of people.

Stage 6: Moral reasoning is based upon universal ethical principles. Laws are valid only insofar as they are grounded in justice and a commitment to justice carries with it an obligation to disobey unjust laws. Whilst Kohlberg insisted that stage 6 exists, he had difficulty finding participants who consistently demonstrated it.

In our research to date, we have focussed on stages 2 – 4. The reasoning of stage 1 seems particularly suited to the representation of child-like characters. Meanwhile, to represent the reasoning of stages 5 and 6, mechanisms different to the ones we will outline throughout the rest of this paper are likely to be needed, with agents needing to maintain models of society rather than just other agents within the scenarios under consideration.

3 Appraisal theory and the OCC model

Appraisal theory [5] has recently become the ‘predominant psychological theory of emotion’ [12]. In appraisal theory, stimuli elicit emotions because of a person’s subjective evaluation or appraisal of them. The questions of which criteria perceived stimuli are appraised against and which reactions are triggered have been explored by a number of researchers. One of the most applied models, the OCC model [8] was proposed by Andrew Ortony, Gerald Clore and Allan Collins and is shown in figure 1.

In the OCC model, emotions are seen as reactions to three types of stimuli: events, agents and objects. Central to appraising events is their desirability with respect to goals; central to appraising agents is the praiseworthiness of their actions with reference to standards; and central to appraising objects is their appealingness as determined by attitudes.

Of the OCC emotions, the ones that we are currently using (and the mechanisms by which they are evoked) are shown in table 1. An extension to the OCC set are the emotions ‘being admired’ and ‘being reproached’ – that represent the emotions of an agent which believes other agents hold these emotions towards it. The need for these emotions will be seen shortly.

In our research we use the OCC emotions to ‘ground’ our use of Kohlberg’s theory by taking morality to be the feeling of the right emotions in the right circumstances. Table 2 shows how the OCC emotions may ‘map’ to Kohlberg’s levels.
The agent is only concerned with the mental states of other agents where they lead them to undertake actions that will ultimately cause the agent joy or distress. The agent wants to avoid the reproach of other agents, instead wanting to earn their admiration. The agents own judgement of its actions is now the most important determinant of the morality of an action.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Emotions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>joy, distress</td>
</tr>
<tr>
<td>3</td>
<td>being_admired,</td>
</tr>
<tr>
<td></td>
<td>being_reproached</td>
</tr>
<tr>
<td>4</td>
<td>pride, shame</td>
</tr>
</tbody>
</table>

Table 2. The emotions of different levels

4 Implementation of the decision model

Prolog pseudocode for the agent is shown in figure 2, it is similar to that of many BDI agents (such as AgentSpeak(L) [11]) but has been augmented with emotions.

```prolog
agent_cycle :-
    get_percepts,
    update_beliefs,
    update_emotions,
    update_mode,
    ((
        reacting = true,
        execute_intention
    );
    reacting = false,
    form_plans,
    select_plans,
    execute_intention
),
agent_cycle.
```

Figure 2. The agent cycle

After perceiving the environment (that is represented using the event calculus [9]), the agent updates its beliefs. Changes in belief lead to appraisal and an update of emotions. The update_emotions predicate not only updates the agent’s own emotions but the emotions it believes other agents are experiencing. The criteria for an agent’s appraisal includes the intrinsic pleasantness/unpleasantness of states, interference with goals/expectations/intentions (as represented by the agent’s plans) and the conformance (or non-conformance) of actions to standards.

The appraisal process is parameterised by the agent’s morality (and the morality it assumes of other agents) so, whilst an agent might be aware of a standard (for instance, ‘don’t lie’) and even expect other agent’s to adhere to it, it will only feel the emotions evoked by comparing its actions to the standard if it is a level 4 agent.

Many ontologies of emotion distinguish fullblown emotional episodes in which a protagonist may be consumed by an emotion and underlying emotions for which the relationship between emotion and action is less clear. In order to account for fullblown emotional episodes, the update_mode predicate can cease further deliberation in favour of the adoption of a particular plan that will be executed without appraisal.

Seeking a preferable emotional state drives the form_plans and select_plans predicates. Plans are formed through abduction (using [14]) with possible goals the removal of sources of unhappiness (distress) or bringing about conditions that cause happiness (joy). Appraisal of plans is central to their selection and a number of processes are involved:

- The beliefs and emotions of every agent at every state within a plan are identified—however agents’ preferences and expectations affect the emotions felt (for an agent to feel disappointed, it must have expected to be in a different, more preferable state) so until a plan is annotated with preferences and expectations an incomplete picture is produced.
- Preferred states for every agent need to be identified, using estimates of the morality of other agents. An emotional state A is preferred over an emotional state B if A − B contains a good emotion of a level higher than any in B − A or if B − A contains a bad emotion of a level higher than any in A − B. The emotional preference algorithm is inspired by Kohlberg’s observation that respondents prefer the reasoning of higher levels—so the judgement of the highest level of moral reasoning is the most important in decision-making.
- Expectations are determined by using knowledge of which agents participate in the plan together with their preferences to identify a path through the plan preferable for all involved. The agent cannot expect co-operation in plans that are not preferable for all, since, the emotional preferences it estimates for other agents already account for their desire to earn its approval and conform to certain values.
- A state may prompt a reaction from agents. In our implementation to date, the agent only modifies the states of the plan so as to account for the strong emotions evoked in agents by the plan (and the actions they may subsequently take) but not other (non-reactive) deliberation or counter-planning.
- Ultimately, if the agent has no expectations with respect to the plan (representing a path through the plan preferable to all agents whose cooperation is needed) then the plan is abandoned, otherwise it may be adopted (subject to resource constraints/absence of preferable plans).

4.1 The three little pigs

Three little pigs leave home to seek their fortunes. Two of the pigs build themselves flimsy homes and are eaten by a wolf that blows their houses down whilst one builds a sturdy brick house and ultimately foils the wolf.

The aspect of the three little pigs story that we are most interested in is the decision-making of the pigs at the start of the story—when they decide what kind of house to build. A scenario representing the story has been created. It consists of a description of the agents in the story: their names, their morality levels and their beliefs as to the levels of morality
of other agents in the scenario. It also contains event calculus axioms describing the initial situation and the effects of actions and finally rules that describe under which circumstances particular emotions are evoked. One such set of rules is:

- Having any kind of home causes joy (for the pigs).
- Building a brick house takes a lot of effort, causing distress (more so than having a home).
- Building a brick house causes pigs pride (if their level of morality $\geq 4$).
- Building houses other than a brick house cause pigs shame (if level of morality $\geq 3$).
- Pigs admire other pigs that build brick houses (if level of morality $\geq 3$).
- Pigs feel reproach towards other pigs that build houses other than brick houses (if level of morality $\geq 3$).

Given these rules, pigs set as having a low morality level (2) always choose to build straw or stick houses, whilst pigs at a high morality level (4) will always choose to build brick houses. Interestingly though, pigs at an intermediate level (3) will choose which type of house to build according to their beliefs about the other agents in the scenario—in other words, if there is no-one around who they believe will judge them (another agent with morality level $\geq 3$) they will build a straw or stick house.

This highlights scope for an interesting extension. Currently, agents’ estimations of other agents are fixed—but if agents were to assume a default level for others and then re fine that through observation, a pig might observe another pig building a solid house and then build a solid house themselves to avoid feeling bad as a result of the judgement of that pig.

In addition, the story could equally have been represented through other sets of rules—perhaps highlighting the feeling of safety that having a brick house would provide. This might correspond to a younger child’s understanding of the story: the pigs, having left home, no longer fear the punishment of their mother. The behaviour of level 1 agents would no longer be constrained (hence building straw/stick houses). A level 2 agent with a more refined/common-sense approach to self interest, more able to look after itself, might choose to build a brick house, not because of high level emotions but simply out of emotions like fear and hope.

### 4.2 The pied piper of Hamelin

In the story of the pied piper of Hamelin, a village is overrun with rats. An enigmatic stranger (the piper) offers to rid the town of rats for which the villagers promise to pay. After he fulfils his end of the bargain the villagers renege on the agreement. To punish the villagers, the piper leads away the children of the town.

Figure 3 shows the piper’s representation of his plan to get money. State $s_1$ is the initial state. In state $s_2$ the piper has offered to remove the rats from the village. In state $s_3$, the villagers agree whilst in $s_7$ they don’t. In state $s_4$ the piper has led the rats away. In $s_5$ the villagers keep the agreement whilst in state $s_6$ they break it (which leads to the piper taking the children away in state $s_{17}$).

The piper estimates both his and the villagers’ emotions in each of these states. If the piper’s own morality and his estimation of the villagers’ morality is set at level 4, he predicts that state $s_6$ will evoke the emotions shown in figure 4. He will be disappointed (because he expected to be in state $s_5$), reproachful because the villagers have broken their obligation and angry as a result of the presence of the other emotions. Meanwhile, the villagers will be ashamed of their own actions both intrinsically (shame) and because of the damage it will done to their relationship with the piper (being reproached). Additionally, they themselves will feel disappointed because they too would have preferred state $s_5$ where they might have less money but would feel better about themselves!

Table 3 describes the results of running the scenario with differing values for the piper’s morality and piper’s estimation of the villagers’ morality. The villagers’ morality and their estimation of the piper’s morality is fixed at (4,4).

Currently, no parameter settings lead to the recreation of the actual story. Some of the factors that inhibit the story emerging include:

- Agents appraise every state of a plan before selecting one rather than employing a short term lookahead. The villagers may have intended to keep the agreement when they made it but the system does not accommodate this possibility.
- Agents assume that their morality is known to other agents so even if the villagers did intend to break the agreement, they believe that the piper will see through them.

![Figure 3. Pied piper’s view (at morality level 4)](image)

![Figure 4. State s6](image)
and has only been run in small, tightly constrained environments. In future work, we plan to:

- Establish ‘relationships’ between the agents (by making admiration and reproach important to an agent only when the emotions come from particular agents).
- Utilise more of the OCC emotions, beginning with ‘happy-for’, ‘pity’, ‘gratitude’ and ‘remorse’. These emotions are important to building relationships and may lead to agents (of level >= 3) adopting other agents’ goals as their own.
- Create larger environments involving greater numbers of agents.

In addition, in order to evaluate our model, we plan to generate stories with morals by attempting to match runs of the system (with varying parameters) against a template for a moral story, in which a character of little moral virtue ends up unhappy as a (possibly indirect) consequence of their own actions. It is likely though, that some of the factors which inhibited the generation of the pied piper of Hamelin story may similarly inhibit the generation of other fable-like stories so these issues will need to be addressed.

REFERENCES


<table>
<thead>
<tr>
<th>Morality</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4,4), (3,3)</td>
<td>Agreement made and kept, irregardless of the presence/absence of the ‘take away children’ action.</td>
</tr>
<tr>
<td>(3,4), (3,3), (2,4)</td>
<td>Agreement made and kept only if the ‘take away children’ action is present in the scenario. Otherwise, the piper believes the villagers will be untroubled at breaking the agreement and he will have no recourse against them.</td>
</tr>
<tr>
<td>(4,2), (3,2)</td>
<td>Agreement not made or kept, irregardless of the ‘take away children’ action. The piper does not wish to make the agreement because he does not believe the villagers are concerned about his judgement. Additionally, since he wouldn’t be surprised/reproachful/angry if the villagers did break the agreement- he doesn’t even consider the possibility that he might end up taking the children of the village away and so realise that keeping the agreement would actually be in the villagers’ best interests.</td>
</tr>
<tr>
<td>(2,3), (2,2)</td>
<td>Agreement made and kept, irregardless of the presence/absence of the ‘take away children’ action.</td>
</tr>
</tbody>
</table>

Table 3. Pipers expectations at/assuming different levels of morality

5 Related work

We are not aware of any applications of Kohlberg’s theory in AI, but there have recently been a number of applications of appraisal theory (because of an increasing interest in the use of emotions in computing [3], [2]). ‘Double appraisal’ in which an agent predicts how its actions will make another agent feel has been utilised in the FearNot! project, an educational program that models bullies and victims in a school setting [13]. Meanwhile, the creation of characters to populate ‘story worlds’ is the concern of [10], in which emotions may trigger behavioural switches.

Appraisal equipped agents tend to behave in a way more faithful to the psychological roots of appraisal theory whereas our approach combines appraisal and planning in a different way so that agents plan in anticipation of emotions as well as because of them.

6 Conclusions and future work

In this paper, we have presented a decision-making model for intentional agents that has been inspired by Kohlberg’s theory of moral development and the appraisal theory of emotion. The agent’s distinguishing feature is its anticipation of how undertaking actions will make itself and other agents feel, a process parameterised by the agent’s sense of right and wrong, and it utilising this knowledge as the basis for its decision-making. We have presented some initial findings from runs of the agent system over scenarios representing well known children’s stories.

We believe our approach is promising, in the scenarios we have considered the differing representations that our model supports and extensions we envisage seem to correspond to quite plausible understandings of the respective stories. However, at the moment our decision-model categorises only three types of agents, utilises relatively few of the OCC emotions...
Connecting PDDL-based off-the-shelf planners to an arcade game

Olivier Bartheye and Éric Jacopin

Abstract. First, we explain the Iceblox game, which has its origin in the Pengo game. After carefully listing requirements on game playing, the contents of plans, their execution and planning problem generation, we design a set of benchmarks to select good playing candidates among currently available planners. We eventually selected two planners which are able to play the Iceblox video game well and mostly in real time. We outline our planning architecture and describe both the predicates and a couple of the operators we designed. We wish to report that we never tweaked the planners neither during the benchmarks nor during video game playing.

1 ICEBLOX

Description In the Iceblox video game [3], the player presses the arrow keys to move a penguin horizontally and vertically in rectangular mazes made of ice blocks and rocks, in order to collect coins. But flames patrol the maze (their speed is that of the penguin) and can kill the penguin in a collision; moreover, each coin is iced inside an ice block which must be cracked several times before the coin is ready for collection. The player can push (space bar) an ice block which will slide until it collides with another ice block, a rock or any of the four sides of the game. A sliding ice block stops when it collides into another ice block, a rock or any of the four sides of the game and kills a flame when passing over it. If the player pushes an ice block which is next to another ice block, a rock or a side of the game, it cannot slide freely and shall begin to crack. Once cracked, an ice block cannot move any more and thus pushing a cracked ice block eventually results, after seven pushes, in its destruction. An ice block which contains a coin slides and cracks as does an ordinary ice block. A coin cannot slide; it can only be collected once revealed after the seventh push. The player gets to the next, randomly generated, level when all coins have been collected. Killing a flame when a bee is next to this side shall stun the bee for some short time. The player is given 60 seconds to kill all the bees of any of the 16 levels of the game (after the sixteenth level, the game cycles back to the first level). Bees accelerate when reaching the 60 seconds time limit and the player is given a short extra time to kill them before they vanish, thus making the level impossible to finish. There also are pushable diamonds which cannot be collected but must be aligned to score extra points. As in any arcade game of the time, there are many bonuses, making the high score a complex goal; and both the killing of the bees and the alignment of the diamonds score differently according to the situation. We refer the reader to [9] for further details about the game.

Why Iceblox? Iceblox is an open and widely available java implementation [1, pages 264–268] of the Pengo arcade game, published by Sega in 1982 (see Figure 2). In Pengo, there are pushable and crackable ice blocks but neither rocks nor coins; the main objective of the game is to kill all the bees patrolling the ice blocks mazes. Bees hatch from eggs contained in some ice blocks; the destruction of an ice block containing eggs results in the destruction of these eggs. Bees can be killed with a sliding ice block and by a collision with the penguin when they are stunned. Pushing a side of the game when a bee is next to this side shall stun the bee for some short time. The player is given 60 seconds to kill all the bees of any of the 16 levels of the game (after the sixteenth level, the game cycles back to the first level). Bees accelerate when reaching the 60 seconds time limit and the player is given a short extra time to kill them before they vanish, thus making the level impossible to finish. There also are pushable diamonds which cannot be collected but must be aligned to score extra points. As in any arcade game of the time, there are many bonuses, making the high score a complex goal; and both the killing of the bees and the alignment of the diamonds score differently according to the situation. We refer the reader to [9] for further details about the game.

Iceblox obviously is easier than Pengo: no time constraint, uniform speed and scoring, no bonuses, ... Main targets (coins) fixed rather than moving (bees or flames), flames cannot push ice blocks and fighting them is optional, ... However, the locking of flames in a closed maze of ice blocks and rocks possess a geometrical spirit similar to that of the alignment of diamonds in the Pengo game.

Consequently, we chose Iceblox because it is a simpler start in the world of arcade video games than the commercial games of the time.
2 REQUIREMENTS

Game playing  On its way to collect coins, the planning system playing the Iceblox game shall badly either fight or avoid flames: disorganized movements or paths longer than necessary shall be allowed as long as they do not prevent the penguin from collecting coins. Both the fighting and avoidance of flames shall be realized in (near) real time: game animation might look sometimes slow or sometimes irregular but shall never stop; in particular, flames shall always be patrolling the maze, even when traditional problem solving (that is, Planning) shall take place.

Plans (what’s in a Plan?)  A Plan shall be a set of partially ordered operators which represent actions in the Iceblox domain. What kind of actions shall we allow to be part of a plan?

Let us distinguish five kinds of actions that may happen in an arcade video game such as (Pengo or) Iceblox:

1. Drawing a frame to animate a sprite; this is the pixel kind of action. Animation always takes place, even if the player does nothing: for instance, a flame is always animated in the Iceblox game, even in the case where it cannot move because it is surrounded by ice blocks.

2. Basic moving of one step left, right, up or down and pushing an ice block. This is the sprite kind of action which is uninterruptible for a certain number of pixels, usually the number of pixels of a side of the rectangle where the sprite is drawn (in the case of Iceblox, sprites are squares of 30 by 30 pixels). Each action of this kind exactly corresponds to a key pressed by the player: arrows keys to move left, right, up and down and the space bar to push an ice block.

3. Moving horizontally or vertically in a continuous manner, that is making several consecutive basic moves in the same direction; this is the path-oriented kind of action and means following a safe path in the maze.

4. Avoiding, fleeing, fighting or locking the flames in a closed maze of ice blocks and rocks. This is the problem solving kind of action and concerns survival and basic scoring. Ice block cracking until destruction to collect a coin also is a problem solving kind of action.

5. Defining goals and setting priorities on them; this is the game level strategy kind of action and is oriented towards finishing the game with the highest score. In the case of Iceblox, this means ordering the collection of coins, noticing the opportunity to lock the flames in a maze and setting its importance in finishing the level; or else seizing the opportunity to seek cover behind ice blocks of a geometrical configuration created during game playing.

Out of the five kinds of action, only push actions of kind 2, abstract Moves based on rectilinear moves of kind 3 and the coin collection action of kind 4 shall be represented as operators.

The Plans just described are not as simple as they may appear: it is not the case that either their construction shall be too simple or their use shall be redundant with the player’s actions. First, these Plans look like the plans for the storage domain which is part of the deterministic IPC benchmarks [4]. Second, as Table 1 shows, current planners agree with their non easiness. Third, there are Plans (not literally constructed but) used in commercial video-games which appear to be even shorter and simpler [7, 8].

Plan execution  The Plan execution system receives a plan from the Planning system in order to execute it in the Iceblox game. Execution of a Plan means executing the players actions (that is, actions of kind 2) corresponding to the operators of the plan, in an order compliant with the partial order of the plan. For instance, the Plan execution system shall decide of several basic moves actions to execute a Move operator in the Plan; which can include ice block pushing to facilitate the realization of the Move operator. The Plan execution system shall also decide of taking advantage of the current Iceblox situation to avoid pushing an ice block when a flame is no longer aligned with it, to choose a new weapon or to avoid unexpected flames. These decision shall result from a local analysis and no global situation assessment shall be realized to take such advantages. The plan execution system shall eventually warn the user when it terminates (whatever the outcome, success or failure). The player can terminate the Plan execution at any time by pressing any arrow key or else the space bar.

In case of emergency situations (e.g. no weapon is locally available, fleeing seems locally impossible, etc), the Plan execution system shall call for re-Planning.

Planning  Planning refers to the plan formation activity. The player shall intentionally start the planning activity by pressing a designated key (e.g. the “p” key). Once running, planning shall not stop the Iceblox game: flames shall patrol the maze and sliding ice blocks shall continue to slide while the plan formation activity is running. The user shall be warned when the activity ends, either with success or else failure (e.g. a different sound for each case). If any of the arrow keys or else the space bar (push) is pressed before the end of the planning activity, then it is terminated. The user shall be able to play at any time.

Planners  The main idea of the work reported in this paper is to determine whether any of today’s planners is able to play the Iceblox game: no planner shall be specifically designed to play Iceblox and no existing planner shall be tweaked to play Iceblox.

Due to the on-going effort of the International Planning Competition (IPC) [4], many planners are available today and most of them
accept Planning data (i.e. states, operators) written in the PDDL language. Because of this wide acceptance, the overhead of both generation and processing of PDDL shall first be ignored; writing directly to a planner’s data structures shall be considered only if game playing requirements are not met. Consequently, the Planning activity in the Iceblox domain shall take as input an initial state, a final state and a set of operators all written in the PDDL language.

Available planners are in general more ready to process PDDL text files than ready to be connected to a video game. We shall thus begin with the design of a set of Iceblox planning problems of increasing difficulty, in the spirit of the IPC: executable files for the planners, PDDL Iceblox problems files and scripts files to automate this Iceblox benchmarking process. If a planner fails these off line tests then it shall not be a good candidate for Iceblox video game playing.

What shall demonstrate that a planner is a good candidate for a connection to Iceblox? Solving the Iceblox benchmarks fast enough seems the obvious answer. Rather, the question should be: what is the time limit beyond which the current Iceblox situation is so dangerous that an action has to be taken right away, thus changing the initial state of the planning problem and consequently asking for re-planning? Iceblox has a good playability when the frame rate is about 30 frames per second, that is, when the frames coordinates (in pixel) are updated about 30 times per second. Luckily, the sprites are 30 by 30 pixels; it then takes about 1 second for a flame to move from one crossroad to another. According to the Iceblox code, a flame gets a random new direction between 1 and 4 crossroads, while keeping in mind that the new direction at the fourth crossroad might just be the same than the previous one. We can then consider that the limit is when the penguin is 4 crossroads away from a flame, which gives a plan search runtime of at most 4 seconds. If a plan has been found then it needs to be executed, which undoubtedly takes time to trigger. Consequently, the flame should not reach the fourth crossroad and since movement between two crossroads is uninterruptible, the flame should not reach the third crossroad before the plan search ends, which gives us a time limit of 3 seconds.

A planner shall be a good candidate for Iceblox video-game playing if it can (off line) solve Iceblox planning problems within the time limit of 3 seconds. This time limit sorts out dangerous flames from harmless flames.

## 3 MINIMAL ICEBLOX SITUATIONS

### Three examples

We here describe three Iceblox planning problems of increasing difficulty which we designed in order to get a set of good candidate planners. Both plan length and planning problem size shall be used as criterias of difficulty: the larger number of predicates describing both the initial and the final states and the larger number of operators in the plan solution, the more difficult the problem.

It is necessary to collect a coin in each of the three problems. Figures 3, 4 and 5 contain an Iceblox screen shot illustrating the problem and its PDDL code. See Figure 3 for the simplest of our three problems; this is obviously a welcome-to-the-Iceblox-world problem. In Figure 4, we introduce danger from one flame with only one weapon to kill this flame. Finally, in Figure 5 we provide the case for two weapons to kill a flame guarding a coin. Let us sum up the properties of these three problems:

<table>
<thead>
<tr>
<th>See Figure</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires flame fighting?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of weapons</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of predicates (initial and final states)</td>
<td>7</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Number of paths leading to the coin</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Length of solution plan</td>
<td>2</td>
<td>4</td>
<td>4 or 5</td>
</tr>
</tbody>
</table>

Why these three problems? First because their number of predicates all are very small and yet they cover the basics of Iceblox game playing: a planner failing at these problems shall not be able to play Iceblox. Moreover, although more problems have been designed, they confirm the results of Table 1. Given as Iceblox levels to an Iceblox planning system, these problems can be solved in real time; screen shots of the final Iceblox situations are gathered in Figure 6.

### Planning problems predicates

The following predicates are used to describe both the initial and final states of the planning problems of Figures 3, 4 and 5 (crossroad, denotes the location at the intersection of line i and column j):

- (at i,j): a sprite (penguin, ice block, iced coined, weapon) is at the crossroad,.
- (extracted i,j): the coin at the crossroad, has been collected by the player.
- (guard i1 j1 i2 j2): the flame at the crossroad, guards the coin at the crossroad,.
- (protected-cell i,j): there exists a path towards the crossroad,.
- (reachable-cell i,j): there exists a path towards the crossroad,; there exists at least one flame putting this path in danger.
- (weapon i1 j1 i2 j2): a weapon at the crossroad,; the penguin should push this weapon from the crossroad,i2,j2. The weapon shall stop sliding at the crossroad,i3,j3; this is useful information when an ice block needs more than one push before the final kick at a flame.

### Operators

use two more predicates:

- (blocked-path i,j): there is an ice block at the crossroad, which contains a coin.
- (protected-cell i,j): there exists a path towards the crossroad,.
- (reachable-cell i,j): there exists a path towards the crossroad,; there exists at least one flame putting this path in danger.
- (weapon i1 j1 i2 j2): a weapon at the crossroad,i2,j2; the weapon should push this weapon from the crossroad,i1,j2. The weapon shall stop sliding at the crossroad,i3,j3; this is useful information when an ice block needs more than one push before the final kick at a flame.

Over all the operators we designed, proved to be critical for Iceblox game playing: move-to-crossroad, destroy-weapon, kick-to-kill-guard and extract. We hope their names are self explanatory. There are more (e.g. pushing a block to lock flames in a maze of ice blocks and rocks) but basic Iceblox playing is impossible if you don’t get those 4 operators. Due to space limitations, we only give the PDDL code of the last two; these operators are given to the planners for benchmarking and playing:

```pddl
(aicition extract)
(parameters ("coinx" coord-i "coiny" coord-j))
(precondition and (protected-cell "coinx" "coiny")
(iced-coin "coinx" "coiny") (at "coinx" "coiny")
```
These operators can be much simpler and indeed we designed and used very simple versions of them. But of course, there is a compromise between simplicity which questions the overall utility of all this, and complexity which gets the planners nowhere. Believe us, the easiest thing to do is to put more information in the operators than necessary; and then: bye bye runtimes!

Figure 3. The danger is away, the path is obvious: change job if you don’t solve this one.

Figure 4. There is danger, but the fight is easy.

Results Several planners have been tested against these three problems (and others), with a 3.2 GHz Pentium 4 PC, loaded with 1GB of memory. The runtimes, in seconds, averaged over ten runs, are gathered in Table 1. The planners appearing in Table 1 reflect all the cases which happened during the tests: some planners rejected part or all of the PDDL files, reporting PDDL mistakes and some planners found no solution to the problems. Hopefully, others correctly read the PDDL files; of course, among those planners, some found solutions within the time limit and others did not. All the solutions found were correct (that is, no planner we tested returned an incorrect Plan).

Four planners, FF, Metric-FF, Qweak and SGPlan compose the set of good candidates for playing Iceblox. We eventually reduced the set to the well known and successful FF [2] and the less known and fastest but exotic Qweak [6].

Again, no planner was tweaked during both benchmarking and game playing (see the requirements for planners).

Table 1. Performances of several planners on PDDL files of the 3 problems of Figures 4, 5 and 6. A typed and untyped coordinates version was tested for each problem. All times, averaged over ten runs, are in seconds; planners were given 120 seconds to search for a solution although the required time limit is 3 seconds (see the requirements for planners).

4 GAME PLAYING ARCHITECTURE

There eventually are efficiency requirements to achieve the “in near real time” game playing requirement; something we clearly missed in our requirements. After several unsatisfactory prototypes, we implemented Iceblox in C++ using Microsoft’s DirectX graphical API. This turned out to be an easy way to achieve real-time playability on large game levels such as Figure 1 (since our goal was not to implement a video game but a testbed for today’s planners). Both FF and Qweak were linked to the C++ code of the Iceblox game; see Table 2 for the main game loop and Table 3 for the Plan execution process.
Table 4 presents how the collection-of-a-single-coin is generated as a PDDL planning problem during game playing. This very fast procedure finds weapons by looking in the four directions around a dangerous flame. Due to space limitation, some procedures have been left aside, such as the path planning computation. On the contrary of the Penguin system, our path planning system tries to produce rectilinear paths (see requirements for plans) and introduces ice block pushing to achieve shorter rectilinear paths.

Table 2. The main loop of the game playing system.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>While the Icebox game is not over loop</td>
<td>If the player presses a key ∈ {←, →, ↑, ↓} then</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>Check for collision; update the penguin data structure</td>
</tr>
<tr>
<td></td>
<td>If the player pressed the space bar then</td>
</tr>
<tr>
<td></td>
<td>Remember which ice block (maybe none) was pushed and in which direction</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>Else if the player presses the “p” key then</td>
</tr>
<tr>
<td></td>
<td>If the planning task is not running then</td>
</tr>
<tr>
<td></td>
<td>Build a Planning problem and start the planning task</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>For each sprite do</td>
</tr>
<tr>
<td></td>
<td>Decide what to render in the next frame</td>
</tr>
<tr>
<td></td>
<td>End for each</td>
</tr>
<tr>
<td></td>
<td>Warn the player when</td>
</tr>
<tr>
<td></td>
<td>– Plan search task ends with failure</td>
</tr>
<tr>
<td></td>
<td>– Plan execution task starts</td>
</tr>
<tr>
<td></td>
<td>– Plan execution task ends</td>
</tr>
<tr>
<td></td>
<td>End of warnings</td>
</tr>
<tr>
<td>End loop</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. The plan execution process.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat</td>
<td>If the first operator of the Plan is Move-To-Crossroad((i, j)) then</td>
</tr>
<tr>
<td></td>
<td>Compute a path from current location to crossroad, (i, j)</td>
</tr>
<tr>
<td></td>
<td>Repeat</td>
</tr>
<tr>
<td></td>
<td>Execute one basic move following that path</td>
</tr>
<tr>
<td></td>
<td>Until crossroad, (i, j) is reached</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>If the ice block at crossroad, (i, j) is not destroyed then</td>
</tr>
<tr>
<td></td>
<td>Execute a push action in direction of crossroad, (i, j)</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Execute a basic move towards crossroad, (i, j)</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>End if</td>
</tr>
<tr>
<td></td>
<td>When the flame is no longer dangerous</td>
</tr>
<tr>
<td></td>
<td>Ignore it</td>
</tr>
<tr>
<td></td>
<td>When the flame is no longer aligned with the weapon or an unexpected flame becomes dangerous</td>
</tr>
<tr>
<td></td>
<td>Find a weapon and use it</td>
</tr>
<tr>
<td></td>
<td>When the penguin must leave the computed path</td>
</tr>
<tr>
<td></td>
<td>Remember the current crossroad, (i, j)</td>
</tr>
<tr>
<td></td>
<td>Get the penguin back to crossroad, (i, j) as soon as possible</td>
</tr>
<tr>
<td></td>
<td>Until the Plan is empty</td>
</tr>
</tbody>
</table>

Table 4. The PDDL planning problem generation for the collection of a single coin.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate PDDL planning problem header</td>
<td></td>
</tr>
<tr>
<td>Generate PDDL initial state header</td>
<td></td>
</tr>
<tr>
<td>Generate (at i Penguin j-penguin)</td>
<td></td>
</tr>
<tr>
<td>Find the (nearest or possibly flame-less) iced coin</td>
<td></td>
</tr>
<tr>
<td>Generate (iced-coin i-coin j-coin)</td>
<td></td>
</tr>
<tr>
<td>If there exists a path to this coin then</td>
<td></td>
</tr>
<tr>
<td>Generate (reachable-cell i-coin j-coin)</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>Generate (blocked-cell i-coin j-coin)</td>
<td></td>
</tr>
<tr>
<td>End if</td>
<td></td>
</tr>
<tr>
<td>For each flames do</td>
<td></td>
</tr>
<tr>
<td>If this flame is close to the iced coin then</td>
<td></td>
</tr>
<tr>
<td>Generate (guard i-flame j-flame i-coin j-coin)</td>
<td></td>
</tr>
<tr>
<td>For each nearest weapon in each of the four direction do</td>
<td></td>
</tr>
<tr>
<td>Generate (weapon i-push j-push i-weapon j-weapon)</td>
<td></td>
</tr>
<tr>
<td>End for each</td>
<td></td>
</tr>
<tr>
<td>End if</td>
<td></td>
</tr>
<tr>
<td>End if</td>
<td></td>
</tr>
<tr>
<td>End for</td>
<td></td>
</tr>
<tr>
<td>Generate PDDL final state header and (extracted i-coin j-coin)</td>
<td></td>
</tr>
</tbody>
</table>

5 CONCLUSIONS

Both planners play large Iceblox levels well in near real time. Sometimes FF plays in real time while Qweak plays in real time most of the time. Due to the quality of the path planning computation and of the DirectX implementation, the penguin looks pretty rational, at the speed of the flames. Sincerely, we thought many times we’d never make it and the granularity of our Plans is crucial to achieve our goal: we just produce Planning problems with a small number of predicates. Our goal is now to work on the locking of flames and the handling of multiple flames per coin, before tackling the problem of getting the high score.

ACKNOWLEDGEMENTS

This work is part of a 3 year project founded by the Fondation Saint-Cyr. Thanks and Rick Alterman, Marc Cavazza, Bernard Conein, Jon Gratch and David Kirsh for helpful discussions; to Saint-Cyr cadets Defacqz, Marty, Pertuisel and Yvinec for their implementation of Iceblox in Adobe’s Flash; and to an anonymous reviewer for his constructive review.

REFERENCES


2 The interested reader should find an appropriate path planning algorithm in [5].
Using Abstraction in Two-Player Games

Mehdi Samadi, Jonathan Schaeffer¹, Fatemeh Torabi Asr, Majid Samar, Zohreh Azimifar²

Abstract. For most high-performance two-player game programs, a significant amount of time is devoted to developing the evaluation function. An important issue in this regard is how to take advantage of a large memory. For some two-player games, endgame databases have been an effective way of reducing search effort and introducing accurate values into the search. For some one-player games (puzzles), pattern databases have been effective at improving the quality of the heuristic values used in a search.

This paper presents a new approach to using endgame and pattern databases to assist in constructing an evaluation function for two-player games. Via abstraction, single-agent pattern databases are applied to two-player games. Positions in endgame databases are viewed as an abstraction of more complicated positions; database lookups are used as evaluation function features. These ideas are illustrated using Chinese checkers and chess. For each domain, even small databases can be used to produce strong game play. This research has relevance to the recent interest in building general game-playing programs. For two-player applications where pattern and/or endgame databases can be built, abstraction can be used to automatically construct an evaluation function.

1 Introduction and Overview

Almost half a century of AI research into developing high-performance game-playing programs has led to impressive successes, including Deep Blue (chess), Chinook (checkers), TD-Gammon (backgammon), Logistello (Othello), and Maven (Scrabble). Research into two-player games is one of the most visible accomplishments in artificial intelligence to date.

The success of these programs relied heavily on their ability to search and to use application-specific knowledge. The search component is largely well-understood for two-player games (whether perfect or imperfect information; stochastic or not); usually the effort goes into building a high-performance search engine. The knowledge component varies significantly from domain to domain. Various techniques have been used, including linear regression (as in Logistello) and temporal difference learning (as in TD-GAMMON). All of them required expert input, especially the Deep Blue [10] and Chinook [16] programs.

Developing these high-performance programs required substantial effort over many years. In all cases a major commitment had to be made to developing the program’s evaluation function. The standard way to do this is by hand, using domain experts if available. Typically, the developer (in consultation with the experts) designs multiple evaluation function features and then decides on an appropriate weighting for them. Usually the weighted features are summed to form the assessment. This technique has proven to be effective, albeit labour intensive. However, this method fails in the case of a new game or for one in which there is no expert information available (or no experts). The advent of the annual General Game Playing (GGP) competition at AAAI has made the community more aware of the need for general-purpose solutions rather than custom solutions.

Most high-performance game-playing programs are compute intensive and benefit from faster and/or more CPUs. An important issue is how to take advantage of a large memory. Transposition tables have proven effective for improving search efficiency by eliminating redundancy in the search. However, these tables provide diminishing returns as the size increases [3]. For some two-player games, endgame databases (sometimes called tablebases) have been an effective way of reducing search effort and introducing accurate values into the search. These databases enumerate all positions with a few pieces on the board and compute whether each position is a provable win, loss or draw. Each database position, however, is applicable to only one position.

The single-agent (one-player) world has also wrestled with the memory issue. Pattern databases have been effective for improving the performance of programs to solve numerous optimization problems, including the sliding-tile puzzles and Rubik’s Cube [8]. They are similar to endgame databases in that they enumerate a subset of possible piece placings and compute a metric for each (e.g., minimum number of moves to a solution). The databases are effective for two reasons. First they can be used to provide an improved lower bound on the solution quality. Second, using abstraction, multiple states can be mapped to a single database value, increasing the utility of the databases.

The main theme of this paper is to investigate and propose a new approach to use endgame and pattern databases to assist in automating the construction of an evaluation function for two-player games. The research also carries over to multi-player games, but this is not addressed in this paper. The key idea is to extend the benefits of endgame and pattern databases by using abstraction. Evaluation of a position with N pieces on the board is done by looking up a subset of pieces M < N in the appropriate database. The evaluation function is built by combining the results of multiple lookups and by learning an appropriate weighting of the different lookups. The algorithm is simple and produces surprisingly strong results. Of greater importance is that this is a new general way to use the databases.

The contributions of this research are as follows:

1. Abstraction is used to extend pattern databases (even additive pattern databases) for constructing evaluation functions for a class of two-player games.

2. Pattern-database-based evaluation functions are shown to produce state-of-the-art play in Chinese checkers (10 pieces a side). Against a baseline program containing the latest evaluation func-

¹ Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8, email: {msamadi,jonathan}@cs.ualberta.ca
² Department of Computer Science and Engineering Shiraz University, Shiraz, Iran, email: {torabi,samar,azimifar}@cs.shirazu.ac.ir
Abstraction is used to extend endgame databases for constructing evaluation functions for a class of two-player games.

Chess evaluation functions based on four- and five-piece endgame databases are shown to outplay CRAGTY, the strongest freeware chess program available. On seven- and eight-piece chess endgames, the endgame-database program scores 54% to 80% of the possible points.

Abstraction is a key to extending the utility of the endgame and pattern databases. For domains for which these databases can be constructed, they can be used to build an evaluation function automatically. As the experimental results show, even small databases can be used to produce strong game plays.

2 Related Work

Endgame databases have been in use for two-player perfect information games for almost thirty years. They are constructed using retrograde analysis [18]. Chess was the original application domain, where databases for all positions with six or fewer pieces have been built. Endgame databases were essential for solving the game of checkers, where all positions with ten or fewer pieces have been computed [6]. The databases are important because they reduce the search tree and introduce accurate values into the search. Instead of using a heuristic to evaluate these positions (with the associated error), a game-playing program can use the database value (perfect information). The limitation, however, is that each position in the database is applicable to a single position in the search space.

Pattern databases also use retrograde analysis to optimally solve simplified versions of a state space [4]. A single-agent state space is abstracted by simplifying the domain (e.g., only considering a subset of the features) and solving that problem. The solutions to the abstract state are used as lower bounds for solutions to a set of positions in the original search space. For some domains, pattern databases can be constructed so that two or more database lookups can be added together while still preserving the optimality of the combined heuristic [13]. Abstraction means that many states in the original space can use a single state in the pattern database. Pattern databases have been used to improve the quality of the heuristic estimate of the distance to the goal, resulting in many orders of magnitude reduction in the effort required to solve the sliding-tile puzzles and Rubik’s Cube [8].

The ideas presented in this paper have great potential for General Game Playing (GGP) programs [9]. A GGP program, given only the rules of the game/puzzle, has to learn to play that game/puzzle well. A major bottleneck to producing strong play is the discovery of an effective evaluation function. Although there is an interesting literature on feature discovery applied to games, to date the successes are small [7]. It is still early days for developing GGP programs, but the state of the art is hard coding into the program several well-known heuristics that have been proven to be effective in a variety of games, and then testing them to see if they are applicable to the current domain [15]. It remains an open problem how to automate the discovery and learning of an effective evaluation function for an arbitrary game.

3 Using Abstraction in Two-Player Games

Abstraction is a mapping from a state in the original search space into a simplified representation of that state. The abstraction is often a relaxation of the state space or a subset of the state. In effect, abstraction maps multiple states in the original state space to a single state in the abstract search space. Information about the abstract state (e.g., solution cost) can be used as a heuristic for the original state (e.g., a bound on the solution cost).

Here we give the background notation and definitions using chess as the illustrative domain. Let $S$ be the original search space and $S'$ be the abstract search space.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{original_abstract.png}
\caption{Original states and edges mapped to an abstract space.}
\end{figure}

**Definition 1 (Abstraction Transformation):** An abstraction transformation $\phi : S \rightarrow S'$ maps 1) states $u \in S$ to states $\phi(u) \in S'$, and 2) actions $a \in S$ to actions $\phi(a) \in S'$. This is illustrated in Figure 1.

Consider the chess endgame of white king, rook, and pawn versus black king and rook (KRKP). The original space $S$ consists of all valid states where these five pieces can be placed on the board. Any valid subset of the original space can be considered as an abstraction. For example, king and rook versus king (KRK) and king, rook, and pawn versus king (KRPK) are abstractions (simplifications) of KRKP. For any particular abstraction $S'$, the search space contains all valid states in the abstract domain (all piece location combinations). The new space $S'$ is much smaller than the original space $S$, meaning that a large number of states in $S$ are being mapped to a single state in $S'$. For instance, for every state in the abstracted KRK space, all board positions in $S$ where the white king, white rook and black king are on the same squares as in $S'$ are mapped onto a single abstract state (i.e., white pawn and black rook locations are abstracted away). Actions in $S'$ contain all valid moves for the pieces that are in the abstracted state.

**Definition 2 (Homomorphism):** An abstraction transformation $\phi$ is a homomorphism transformation if for all series of actions that transforms state $u$ to state $v$ in $S$ then there is a corresponding transformation for $\phi(u)$ to $\phi(v)$ in $S'$. This is illustrated in Figure 1, where $a^*$ represents zero or more actions.

If there is a solution for a state in the original space $S$, then the homomorphism property guarantees the existence of a solution in the abstracted space $S'$. Experimental results indicate that this characteristic can be used to improve search performance in $S$.

Various abstractions can be generated for a given search problem. The set of relaxing functions is defined as $\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, where each $\phi_i$ is an abstraction. Define the distance between any two states $u$ and $v$ in the relaxed environment as $d_i$ with $h_a(u, v)$. For example, for an endgame or pattern database, $v$ is usually set to a goal state meaning that $h_a(u, v)$ is the minimal number of moves needed to achieve the goal.

Using off-line processing, the distance from each state in $\phi_i$ to the nearest goal can be computed and saved in a database (using retrograde analysis). For a pattern database (one-player search), the minimal distance to the goal is stored. For an endgame database (two-
player search), the minimal number of moves to win (maximal moves to postpone losing) are recorded. This is the standard way that these databases are constructed.

Given a problem instance to solve, during the search all values from those lookup tables are retrieved for further processing. To evaluate a position $p$ from the original space, the relaxed state, $\phi_i(p)$, is computed and the corresponding $h_{\phi_i(p)}$ is retrieved from the database. The abstract values are saved in a heuristic vector $h = <h_{\phi_1}, h_{\phi_2}, \ldots, h_{\phi_n}>$. The evaluation function value for state $p$ is calculated as a function of $h$. For example, popular techniques used for two-player evaluation functions include temporal difference learning to linearly combine the $h_{\phi_i}$ values [1], and neural nets to achieve non-linear relations [17].

For example, let us evaluate a position $p$ in the KRPKR chess endgame. In this case, the abstracted states could come from the databases KRPK, KRKR, KRK and KPK. First, for each abstraction, the abstract state is computed and the heuristics value $h_{\phi_i(p)}$ is retrieved from the database. In this case, the black rook is removed and the resulting position is looked up in the KRPK database; the white pawn is removed and the position looked up in the KRKR database; etc. The heuristic value for $p$ could be, for example, the sum of the four abstraction scores.

### 4 Experimental Results

In this section, we explore using abstraction to apply pattern database technology to two-player Chinese checkers and chess endgame database technology to playing more complicated chess endgames. Unlike chess, Chinese checkers has the homomorphism property (the proof is simple, but not shown here for reasons of space).

#### 4.1 Chinese Checkers

Chinese checkers is a 2-6 player game played on a star shaped board with the squares hexagonally connected. The objective is to move all of one’s pieces (or marbles) from the player’s home zone (typically 10) to the opposite side of the board (the opponent’s home zone). Each player moves one marble each turn. A marble can move by rolling to an adjacent position (one of six) or by repeatedly jumping over an adjacent marble, of any color, to an adjacent empty location (the same as jumps in 8 x 8 checkers/draughts). In general, to reach the goal in the shortest possible time, the player should jump his pieces towards the opponent’s home zone.

Here we limit ourselves to two-player results, although the results presented here scale well to more players (not reported here). Due to the characteristics of Chinese checkers, three different kinds of abstractions might be considered. Given $N$ pieces on each side of the original game:

1. Playing $K \leq N$ white pieces against $L \leq N$ black pieces;
2. Playing $K \leq N$ white pieces to take them to opponent’s home zone (a pattern database including no opponent’s marble); and
3. Playing $K \leq N$ white pieces against $L \leq N$ black pieces, but with a constraint that the play concentrates on a partition of the board.

For any given search space, the more position characteristics that are exploited by the set of abstractions, the more likely that the combination of abstraction heuristics will be useful for the original problem space. The first two abstractions above have the homomorphism property, and the empirical results indicate that they better approximate the original problem space. In the first abstraction, a subset of pieces for both players (e.g., the three-piece versus two-piece game) is considered and the minimal number of moves to win (most moves to lose) is used. The second abstraction ignores all the opponent’s pieces. This abstraction gives the number of moves required to get all of one’s pieces into the opponent’s zone. This value is just a heuristic estimate (not a bound), since the value does not take into account the possibility of jumping over the opponent’s pieces (which precludes it from being a lower bound) and does not take into account interference from the opponent’s pieces (precluding it from being an upper bound). Clearly, the first abstraction is a better representation of the original problem space. The third abstraction considers only a part of the board to build a pattern database. For example, the goal of the abstraction can be changed so that the pieces only have to enter the goal area (without caring about where the end up).

The state space for the first abstraction is large; the endgame database of three versus two pieces requires roughly 256MB. The second relaxation strategy makes the search space simpler, allowing for pattern databases that include more pieces on the board. The database size for five pieces of the same side needs roughly 25MB, 10% of the first abstraction database. Our experience with Chinese checkers shows that during the game five cooperating pieces will result in more (and longer) jump moves (hence, less moves to reach the goal) than five adversarial pieces. Although the first abstraction looks more natural and seems to better reflect the domain, the second abstraction gives better heuristic values. Thus, here we present only the second and third abstractions.

The baseline for comparison is a Chinese checkers program (10 pieces a side) with all the current state-of-the-art enhancements. The evaluation function is based on the Manhattan distance for each side’s pieces to reach the goal area. Recent research has improved on this simple heuristic by adding additional evaluation terms: 1) a curved board model, incremental evaluation, left-behind marbles [19]; and 2) learning [11]. All of these features have been implemented in our baseline program.

Experiments consisted of the baseline program playing against a program using a PDB- or endgame-based evaluation function. Each experimental data point consists of a pair of games (switching sides) for each of 25 opening positions (after five random moves have been made). Experiments are reported for search depths of three and five ply (other search depth results are similar). The branching factor in the middlegame of Chinese checkers is roughly 60-80. Move generation can be expensive because of the combination of jumps for each side. This slows the program down, limiting the search depth that can be achieved in a reasonable amount of time. The average response time for a search depth of six in the middlegame is more than thirty seconds per move (1.5 hours per game). Our reported experiments are limited to depths three through five because of the wide range of experiments performed.

In this paper, we report the results for three interesting heuristic evaluation functions. Numerous functions were experimented with and achieved similar performance to those reported here. For the following abstractions, the pieces were labeled 1 to 10 in a right-to-left, bottom-up manner. The abstractions used were:

- **PDB(4):** four-piece pattern database (second abstraction) with the goal defined as the top four squares in the opponent’s home zone. Three abstractions (three lookups) were used to cover all available ten pieces: pieces 1-4, 4-7, and 7-10. We also tested other lookups on this domain. Obviously increasing the number of lookups can increase the total amount of time to evaluate each node. On the other hand, the overlap of using pieces four and seven in the evaluation function does not have a severe effect on the cost of an
evaluation function.

PDB(6): six-piece pattern database (second abstraction) with the goal defined as the top six squares in the opponent’s home zone. Two abstractions (two lookups) were used to cover all 10 pieces: pieces 1-6 and 5-10. Again, two pieces are counted twice in an evaluation (pieces 5 and 6), as a consequence of minimizing the execution overhead.

PDB(6+4): a probe from the six-piece PDB is added to a probe from the four-piece PDB (a combination of second and third abstraction). Two abstractions (two lookups) were used to cover all 10 pieces: pieces 1-6 from the PDB(6) and 7-10 from the PDB(4) with its goal defined as passing all pieces from the opponent’s front line (third abstraction). In other words, for the four-piece abstraction we delete the top six squares of the board such that the new board setup introduces our new goal.

The weighting of each probe is a simplistic linear combination of the abstraction heuristic values.

The additive evaluation function (using PDB(4+6)) gives the best empirical results. Not only is it comparable to the PDB(4), but also it achieves its good performance with one fewer database lookup per evaluation. Although the experiments were done to a fix search depth (to ensure a uniform comparison between program versions), because of the relative simplicity of the evaluation function an extra database lookup represented a significant increase in execution cost. In part this is due to the pseudo-random nature of accessing the PDB, possibly incurring cache overhead. Our implementation takes advantage of obvious optimizations to eliminate redundant database lookups (e.g., reusing a previous lookup if still applicable). By employing these optimizations, we observed that the time for both heuristic functions are very close and does not change the results.

Results are reported using four- and five-piece endgame databases to play seven- and eight-piece chess endgames. The abstracted state space is constructed using a subset of available pieces. For example, for the KRKPR endgame one can use the KRK, KRPK, and KRKR subset of pieces as abstractions of the original position. All the abstractions are looked up in their appropriate database. The endgame databases are publicly available at numerous sites on the Internet. For each position, they contain one of the following values: win (the minimum number of moves to mate the opponent), loss (the longest sequence of moves to be mated by the opponent) or draw (zero). The values retrieved from the abstractions are used as evaluation function features. They are linearly combined; no attempt at learning proper weights has been done yet.

In chess, as opposed to Chinese checkers, ignoring all the opponent pieces does not improve the performance given the tight mutual influence they have on each other (i.e., piece captures are possible). Hence pattern databases are unlikely to be effective. One could use pattern databases for chess, even though, we expect a learning algorithm to discover a weight of zero for such abstractions.

The chess abstraction does not have the homomorphism property because of the mutual interactions among the pieces. In other words, it is possible to win in the original position while not achieving this result in the abstract position. For example, there are many winning positions in the KRKPRR endgame but in the abstraction of KRKR almost all states lead to a draw.

Our experiments used the four- and five-piece endgame databases. Note that the state-of-the-art in endgame database construction is six pieces [2]. These databases are too large to fit into RAM, making their access cost prohibitively high. Evaluation functions must be fast, otherwise they can dramatically reduce the search speed. Hence we restrict ourselves to databases that can comfortably fit into less than 1GB of RAM. This work will show that even the small databases can be used to improve the quality of play for complex seven- and eight-piece endgames.

In our experiments the proposed engine (a program consisting solely of an endgame-database-based evaluation) played against the baseline program (as the opponent). Each experimental data point consisted of a pair of games (switching sides) for each of 25 endgame positions. The programs searched to a depth of seven and nine ply. Results are reported using four- and five-piece abstractions of seven- and eight-piece endgames. Because of the variety of experiments performed, the search depth was limited to nine.

The baseline considered here is CRAFTY, the strongest freeware chess program available [12]. It has competed in numerous World Computer Chess Championships, often placing near the top of the standings.

Table 2 shows the impact of two parameters on performance: the endgame database size and the search space depth. The table gives results for three representative seven-piece endgames. The first column gives the endgame, the second gives the win percentage (as stated before, wins is counted as two, draws as one and losses as zero), and
the last column shows the abstractions used. The first six lines are for a search depth of seven; the remaining six for a search depth of nine. For each depth, the first three lines show the results for using three- and four-piece databases as an abstraction; the last three rows show the results when five-piece databases are used.

RAFTY was used unchanged. It had access to the same endgame databases as our program, but it only used them when the current position was in the database. For all positions with more pieces, it used its standard endgame evaluation function. In contrast, our program, using abstraction, queried the databases every time a node in the search required to be evaluated. By eliminating redundant database lookups, the cost of an endgame-database evaluation can be made comparable to that of CRAFTY's evaluation.

Not surprisingly, the five-piece databases had superior performance to the four-piece databases (roughly 8% better for depth seven and 4% better at depth nine). Clearly, these databases are closer to the original position (i.e., less abstract) and hence are more likely to contain relevant information. Further, a significant drawback of small-size abstraction models is the large number of draw states in the database (e.g. KKKR), allowing little opportunity to differentiate between states. The five-piece databases contain fewer draw positions, giving greater decision power to the evaluation function.

As the search depth is increased, the benefits of the superior evaluation function slightly decrease. This is indeed expected, as the deeper search allows more potential errors by both sides to be avoided. This benefit the weaker program.

<table>
<thead>
<tr>
<th>Position</th>
<th>Search Depth</th>
<th>Win %</th>
<th>Abstractions Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>KOP-KRNP</td>
<td>7</td>
<td>64</td>
<td>KQKP, KQKNP, KPKRN, KQKNR</td>
</tr>
<tr>
<td>KRPP-KQRP</td>
<td>7</td>
<td>76</td>
<td>KQKP, KQKNP, KPKRN</td>
</tr>
<tr>
<td>KRPP-KRNP</td>
<td>7</td>
<td>40</td>
<td>KRPK, KRP, KPKRN</td>
</tr>
<tr>
<td>KQK-KQRP</td>
<td>7</td>
<td>76</td>
<td>KQKP, KQKNP, KPKRN</td>
</tr>
<tr>
<td>KOP-KRPP</td>
<td>7</td>
<td>64</td>
<td>KQKP, KQKNP, KPKRN</td>
</tr>
<tr>
<td>KOP-KRNP</td>
<td>9</td>
<td>64</td>
<td>KQKP, KQKNP, KPKRN, KQKNR</td>
</tr>
<tr>
<td>KRPP-KQRP</td>
<td>9</td>
<td>76</td>
<td>KQKP, KQKNP, KPKRN</td>
</tr>
<tr>
<td>KRPP-KRN</td>
<td>9</td>
<td>64</td>
<td>KRPK, KPKRN</td>
</tr>
<tr>
<td>KQPK-NKRP</td>
<td>9</td>
<td>72</td>
<td>KPKN, KQKN, KQKNP</td>
</tr>
<tr>
<td>KQKP-KQRP</td>
<td>9</td>
<td>62</td>
<td>KQKP, KQKNP, KPKRN</td>
</tr>
</tbody>
</table>

Table 3. Experiments for chess.

Table 3 shows the results for some interesting (and complicated) seven- and eight-piece endgames, all using five-piece abstraction. These represent difficult endgames for humans and computers to play. Again, the endgame-database-based evaluation function is superior to CRAFTY, winning 60% to 76% of the games. This performance is achieved using three or four abstraction lookups, in contrast to CRAFTY's hand-designed rule-based system.

Why is the endgame database abstraction effective? The abstraction used for chess is, in part, adding heuristic knowledge to the evaluation function about exchanging pieces. In effect, the smaller databases are giving information about the result when pieces come off the board. This biases the program towards lines which result in favorable piece exchanges, and avoids unfavorable ones.

5 Conclusion and Future Works

The research presented in this paper is a step towards increasing the advantages of pre-computed lookup tables for the larger class of multi-agent problem domains. The main contribution of this research was to show that the idea of abstraction can be used to extend the benefits of pre-computed databases for use in new ways in building an accurate evaluation function. For domains for which pattern and/or endgame databases can be constructed, the use of this data can be extended beyond its traditional usage and be used to build an evaluation function automatically. As the experimental results show, even small databases can be used to produce strong game play.

Since 2005, there has been interest in the AI community in building a general game-playing (GGP) program. The application-specific research in building high-performance games is being generalized to handle a wide class of games. Research has already been done in identifying GGP domains for which databases can be built [14]. For those domains, abstraction is a promising way to automatically build an evaluation function. An automated system has been developed to build a pattern database for planning domains using bin-packing algorithm to select the appropriate symbolic variables for pattern database [5]. Similar approach can be used to automatically select variables in GGP to build endgame/pattern databases.

REFERENCES

Bayesian Iteration: Online Learning in Timed Zero-Sum Games with Unknown Enemy

Hirotaka Moriguchi 1, Fuyuki Ishikawa 2 and Shinichi Honiden 1,2

Abstract. This paper discusses the problem of optimal policy selection for timed, zero-sum games. These games can be modeled as thresholded rewards Markov decision processes (trMDPs). Recently, McMillen et al. devised an efficient solution algorithm for trMDPs. With this algorithm, the agent can find the exact optimal policy efficiently. However, the algorithm can only be applied to limited problems, because it requires the parameters controlling the whole game to be completely known. To enhance the applicability of the algorithm, the authors propose an algorithm that can deal with incomplete information problems by using stochastic inference. This paper also investigates several heuristic techniques that efficiently find approximate solutions for intractable problems.

1 INTRODUCTION

Recently, there has been a lot of interest in finding the optimal policy in timed zero-sum games. Timed zero-sum games are zero-sum games with scores and limited times, for example, robot soccer. In timed zero-sum games, winning against the opponent is more important than the final score. For example, a team that is losing near the end of the game should play aggressively to even the score even if an aggressive strategy allows the opponent to score more easily. Therefore, the optimal policy that maximizes the probability of being ahead at the end of the game is generally nonstationary: the optimal action from a given state depends on the number of time steps remaining and the current score difference.

To find the optimal policy for timed zero-sum games, McMillen et al. proposed thresholded rewards Markov decision processes (trMDPs) [6]. In trMDPs, a threshold function $f$ is applied to the final cumulative reward. The trMDP problem is not to maximize the cumulative long-term reward over $h$ time steps but to maximize the expected value of $f$. As trMDPs are well designed, the nonstationary optimal policy for them can be found by using solution algorithms for general MDPs, i.e., value iteration, policy iteration, etc. [4].

The general solution algorithms for MDPs are time consuming, and finding the optimal policies for realistic trMDPs by those algorithms requires an intractable amount of computation. McMillen et al. [6] exploit the graphical characteristics of trMDPs to devise an efficient algorithm that is run-time quadratic on the number of states and the length of the time horizon. The algorithm is much faster than general solution algorithms and quite promising for solving other intractable reinforcement learning problems [10].

However, these methods all assume that the parameters of the trMDPs, in particular transition probabilities, are completely known before the game. Such assumptions are not realistic in many cases. For example in robot soccer, while an agent may know that an aggressive strategy means high risk and high return, the precise relationship between a strategy and the chance of scoring cannot be known in advance. With general MDP solution algorithms, if the agent has the wrong prior information and derives a wrong policy from it, the agent fails to behave effectively.

In this paper, we consider that the agent can only obtain incomplete information by estimating in advance and the opponent does not change its strategy in the games. As opponents' behaviors are stationary, the parameters that control the game are stationary too. The incomplete prior knowledge only provides incomplete information about the parameters of trMDPs. To deal with incomplete information problems, the authors propose an adaptive algorithm called 'Bayesian Iteration', which utilizes bayesian inference to learn the actual parameters of trMDPs. The algorithm combines prior knowledge and contemporary experience to improve the policy dynamically. However, Bayesian iteration consumes more time than non-adaptive solutions. For games with long time horizons or large state spaces, this algorithm may be intractable. Therefore, we investigate several efficient approximate solution techniques.

The remainder of this paper shows how this is achieved. Section 2 introduces MDPs and models timed zero-sum games with trMDPs. Section 3 introduces the exact solution algorithm for trMDPs. Section 4 shows how to infer the MDP's parameters with a stochastic estimation algorithm. Section 5 describes the proposed algorithm and its implementation. The results of simulation experiments are evaluated in Section 6. Section 7 draws conclusions. Section 8 identifies further avenues for research.

2 MODEL OF TIMED, ZERO-SUM GAMES

Timed, zero-sum games are realistic game models inspired by robot soccer. In this model, the primary objective is to find the optimal policy which maximizes the probability of winning. To obtain the policy, we utilize Markov decision processes (MDPs). We use the standard $(S, A, R, T, s_0)$ notation for representing MDPs. $S$ and $A$ are discrete sets of states and actions. $T$ is a stochastic state-action transition function. For each state $s$, the agent obtains a reward $R(s)$. The optimal policy $\pi$ can be exactly found by using a technique known as value iteration, which uses the Bellman equation (Puterman 1994) [7]:

$$V^{n+1}(s) = \max_{a \in A} \{ R(s) + \gamma \sum_{s' \in S} T(s,a,s') V^n(s') \},$$  \hspace{1cm} (1)$$

where $V^0(s) = R(s)$ and $\gamma = (0, 1]$ is a discount factor. For an infinite-horizon problem, $V^k$ converges to some $V^*$ as $k \to \infty$ (for

---

1 The University of Tokyo, Tokyo, Japan
2 National Institute of Informatics, Tokyo, Japan, email: {hmori, i-fishikawa, honiden} @nii.ac.jp
The optimal policy \( \pi^* \) for an infinite-horizon MDP is stationary (does not depend on time)[5]. For a finite-horizon problem with \( k \) time steps remaining, \( V^k \) enables us to find the optimal next action from any state. This optimal action may depend on the number of time steps remaining; such a policy is said to be nonstationary.

### 2.1 Thresholded rewards MDPs

To use more effective value iteration for timed, zero-sum games, we use thresholded rewards MDPs (trMDPs) [6]. Let a trMDP be a tuple \((M, f, h)\), where \( M \) is an MDP \((S, A, T, R, s_0)\), \( f \) is a threshold function, and \( h \) is an integer (the time horizon). Informally, \( M \) runs for \( h \) time steps while the agent collects cumulative intermediate rewards \( r_{\text{intermediate}} \); at the end, the agent receives a true reward \( r_{\text{true}} \) according to \( f(r_{\text{intermediate}}) \). A policy \( \pi \) for a TRMDP is nonstationary: it takes in a state \( s \in S \), the time remaining, and the intermediate reward achieved so far. Formally, the dynamics of a TRMDP are as follows:

**Algorithm 1** Dynamics of thresholded-rewards MDP.

\[
\begin{align*}
&\text{s} \leftarrow s_0 \\
&r_{\text{intermediate}} \leftarrow 0 \\
&\text{for } t = h \text{ to } 1 \text{ do} \\
&\quad a \leftarrow \pi(s, t, r_{\text{intermediate}}) \\
&\quad s \leftarrow s' \sim T(s, a) \\
&\quad r_{\text{intermediate}} \leftarrow r_{\text{intermediate}} + R(s) \\
&\quad r_{\text{true}} \leftarrow f(r_{\text{intermediate}}) \\
\end{align*}
\]

Our main focus in this paper is timed, zero-sum games. We consider the intermediate reward to be the difference between the agent’s score and its opponent’s score. We define the zero-sum reward threshold function \( f \) as:

\[
r_{\text{true}} = \begin{cases} 
1 & \text{if } r_{\text{intermediate}} > 0 \\
0 & \text{if } r_{\text{intermediate}} = 0 \\
-1 & \text{if } r_{\text{intermediate}} < 0.
\end{cases}
\]

This function assigns a true reward of 1 for a win, -1 for a loss, and 0 for a tie. Our objective is to find the optimal policy \( \pi^* \) that maximizes the expected value of \( r_{\text{true}} \).

### 2.2 Example

We present an example of timed zero-sum games — inspired by robot soccer — to make our problem clear and to support the description of our algorithm in the latter part of this paper. We model the robot soccer domain as a trMDP \( M \) with three states:

- For: our team scores a goal (reward +1)
- Against: our opponents score a goal (reward −1)
- None: no score (reward ±0)

Our agent is a team of robots, and at each time step, the agent adopts one of the strategies (actions). In this example, the agent can select one of three strategies — balanced, offensive, and defensive. In addition, there are two types of transition probabilities — actual and estimated. Actual transition probabilities are the actual model parameters of MDPs that directly control the whole game. Estimated transition probabilities are estimations the agent has in advance. These probabilities are listed in Table 1 and 2.

Of course, the agent knows that an offensive strategy seeks to score aggressively without care of giving up goals. Until the game starts, the only information that our agent knows is estimated transition probabilities, that are not much more than transition tendencies. In this example, while the agent considers the offensive strategy has a 25% chance of scoring, it actually has a 40% chance of scoring. Consequently, at the beginning of the game, the agent can only find the non-optimal policy calculated with estimated parameters. The actual transition probabilities are hidden parameters of the MDP’s model, and they will never be revealed during the whole game. Therefore, to find a near-optimal policy, the agent needs to learn the actual probability from a combination of prior estimated information and actual experience. The more the estimated transition probabilities are similar to actual probabilities, the more effective a policy can be obtained.

This example game can be modeled with trMDP \((M, f, h)\). The transition function \( T \) of \( M \) is defined with actual transition probabilities. The objective of trMDP is to find the policy that maximizes the expected value of \( f(r_{\text{intermediate}}) \) in \( M \).

### 3 EXACT SOLUTION ALGORITHMS

To find the optimal policy for trMDP by using the MDP solution technique, we have to convert a trMDP \((M, f, h)\) into an MDP \( M' \). If this conversion is designed such that finding the policy that maximizes the expected reward in \( M' \) is equivalent to finding the optimal policy of \( M \), we can use any MDP solution technique to find the optimal policy in timed zero-sum games. The algorithm of such conversions is shown in Algorithm 2.

**Algorithm 2** trMDP conversion into MDP

1: Given: MDP \( M = (S, A, T, R, s_0) \), threshold function \( f \) and time horizon \( h \)
2: \( s_0' \leftarrow (s_0, h, 0) \)
3: \( S' \leftarrow \{s_0'\} \)
4: for \( i \leftarrow h \) to 1 do
5: for all states \( s_1' \in (s_1, t, ir) \in S' \) such that \( t = i \) do
6: for all transitions \( T(s_1, a, s_2) \) in \( M \) do
7: \( s_2' \leftarrow (s_2, t - 1, ir + R(s_2)) \)
8: \( S' \leftarrow S' \cup \{s_2'\} \)
9: \( T'(s_1', a, s_2') \leftarrow T(s_1, a, s_2) \)
10: for all states \( s' = (s, t, ir) \) in \( M' \) do
11: if \( t = 0 \) then
12: \( R'(s') \leftarrow f(ir) \)
13: else
14: \( R'(s') \leftarrow 0 \)
15 return \( M' = (S', A, T', R', s_0') \)
Each state \( s' \) in \( M' \) is a tuple \((s, t, i, r)\), where \( s \) is the base state from \( M \), \( t \) is the number of time steps remaining, and \( i, r \) is cumulative intermediate rewards. After this conversion, we can find the optimal policy for \( M \) by applying any MDP solution techniques to \( M' \). However, general MDP solution techniques are infeasible in realistic environments, because their worst-case running time is \( O(|A|^h) \). Therefore, we use an efficient value iteration algorithm proposed by McMillen et al. that exploits the characteristics of trMDPs [6]. Algorithm 3 shows the details of the efficient value iteration algorithm.

**Algorithm 3 Efficient value iteration**

1: Given: MDP \( M' = (S', A, T', R', s_0') \), threshold function \( f \) and time horizon \( h \)
2: for all states \( s' = (s, 0, i, r) \in S' \) do
3: \( V'(s') \leftarrow f(s') \)
4: for \( i \leftarrow 1 \) to \( h \) do
5: for all states \( s'_i = (s_i, t, i, r) \in S' \) such that \( t = i \) do
6: \( \pi(s'_1) \leftarrow \arg \max_{a \in A} \sum_{s'_2 \in S'} T(s'_1, a, s'_2) V'(s'_2) \)
7: \( V'(s'_1) \leftarrow \sum_{s'_2 \in S'} T(s'_1, \pi(s'_1), s'_2) V'(s'_2) \)

In this algorithm, \( V'(s') \) is the expected thresholded value at state \( s \), and \( V'(s'_0) \) is the expected value obtained by the policy. This algorithm is definitely different from general algorithms in terms of computational complexity. When computing the value of each state, a general algorithm sums over all states reachable from the state. However with this effective value iteration algorithm, we do not need to sum over all reachable states when computing each value. The algorithm needs only to sum over its \( O(|S|) \) potential successors. Each strategy selection at a state requires maximization over each value of \(|A|\) actions, which is a sum of \(|S| \) possible successor states. Therefore, the worst-case running time of efficient value iteration for \( M' \) is \( O(|S'| |S| |A|) \). \(|S'| \) is decided upon trMDP conversion, and if we assume that intermediate rewards are drawn from a set of small integers, we can decrease the number of states of \( S' \) with mergers. The number of states in \( S' \) can be decreased to \( O(|S|h^2m) \), where \( m \) is the magnitude of the largest element of the intermediate reward. The total worst-case running time decreases to \( O(|S|h^2m) \). The algorithm is an incorrect expected reward of \(-0.11529\).

## 4 STOCHASTIC INFERENCE

Although trMDPs’ efficient value iteration algorithm runs exactly and rapidly, it doesn’t have adaptability against unknown opponents. In this non-adaptive algorithm, once the agent finds an incorrect policy calculated with incorrect parameters, it fully believes in the policy and the policy will never be improved. To ensure adaptability, the unknown transition probability needs to be modified through games. In this section, we introduce a Bayesian inference framework that is useful to guess the transition probability from the experience of action-state pairs. Bayesian inference is applied to many parameter estimation problems of reinforcement learning [9]. It is a means of statistical inference, which infers the probability that a hypothesis may be true by using evidence or observations [8]. Bayesian inference collects evidence that is meant to be consistent or inconsistent with a given hypothesis. Before the evidence is collected, it numerically estimates the degree of belief in the hypothesis and calculates the degree of belief in the hypothesis after evidence has been collected. Given new evidence, the probabilities are adjusted with Bayes’ theorem as follows:

\[
P(H_0|E) = \frac{P(E|H_0)P(H_0)}{P(E)}
\]

where
- \( H_0 \) represents a hypothesis inferred before the new evidence \( E \) became available,
- \( P(H_0) \) is the prior probability of \( H_0 \),
- \( P(E|H_0) \) is the conditional probability of seeing \( E \) given that hypothesis \( H_0 \) is true,
- \( P(E) \) is the marginal probability of \( E \): the probability of witnessing \( E \) under all mutually exclusive hypotheses, and
- \( P(H_0|E) \) is the posterior probability of \( H_0 \) given \( E \).

Generally, a multinomial estimation such as learning of transition probabilities by Bayesian inference makes use of the Dirichlet distribution as a prior distribution [11]. Dirichlet priors are consistent (the estimate converges with probability one to the true distribution), conjugate (the posterior distribution is also a Dirichlet distribution), and efficiently computable [2].

Let \( X \) be a random variable that can take \(|S| \) possible values from a set \( S \). Without loss of generality, let \( S = \{S_1, ..., S_{|S|}\} \). We are given a training set \( D \) that contains the outcomes of \( N \) independent draws \( e^1, ..., e^N \) from an unknown multinomial distribution \( P \). We denote by \( N_i \) the number of occurrences of the symbol \( i \) in the training data. A Dirichlet prior for \( X \) is specified by hyper-parameters \( a_1, ..., a_{|S|} \). According to the characteristics of Dirichlet priors, if the prior is a Dirichlet prior with hyper-parameters \( a_1, ..., a_{|S|} \), the posterior is also a Dirichlet prior with hyper-parameters \( a_1 + N_1, ..., a_{|S|} + N_{|S|} \). Therefore, we find that the initial prediction for each value of \( X^1 \) is

\[
P(X^1 = s_i | \xi) = \frac{a_i}{\sum_j a_j}
\]

and the prediction for \( X^{N+1} \) is

\[
(X^{N+1} = s_i | e^1, ..., e^N, \xi) = \frac{a_i + N_i}{\sum_j (a_j + N_j)}
\]

We can think of the hyper-parameter \( a_i \) as the number of ‘imaginary’ examples in which we saw the outcome \( i \). Thus, the ratio between hyper-parameters corresponds to our initial assessment of the relative probability of the corresponding outcomes. The total weight of the hyper-parameters represents our confidence in the prior knowledge. For our example problem, the ratio can be drawn from prior knowledge of the transition probabilities. The weight can be arbi-trarily chosen to control the learning behaviors. With a large weight, we can make the learning process stable. On the other hand, a small weight enables learning with fast convergence.

## 5 BAYESIAN ITERATION

With trMDPs’ efficient value iteration algorithm and the Bayesian inference framework, we are ready to design an adaptive algorithm for timed, zero-sum games with an unknown enemy. We propose an adaptive algorithm called Bayesian iteration. Bayesian iteration is
appropriately designed to deal with the incomplete information problem in timed zero-sum games. It utilizes Bayesian inference to learn actual transition probability and an efficient value iteration algorithm to modify the policy with low computational cost.

In this section, we describe the details of the Bayesian iteration algorithm. In advance of the games, the algorithm computes the policy for a given MDP with the estimated prior knowledge. In the following steps, it modifies the transition probabilities with Bayesian inference and recalculates the policy for the remaining steps with the modified information. A single modification does not need the policy to be recalculated over all states in the MDP. Only the states reachable from the current state need to be included in recalculation. Therefore, as the game proceeds and time steps pass, computational cost for a modification decreases.

The detailed explanation of Bayesian iteration is as follows:

<table>
<thead>
<tr>
<th>Algorithm 3 Bayesian iteration algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Given: MDP $M = (S, A, T, R, s_0)$, threshold function $f$</td>
</tr>
<tr>
<td>2: convert ($M, f, h$) to $M'((S', A, T', R', s'_0)$ (Algorithm 2)</td>
</tr>
<tr>
<td>3: current state $s_t \leftarrow s'_0$</td>
</tr>
<tr>
<td>4: for $t \leftarrow 1$ to $h$ do</td>
</tr>
<tr>
<td>5: $\bar{T}_t \leftarrow$ estimated with $\bar{T}$ and evidences</td>
</tr>
<tr>
<td>6: compute policy $\pi_t$ for $M_t$ with $\bar{T}_t$ (Algorithm 3)</td>
</tr>
<tr>
<td>7: $a \leftarrow \pi_t(s)$</td>
</tr>
<tr>
<td>8: $s_{t+1} \leftarrow s_t' \sim T'(s_t, a, s_t')$</td>
</tr>
</tbody>
</table>

where $t$ is the time steps that have passed since the start of the game, $\bar{T}$ is the initial transition probability initialized with equation (3), $\bar{T}_t$ is transition probability at time step $t$ as modified only with equations (4), $s_t$ is the agent’s current state with at time step $t$, $M_t$ is a subset of $M'$, whose states are reachable from $s_t$ and it is an MDP whose root state is $s$, $\pi_t$ is the policy for $M_t$ calculated according to the current $\bar{T}_t$. At each time step, the evidence should be stored with each state-action pair.

6 EXPERIMENTAL RESULTS

In this section, we show an application of our method to a timed zero-sum game. We experimented with over 120 time steps and the transition probabilities shown in Table 1 and Table 2. The weight of the hyper-parameter was varied from 0 to 300 to observe the properties and convergence of the expected reward of following the algorithm. Informally, the weight $w$ implies that the prior knowledge of each transition probability corresponding to an action-state pair is estimated with $w$ examples of each action. Figure 1 shows the result of the simulation. In this result, there are two lines: ’adaptive’ and ‘non-adaptive’. The adaptive line shows the outputs of our simulation. The non-adaptive line is an expected reward of -0.1153, which indicates the expected reward obtained if the agent believes in the estimated prior information. Note that the small hyper-parameter weight implies drastic learning behavior and a tendency of over-training. On the other hand, the learning process behaves stably when the hyper-parameters have a large weight. We can see the former behavior in the left part of Figure 1. With a very small weight, the expected reward becomes too low because of over-training. As the weight increases, over-training has less effect on the expected reward. The peak of the expected reward is -0.07097 at a weight of 115. At the peak, our algorithm exceeds the ‘non-adaptive’ reward by 0.0443. Moreover, the expected reward of our algorithm exceeds the ‘non-adaptive’ reward for almost all weights in Figure 1.

Note that the expected reward can be analytically computed. This experiment, however, shows the expected reward as the average reward over iterative simulations. We cannot compute the exact expected reward of large MDPs because of the computational complexity.

6.1 Heuristic techniques

The efficient solution algorithm presented above finds solutions to TRMDPs in $O(|A||S|^h h^3 m)$ time. The cubic dependence on the time horizon length will be an issue for problems in which the time horizon is long. The reason Bayesian iteration consumes much time is that the agent updates its policy at every time step.

The following facts about Bayesian iteration enable efficient learning:

**Fact 1** The agent can skip updating the policy at any time step.

Whenever the agent finds the optimal action for the current state, optimal actions for all the following states are necessarily found together. The agent can act without updating the policy according to an old policy.

**Fact 2** The quality of a policy updated at a state depends on the number of examples.

The number of past updates does not affect the quality of the policy. Skipping some updates does not result in a poor policy. In this section, we present two heuristic techniques that allow us to arrive at an approximate solution.

*interval-k heuristics.* With these heuristics, the agent considers updating its policy every $k$ time steps. The main idea of this technique is to reduce the frequency of policy updates. The time horizon is compressed by a factor of $k$. This change directly leads to a decrease in the state space in which the agent searches to find the policy. In this heuristic, the policy is updated uniformly between initial and later parts of the game.

*lazy-k heuristics.* With these heuristics, the agent does not update the policy until there are $k$ steps remaining. The computational complexity of an update depends on the number of remaining time steps. Ignoring the elapsed part of the game saves more time than to ignore non-elapsed part. The main idea of this technique is to concentrate computational effort near the end of run.

We tested the performance of these heuristic techniques, for a variety of parameter settings on the above example MDP. The weight
of hyper-parameters was 120, and time horizon was 120. The performance was in terms of the average reward of $10^6$ simulation runs. Figure 2 summarizes the results. Each point on the graph corresponds to a heuristic technique with certain parameter settings. The x-axis shows the number of states computed with that heuristic technique; the y-axis shows the average reward of that heuristic technique. Ideally, we would like a technique that provides a high average reward with a low number of states. Points in the upper-left area of the graph represent efficient trade-offs between state space size and expected reward.

![Figure 2. Performance of heuristic techniques on the example MDPs.](image)

The standard Bayesian iteration algorithm has an average reward of -0.073564 and requires 1,749,600 states to compute. The primitive ‘non-adaptive’ algorithm has an average reward of -0.011529 and requires 43,200 states. For small values of $k$, the interval heuristic closely approximates the standard solution while significantly reducing the size of the state space. Interval-5, labeled ‘A’, has an average reward of -0.075948 and requires 367,500 states. The selection of $k$ is a trade-off between solution time and quality; interval-24 (labeled ‘B’) uses only 95,040 states, but the reward drops to -0.089546. Lazy-80 (labeled ‘C’) has an average reward of -0.093774 and requires 564,840 states; this is a larger number of states than that of interval-5 and a lower average reward. In general, the interval heuristic has a higher reward than uniform for a given state space size.

However, there are some unexpected points; lazy-3 and lazy-10, labeled ‘D’ and ‘E’, have higher average rewards than that of the standard solution. These heuristic techniques ignored only a few steps at the beginning. We consider this unexpected result to be due to avoidance of over-training. At the beginning of the games, inferring the actual parameters tends to cause over-training, because of the shortage of experience. Therefore, avoidance of over-training by lazy-5 and lazy-10 may have resulted in a higher average reward.

7 CONCLUSIONS

In this paper, we presented an adaptive solution algorithm for tr-MDPs with incomplete information. Our algorithm consists of the trMDP framework and Bayesian inference. Our investigation enables trMDPs to deal with unknown enemy problems. With incomplete prior information, the agent can gradually learn the actual information and adaptively select its strategy. The agent discovers the hidden environment parameters online. We can also control the learning behavior by configuring the weight of the Bayesian inference, which represents the confidence degree of the prior information. For example, if we want the agent to learn actual parameters slowly and stably, we can put a large weight on the Bayesian inference.

However, there are two limitations to our algorithm. The first is that we must manually configure the weight of the Bayesian inference. However, there is no way to find the optimal weight that enables the agent to obtain the optimal expected reward. Automation to find the optimal weight is a difficult problem. The second limitation is that the expected reward obtained with Bayesian iteration is much lower than that of the optimal policy found with complete information. In the example experiment, the agent achieved -0.070979 expected reward. It exceeded the ‘non-adaptive’ solution by 0.0443. However, with complete information, the ‘non-adaptive’ solution is ‘0.04695’, which exceeds the adaptive solution value by 0.1179. This gap clearly shows the qualitative limitations of our algorithm.

We also investigated two heuristic techniques that reduce computational complexity. The interval-$k$ heuristic, which compresses the time horizon uniformly, has high performance with a small state space. In this heuristic, the agent can, with few computations, perform close to the standard Bayesian iteration solution. Note that the lazy-5 and lazy-10 heuristic techniques outperform the standard solution algorithm, by ignoring some of the starting steps. This indicates an interesting characteristic of the algorithm wherein inference near the beginning of the game tends to cause over-training.

Through our investigation, our agents became able to decide their policy adaptively in correspondence with their experience. This adaptiveness also enables decisions to be made in a short runtime. However, we haven’t assumed that our opponents behave in the same way as our agents. If opponents change their policies according to certain patterns, we should recognize the opponents’ patterns in order to confront them adaptively. Our future work will address such an opponent. Our method also has a problem with local optimizations. If the agent finds a suitable action in the early steps, it tends to continue selecting that action. This tendency leaves the probabilities of other actions less learned. To achieve higher rewards, we should make the algorithm examine less-learned actions.

REFERENCES

A Heuristic for Scrabble based in Probability

Alejandro González Romero¹, Francisco González Acuña²,
Arturo Ramírez Flores², Amador Roldán Aguilar³, Rene Alquézar¹ and Enric Hernández¹

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
C/Jordi Girona Salgado, 1-3, Mòdul C6
08034 Barcelona- Spain.
aromero@lsi.upc.edu, yarnalito@gmail.com

Abstract

The game of Scrabble, in its competitive form (one vs. one), has been tackled mostly by using Monte Carlo simulation. Recently [1] Probability Theory (Bayes theorem) was used to gain knowledge about the opponents tiles; this proved to be a good approach to improve even more Computer Scrabble. We used probability to evaluate Scrabble leaves (rack residues); then using this evaluation, a heuristic function that dictates a move can be constructed. To calculate these probabilities it is necessary to have a lexicon, in our case a Spanish lexicon. To make proper investigations in the domain of Scrabble it is important to have the same lexicon as the one used by humans in official tournaments. We did a huge amount of work to build this free lexicon; then to test it, Amitabh’s free source Scrabble program [2] was adequately modified, to be able to play Scrabble in Spanish using the free lexicon built according to the official rules of Spanish Scrabble tournaments (the FISE rules). In this paper a heuristic function that involves leaves probabilities is given. Although we still have not finished an engine that uses this heuristic, we were able to perform some experiments to test this heuristic. The tests are matches against highly expert players; the games played so far give us promising results. Another plan to build a stronger engine that combines heuristics using probabilities, opponent modeling and Monte Carlo simulation is also proposed.

1. Introduction

Scrabble is a popular board game played by millions of people around the world. In this article we consider its tournament variant, which is a two player game. Competitors make plays by forming words on a 15 x 15 grid, abiding by constraints similar to those found in crossword puzzles. Scrabble, in contrast with other games like chess, Go, draughts, etc. is a game of imperfect information. Techniques of Artificial Intelligence have been applied to games of imperfect information, mainly card games such as Bridge and Poker [3,4]. The theoretical support for the games of strategy goes back to the work of Von Neumann and Morgenstern [5].

As to the generation of valid moves, Apple and Jacobson [6] introduced an algorithm, which proved to be the fastest and more efficient at one time. It is based in the data structure DAWG (directed acyclic word graph) derived from the entries of a reference lexicon. More recently, Steve Gordon [7] has introduced a variant of this data structure (GADDAG) which occupies 5 times more space than DAWG, but it duplicates the speed of move generation.

Once we have a move generator, the implementation of basic engines simply based in the move that maximizes the score in each turn, is simple. Shapiro [8] and Stuart [9] are examples of precursors in the development of engines for Scrabble. To solve the obvious deficiencies of this greedy approach, simulation techniques were applied. These techniques took into account the possible replies to a candidate move, the replies to those replies, and so on for many plies. This method, known as simulation, rests on Monte Carlo sampling for the generation of the opponent’s rack, in situations with uncertainty, and it has its theoretical base in the Minimax technique of Game Theory. The program MAVEN [10] developed by Brian Sheppard is one of the references for this paradigm and its excellent results against top-level players are the best demonstration of the appropriateness of this approach. Recently Jason Katz-Brown and John O’Laughlin [11] have implemented QUACKLE, a program distributed under the GNU license, which also exploits the simulation technique.
Although simulation is an excellent mechanism to quantify how good plays are, it requires some complements to improve its efficiency. For example, the generation of the opponent's racks in MAVEN did not take into account that expert players try to keep a good leave (rack residue), which suggests that the Monte Carlo sampling should be biased to model this characteristic. One can also model the strategic component so as to reduce uncertainty, inferring information from the opponent's racks. For this purpose, the probabilistic models are especially adequate. As an example of this line of research, we can cite the work of Richards and Amir [1].

A difference between Heuri and other programs is that the evaluation of a leave is calculated by multiplying an expected value by a probability rather than using simulation to calculate the values of individual tiles, summing them, and finally adjusting for synergy and other factors [12]. Even though Scrabble is already dominated by computer agents [12] there is still room for improvement in Computer Scrabble [1]. The present article uses probability models and techniques to select the best move. We believe that a combination of simulation with probability methods might improve even more Computer Scrabble.

The game of Scrabble is a good platform for testing Artificial Intelligence techniques. The game has an active competitive tournament scene, with national and international scrabble associations and an annual world championship for Spanish-language players.

2. Building the Scrabble tools

2.1 The Lexicon

An important part to conduct research in Scrabble and to build a program that plays Scrabble, in Spanish for our case, is the construction of a lexicon. The length of it is significantly larger than the length of a corresponding lexicon in English.

A regular transitive verb in Spanish, as “amar”, “temer” or “partir” has 46, 55 or 54 verbal inflections depending on whether it is an ar-verb, an er-verb or an ir-verb, whereas a verb in English, like “love”, typically has only four verbal inflections (love, loves, loved, loving). Even though short words are more numerous in English than in Spanish -1350 words of length less than 4 in SOWPODS, the UK scrabble dictionary, and 609 in Spanish- the number of words of length less than 9 (the important lengths are 8 and 7) is roughly 107000 in SOWPODS and 177000 in our Spanish lexicon. Accordingly, more bingos (seven tiles plays) are expected in a Spanish game than in an English game. The total number of words (of length less than 16) is, approximately, 246000 in SOWPODS and 635000 in our Spanish lexicon.

The electronic version of DRAE (Diccionario de la Real Academia Española), the official Scrabble dictionary in Spanish has approximately 83000 lemmas (entries) from which we had to construct the complete lexicon. Let us mention that petitions to RAE (La Real Academia Española) for a text file containing the collection of all conjugated verbs, or even just all verbs in infinitive form, were unsuccessful. Petitions for nouns and adjectives were also unsuccessful.

We therefore have the following two subproblems:

1. Conjugation of verbs. We constructed, in an automated way, the irregular inflections of irregular verbs; similar constructions were done for regular verbs and regular inflections. Some effort was required to express all conjugations with as few collections of endings as possible. For the purposes of Scrabble we group the irregular verbs in 38 families. The last family consists of the verbs “caber, caler, desosar, erguir, errar, estar, haber, ir, jugar, oler, poder and ser”; no verb in this family is conjugated like any other verb in Spanish. We give a model for each of the remaining 37 families. Each verb in a family is conjugated like the model belonging to it. For space reasons we do not give the list with the irregular inflections of each model and also the additional inflections which come from “voseo” (Spanish of some regions including Argentina). We use the following abbreviations of collections of endings:

\[ E_1 = \{o, a, as, amos, ais, an\}, \quad E_2 = \{a, e, o, as, es, an, en\}, \quad E_3 = \{e, es, en\}, \quad E_4 = \{ieron, iera, ieras, ieramos, ierais, iern, ieres, iere, ieres, ieremos, iereis, ieren, iese, ieses, ieseis, iesen\}, \quad E_4' = \{E_4, iendo, io, amos, ais\}, \quad E_4'' = \{E_4, e, o, iste, imos, isteis\}, \quad E_5 = \{nero, nera, neros, eras, eran, ere, eres, eremos, ereis, ese, eses, esosos, eseis, esei, en\}, \quad E_5' = \{E_5, eno, o\}, \quad E_5'' = \{E_5, e, o, iste, imos, isteis\}, \quad E_6 = \{a, as, an, ia, ias, iamos, iais, ian\}, \quad E_6' = \{E_6, e, emos, eis\}, \quad E_6'' = \{E_6, o, amos, ais\}. \]

We only give 4 models out of the 37 models, their irregular inflections, the inflections coming from voseo and the number of verbs in each family.

1. agradecer agradezc(E1) 260
2. acertar aciert(E2) acert(a, as) 157
3. contar cuent(E2) cont(a, as) 126
4. construir construy(E1, E3, E5’) 55

2. Plurals of nouns and adjectives. Plurals of articles are explicit in DRAE and pronouns which admit plurals are few (roughly 20). Adverbs, prepositions, conjunctions,
interjections, onomatopoeias and contractions do not have plurals. If a word is not an entry but is contained in a phrase having the abreviation expr. (expression) or loc. ("locución") it cannot be pluralized. However, an adjective contained in an “envío” having a little square and the abreviation V.("véase") can be pluralized, for example “gamada”.

The FISE (Federacion Internacional de Scrabble en Español) rules are clear as to which inflections of verbs are valid since old verbs cannot be conjugated (old words are those which in all its meanings have one of the abbreviations ant., desus. or germ.) and all verbs which are not old are explicitly conjugated in DRAE. Unfortunately all these conjugations cannot be copied in a text file as they are encrypted in a special format. Therefore we had to develop a way to conjugate verbs. The FISE rules indicating which nouns and adjectives admit a plural are less clear.

2.2 A Spanish Scrabble program

After a lot of work the huge task of building the lexicon was accomplished. Then we modified a free English Scrabble source program developed by Amitabh [2]. Finally after all this work we had (Scrabler II) a Spanish Scrabble program that plays with the official rules of Spanish Scrabble tournaments (the FISE rules).

In order to check the performance of the new lexicon built and the functional capabilities of the Scrabler II program a match was performed between Scrabler II and an expert human Scrabble player who is ranked No. 33 in the Spanish Scrabble League. The match followed the format of a baseball world series match, that is the player who arrives at 4 victories first wins the match. The final result was 4-2 in favour of the human expert.

As can be seen, although at first sight one could think that the computer has a huge advantage against a human because it knows all the permissible words, it turns out that this is not sufficient to beat a Scrabble expert. The reason is that Scrabbler II employs a naive, greedy strategy. It always plays a move that maximizes the points at the current time, ignoring how this could affect the options for making moves in the future. Many times you will end up with low quality racks like {GGLNQPM} and since there is no change strategy these low quality racks will tend to prevail giving the human lots of advantage. For instance, in Scrabble one usually tries to play a bingo, (playing all seven tiles in one turn), since this gives you 50 bonus points. Therefore it is convenient to preserve the blank tiles which are jokers that represent any chosen permissible letter. Since Scrabbler II plays always the highest point move, it tends to get rid of the blank tiles without making any bingo.

Since all these features were observed we decided to make an improved version of Scrabbler II with an inference strategy engine. This engine will follow a heuristic function that tries to play a high scoring move as well as balancing the rack in play.

3. Research work (Scrabble Strategy)

3.1 A Scrabble one side representation

Viewed from a one-player perspective, a game of Scrabble can be represented as a path on a certain tree. This tree could consist of nodes, that represent board positions, which will, almost always, have 3 types of children (there can be several children of one type). The root node will denote the initial (empty) board position. One child could represent a board position (bp) reached after changing tiles or passing, the other child could represent the bp after playing less than 7 tiles, and the other child the bp after playing all 7 tiles (See Figure 1). Each edge in the tree could represent a certain move: pass or change, play less than 7 tiles, bingo (playing all 7 tiles). Let us call a “bingo edge” an edge that corresponds to a bingo move.

A cost can be assigned to each edge according to the amount of points achieved by the move. For instance a change or a pass will give us zero points, while a bingo would give us 50 points plus the points for the word in the board.

An intuitive idea to maximize the amount of points in a path (a one-sided view of a game of Scrabble) is to visit as many bingo edges as possible.

To maximize the probability of visiting bingo edges, we evaluate each possible leave of the rack “in play”. This evaluation is performed by using: all 7-letter words in the lexicon, the letters left inside the bag and the opponents’ rack. All this is used to calculate, for every possible leave of the rack (rack leave), the probabilities of getting a 7-letter bingo in the next move. This strategy is likely to perform better in Spanish than in English, since expert players in Spanish average three or more bingos per game, while English expert players average two or more bingos per game. This is logical since...
the number of 7 and 8 letter words in Spanish is greater than the corresponding numbers in English. There are approximately 132,500 words of either 7 or 8 letters in Spanish versus approximately 68,000 corresponding words in English (SOWPODS).

3.2 Probability and heuristic

An important part of a program that plays scrabble in Spanish is the decision of what to leave in the rack. We propose to give a numerical evaluation as follows:

\[ v = j + p*b - d \]

where

- \( j \) is the number of points made by the move, in which \( t \) tiles are played \(^1\); \( j = 0 \) if the \( t \) tiles are changed rather than played on the board;
- \( p \) is the probability of obtaining a 7-letter bingo if \( t \) tiles are drawn at random from a set that is the union of the bag and the opponent’s rack (which we call the "augmented bag");
- \( b \), the expected value of a bingo, is 77 (an average taken from 1000 games) or better:

\[ b = 50 + 2.5(r + 1.92t) \]

where \( t \) is the number of tiles drawn from the bag and \( r \) is the total number of points of the leave; \( d \) is a nonnegative number which is zero if the move is not weak from a defensive point of view. It is 20, say, if one puts a tile on the edge allowing a possible nine-timer; it is 10 if one plays a vowel next to a premium square allowing a six-timer; it is 5 if one plays allowing a possible four-timer.

The equality for \( b \) was obtained as follows. First we wrote \( b = 50 + k*e \) where \( e \) is the expected total points of (without premiums and bonus) of a bingo. The sum of the values of the 100 tiles is 192 and so the average value of a tile is 1.92 and \( e = r + 1.92t \). As an example, if the leave is \{ZADO\}, with \( r = 14 \), and we take \( t = 3 \) tiles from the bag we would expect these tiles to add, on the average, \( t * 1.92 \) (=3*1.92). But most of the bag have premiums (very often, besides the 50 points bonus, the word score is multiplied by 2 and sometimes 3, 4 or 9). Hence it is reasonable to multiply \( e \) by a constant \( k \) and we took \( k = 2.5 \) because very often it gives for \( b \) a value close to the experimental bingo average 77. For example, in the frequent case \( r = 3, t = 4 \) one gets \( b = 50 + 2.5(3 + 1.92*4) = 76.7 \).

It is better to explain the calculation of \( p \) using an example: What is the probability \( p \) of obtaining the "septet" \( s = \{AAANÑRR\} \) (from which one can form the 7-letter bingo ARĂNARA) if one has the leave \{AAAN\}, one changes \{HQV\}, the augmented bag is the initial one (that is, equals "total bag – {AAAÑHQV}") and one draws 3 tiles from it?

Answer: If \{AAAN\} were not contained in \{AAAÑRR\} \( p \) would be 0. However \{AAAN\} is contained in \{AAAÑRR\} so we consider the difference set \{AAAÑRR\} - {AAAN} = {ARR} = {AR} and the augmented bag:

\{AAAAAAAAARRBBCCCC...XYZ\} = \{A\}^9 {R}^2 {B}^2 {C}^4 ... {XYZ}\}

and one then computes \( p = C(9,1)*C(3,2)/C(93,3) \) (93 is the number of tiles of the augmented bag and the 3 in \( C(93,3) \) is the number of tiles that are taken from the bag) where \( C(m, n) \) is the binomial coefficient:

\( C(m, n) = m(m-1)(m-2)...(m-n+1)/n! \)

The numerator has \( n \) factors and \( C(m, n) \) is the number of \( n \)-element subsets of a set consisting of \( m \) elements. Notice that the denominator \( C(93,3) \) does not depend on the septet \( s \).

The probability \( p \) of obtaining a 7-letter bingo if one has the leave \{AAAN\} and the augmented bag: "total bag – \{AAANHQV\}" is then the sum of all \( p \) when \( s \) runs over all septets.

3.3 An alternative improved strategy or plan

To get a better Scrabble engine, we could combine several approaches in the following way:

First of all we need to define the concepts of “open board” and “closed board”.

Let us call an open board a board position in which there is at least one “bingo line”. A bingo line is a place on the board where a bingo can be inserted legally.

Let a closed position be a board position in which there are no bingo lines.

Then a possible strategy or plan to follow by a Scrabble engine would be:

Determine whether the board is open or closed and then:
1. If the board is open use the heuristic that employs probabilities.
2. If the board is closed use Monte Carlo simulation.

In both cases use opponents’ modeling.

Another possible aspect in Scrabble strategy is to take into account the current score of the game. For instance, if a player is well behind in the score, a possible action would be to make “come back moves”, this type of moves would risk opening bingo lines, for example, to catch up. On the
other hand, if a player has a good lead, he might try to hold
the lead by making “blocking moves”. A blocking move
could be any move that reduces the number of bingo lines,
or covers triple premium squares.

4. Results, conclusions and future work

Due to the unfinished engine we could only perform a few
games to test our heuristic, let this heuristic be called
Heuri. After playing 17 games against several top class
players Heuri results are:

Heuri 10 wins and 7 loses, average per game: 509 pts.

Matches against humans are being performed by internet,
this gives the humans an extra force since they can consult
the words before they actually put them on the board, they
also have a lot of time to play. The matches follow the
format of a baseball world series, the first opponent
arriving at 4 victories wins the match.

In its first match against a human Heuri won 4-2 against
the best player of Colombia, Fernando Manriquez with an
ELO 2110 (the highest ELO in the FISE list is 2191),
ranked No.20 in the FISE list of Spanish Scrabble players.
The other matches are still in progress. Another important
victory of Heuri was against Benjamin Olaizola from
Venezuela who is the current world champion!, his ELO is
2169, he occupies the 2nd place in the international ranking
list. It was a very close game! (Heuri 471 Benjamin
Olaizola 465). Heuri also played a match against an
engine who always plays the highest move, Heuri won 4-0.

The results indicate that indeed Heuri is a good Heuristic
and is probably already stronger than humans, but to
confirm this we should finish the engine to perform many
more matches. It is also convenient to hold matches against
Quackle’s computer agent has the same basic architecture
as Maven.

In the future the results could also help to discover new
strategies that could help human players become the best
Spanish scrabble players in the world, and we would also
have a world competitive Spanish Scrabble program to
challenge any Spanish Scrabble program in a match,
perhaps this match could be held in the computer
Olympiads that are held every year by the ICGA
(International Computer Game Association). The technique
employed can also serve to build Scrabble engines for
other languages like English, French and Catalan.
Finally, in order to get a much stronger engine we could
divide the game into three phases as Maven does. Maven’s
architecture is outlined in [12]. The program divides the
game into three phases: the endgame, the pre-endgame,
and the middlegame. The endgame starts when the last tile
is drawn from the bag. Maven uses B*-search [13] to
tackle this phase and is supposedly nearly optimal. Little
information is available about the tactics used in the pre-
endgame phase, but the goal of that module is to achieve a
favorable endgame situation. All the work presented in
this paper could be used in the middlegame and perhaps in
the pre-endgame too, although the pre-endgame might require
careful study to be improved. Since the endgame phase is
supposedly nearly optimal using B*-search [13], we could
follow Maven steps [12] to tackle this phase of the game.

References

[1] Richards, M., Amir, E., Opponent modeling in
Scrabble, Proceedings of the Twentieth International
Joint Conference on Artificial Intelligence, p.
[2] Amitabh S., Scrabblet award winning Scrabble
program built as part of a programming competition at
IIT Delhi, 1999.
bridge-playing program. In Proceedings of the
Sixteenth International Joint Conference on Artificial
[4] Darse Billings, Aaron Davidson, Jonathan Scafeher,
and Duane Szafron, The challenge of poker, Artificial
Games and Economic behavior. Princeton
University Press, 1953.
Generation Algorithm. Software—Practice and
[8] Shapiro, S.C. A scrabble crossword game playing
program. Proceedings of the Sixth IJCAI, p. 797-799,
1979.
[9] Stuart, S.C. Scrabble crossword game playing
programs. SIGArt Newsletter, 80 p.109-110, April
1982.
[10] Brian Sheppard, World-championship-caliber
is an open source crossword game program released in
March 2006.
PhD thesis, IKA/Computer Science Department,
Universiteit Maastricht, July 2002.
Framework for Evaluating Believability of Non-player Characters in Games

Tero Hinkkanen1  Jaakko Kurhila2  Tomi A. Pasanen1

Abstract. We present a framework for evaluating believability of characters in first-person shooter (FPS) games and look into the development of non-player character’s user-perceived believability. The used framework is composed of two aspects: firstly, character movement and animation, secondly, behavior. Examination of three different FPS games yields that the newer the game was, the better the believability of characters in the game. Moreover, the results from both the aspects of the framework were mutually balanced through all games examined.

1 INTRODUCTION

First-person shooter (FPS) games have been popular ever since their first release in the early 1990’s (Hovertank 3D 1991, Wolfenstein 1992, and Doom 1993). The games are usually straightforward in a sense that the target is to navigate the player’s character through different levels of the game and accomplish different tasks. Normal task is to move from point A to point B and shoot everything that moves or tries to shoot back. The view to the game world consists of a split screen where the narrow lower part of the screen is showing player’s health and ammo, and the upper large part of screen represents player’s eye view to the game world (the large part is the player’s only mean to monitor other characters in the game and draw conclusions about them). Depending on the game, in-game characters run by the player or by the computer can have human-like constraints or not. Because games are made for players’ enjoyment, not every part of real-world laws and rules are included in the games. For example, a player must be able to win even the most superior enemies in the game alone [8].

Main reason for the popularity of FPS games has been their relatively high-level graphics together with a breathtaking pace of interaction. Nowadays, however, players have started to except even more realism in games, such as unpredictability. Because of the games in this genre, many significant improvements on the game activities have been attached to characters run by the computer, in other words “non-player characters” (NPCs) which the player’s character will meet during the game.

It can be said that the ultimate goal of an NPC is to be indistinguishable from a character run by a player. However, recent studies show [2, 14] that players will notice if their opponent is controlled by a computer rather than another human player, or if the opponent is too strong or too weak compared with another human player. According to the studies, the elements increasing the believability of NPCs as human players are natural movement, mistakes and gestures during the game, character appearance and character movement animation.

An NPC can be seen as an intelligent agent trying to do its best (rational action) for an NPC, artificial intelligence (AI) tries at the same time to be as entertaining as possible, using even “cheap tricks” [16, 24]. Cheap tricks are permitted by the players as long as the player stays relatively convinced of the rationality of the actions.

In this paper, we take a look at how the game industry has been promoting non-player character’s (NPC) believability in FPS games and compile a framework for evaluating believability of the NPCs. We start by looking at the elements which build believability in the next section and present the framework in Section 3. In Section 4, we apply our framework to three FPS games revealing improvements along the age of games. We conclude with final remarks in the last section, Section 5.

We note that other authors have also collected a number of techniques or ideas to promote NPCs’ believability but proposed criteria have been either very universal [14], not specific to FPS, or criteria have been too loose giving maximum scores to any NPC in FPS games like Quake or Unreal Tournament [2].

2 BUILDING BELIEVABILITY

Based on different studies among players [2, 14], the believability of the NPCs is most influenced by (1) the game environment where the NPCs appear, (2) another character or player which the NPC is compared to, and (3) the players’ cultural background and age. Because we are aiming to a general framework we skip the last item and divide NPC’s believability into three main categories: movement, animation and behavior. Next we consider how game developers have tackled each of these.

2.1 Movement

In FPS games, NPCs’ movement usually tries to emulate humans’ natural movement: finding the obvious shortest path and reacting to the game environment. One typical way of helping an NPC to find the shortest path is to build a navigation mesh onto the game map. Game designers plant varying number of navigation points or nodes onto the map. When the NPC searches for the shortest way to its destination, it actually calls for the search algorithm of the game to find the shortest path between the two navigation points: the one where the NPC is, and the one where it’s going to go.

The most commonly used search algorithm is A* [16] and its variations. In some FPS games, designers have eased NPCs’ pathfinding algorithms by outlining the area(s) where NPCs can move. This reduces the search space significantly and thus quickens the search. However, if an NPCs’ destination is not within its range of movement or otherwise out of its reach, A* has to search trough every node in the search space, which particularly in large maps demands a great amount of CPU time. In case that game designers have not considered this option, the NPC calls for the pathfinding algorithm over and over again thus slowing the game down. In these cases, NPCs have been killed off so that the CPU time will not be wasted [9].

1 Gamics Laboratory, Department of Computer Science, University of Helsinki, Finland, email: firstname.surname@cs.helsinki.fi
2 Department of Computer Science, University of Helsinki, Finland, email: firstname.surname@cs.helsinki.fi
Even though the performance of computers has been rising continuously, optimizing algorithms are still needed to guarantee the smooth running of ever bigger games [25]. Reducing the navigation mesh is a simple and fast way to increase the speed of A*’s, but it leads to a sparse search space and thus to a clumsy and angular NPCs’ movement.

A good solution to optimize the searches is to cut down their number. This can be reached in two different ways: one is to reuse old searches for other NPCs and the other is to limit the flooding of A*. Flooding is understood as the extensive widening of pathfinding around the optimal path.

Even if A* can not find a path to a location, it is not wise to erase this search result from the computer’s memory. In games where there are several NPCs, it is likely that some other NPC is going to search for a similar or even the same route at some point of the game. Now if the failed results are already in the memory, it saves a lot of the CPU time when there is no need to do the same search again [4]. Keeping a few extra paths in the memory does not notably limit the amount of free memory during the game.

By using path lookup tables, it is possible not to use any pathfinding algorithms during the game at all [2]. Every possible path will be stored in the lookup table, which is loaded in the memory while the game begins. Even though the tables will be quite large, it is still faster to look for a path straight from the table rather than search for the best route to the location. Major problems with the path lookup tables are that they require completely static maps and a lot of free memory.

Humans tend to move smoothly, that is, they attempt to foresee the upcoming turns and prevent too sharp turning. NPCs’ paths can be smoothed in several ways. One is to use weighted nodes, in the upcoming turns and prevent too sharp turning. NPCs’ paths can be fitted to a drawn character.

The third way is to connect sensors to a person and record the person’s different moves onto the computer. Then these moves are included, for example, running animation for the lower part of the body and shooting animation for the upper part while face movements for yelling are shown in the NPC’s face.

Animations are even used to hide programming bugs in the games. In Half-Life when a player throws a grenade amongst a group of enemy NPCs, NPCs’ pathfinding does not always find paths for NPCs to run away from the immediate explosion. Programmers at Valve Software could not localize this bug but they could see when the bug occurred [13]. They programmed NPCs to duck and cover every time this bug appeared, and this solution was warmly welcomed by players saying it added an extra touch of human behavior to NPCs.

### 2.2 Animation

Most of the animations used by NPCs are made with one of the following three methods [2]. One is to draw or otherwise gain the frames of the entire animation and then combine them into one sequence. The other method is to draw a couple of keyframes and later on morph them with computers into one smooth animation. The third way is to connect sensors to a person and record the person’s different moves onto the computer. Then these moves are fitted to a drawn character.

Each one of these methods has the same flaw: Once an animation is done, it can only be changed by recording it again. This obviously can not be done during the game. By recording several different animations for NPCs’ one action, it is possible to change between different animations if the same action occurs again and again. This, however, only prolongs the obvious, which is that player will notice if dozens, or even a few, of NPCs limp or die with in precisely the same way.

Using hierarchically articulated bodies or skeleton models, NPCs’ animations can be adjusted to fit different situations and actions [2]. The skeleton models can also be fitted to different NPCs only by changing the model’s appearance and size. The use of the skeleton models reduces the amount of memory needed for animations, because every animation is now done when needed instead of using pre-recorded sequences.

NPCs’ appearance is very important while their believability is looked into. If gaps occur between the limbs and the torso, or other oddities can be seen in NPCs’ appearance, it decreases their believability. While the polygon mesh is added over the skeleton these flaws can be avoided by paying attention to how the mesh is connected to the skeleton and by adding padding between the skeleton and the mesh [2].

The animation controller (AC) has an important role considering the NPC’s animations. The AC decides what animation is played with each action and at what speed. In case of two animations are played sequentially, the AC decides at what point the switch happens. Some animations have a higher priority than others. Showing the death animation overrides every other animation, because it is the last animation any NPC will ever do.

If NPCs are made with skeleton models, the AC needs to decide which bones are to be moved and how much in order to gain believable movement for an NPC. Some animations or movements can be shown simultaneously with other movements. These include, for example, running animation for the lower part of the body and shooting animation for the upper part while face movements for yelling are shown in the NPC’s face.

### 2.3 Behavior

Making mistakes is human, therefore it is not to be expected that any NPC’s actions are flawless. Intentional mistakes, such as two NPCs talking loudly to each other or an NPC’s noisy loading of guns, reveal the NPC’s location to a player before he/she can even see it. NPCs’ far too accurate shooting tends to frustrate the players so it is recommended that the first time an NPC sees the player’s character, it should miss it thus giving the player time to react and shoot back [5, 13].

NPCs need reaction time for different actions to be able to imitate the physical properties of human [5, 14, 20]. These are made by adding one second delay for each action NPCs have, thus making them appear as if they were controlled by other players.

Both predictability and unpredictability are natural for human players [22]. In FPS games, this becomes apparent when either too weak or too powerful weapons are chosen in the game. Emergent behavior (EB) offers more unpredictability for NPCs. In EB, no simple reason can be given for the NPC’s known actions and therefore the result of the action can benefit either the NPC or the player’s character.
Emergent behavior occurs mostly when timers and goal-based decisions are used to control NPCs’ behavior [19, 22]. Emergent behavior can also be a result from several small rules that NPCs follow. A good example of this is flocking [3, 10, 19]. In flocking, every member of a flock or a group follows exactly the same rules, which can produce more action than the sum of these rules dictates.

Moreover, NPCs should take notice of other NPCs and their actions, and be aware of their existence. If game programmers so desire, NPCs can give support to each other or search for cover together [13]. In the worst case, a guard can walk over his fellow guard without even noticing his dead corpse on the ground [14]. However, it has been stated that the most important thing for NPCs to notice is to avoid friendly fire [20, 26].

NPCs can “cheat” by using information they possibly could not obtain in real life. These include locations of ammo and health in the game, the location of the players’ characters’ or even the characters’ health and fighting capabilities [13, 22]. This information can be programmed for the player’s benefit, too. By letting the player’s character’s health to drop to near zero and then by changing the NPCs from ultimate killing machines to sitting ducks, the game can give the player a feeling of a sudden success and thus keep him/her playing the game longer.

Lately, cheating of NPCs has been reduced by game programmers in order to give the player and the NPCs equal chances to survive in the game. At the same time, NPCs’ abilities to autonomously search for health and ammo through different game levels and remember where it has or has not been have increased. Thus the change has been from cheating to more human-like NPCs [6, 12].

NPCs’ behavior is mostly controlled by finite state machines [2, 3, 7, 28]. In addition to state machines, triggersystems and scripts are used in state transitions. A more developed version of the state machine is a hierarchical state machine, in which every state is divided into smaller state machines which have their own states and state transitions [2].

3 DESCRIPTION OF FRAMEWORK

A framework for evaluating the believability of characters is a means to evaluate user-perceived NPC believability in FPS games. It should be noted that this framework is intentionally limited to provide simplicity and universality in use.

The framework is composed of two main aspects: firstly movement and animation, secondly behavior. It is based on programming techniques and algorithms used in different FPS games. This framework does not take a stance on how some requirement has been executed, but only whether or not it has been implemented so that the player can perceive it.

The basic element of NPCs’ movements and animations is that any NPC can find the most suitable path to its destination. In most cases, NPCs’ destination is the current location of the player’s character. NPCs’ path may not be the shortest, but it must be a reasonable suitable path. Because game maps are divided into smaller blocks to prevent too large search spaces, an NPC has to be able to cross these borders especially after it has noticed the player’s character.

When NPCs move, they must move smoothly and be capable of avoid running into both static and dynamic obstacles. The player will not be convinced of NPCs’ believability if it cannot move around a barrel or wait for a moving vehicle to move out of its way. When two or more NPCs move together, they must pay attention to each other to avoid collisions.

When observing NPCs’ animations, three different things are of importance. First, one should note whether there are several pre-recorded animations for one action or not. Secondly, a shift from one pre-recorded animation to another must be fluent so that no unrealistic movements are made in between. Third, NPCs’ appearance must be done well enough so that no gaps can be seen between their limbs or other unnatural design is apparent.

Tables 1 and 2 show the specific propositions that are used in evaluating the believability of NPC characters. Propositions equal points, and the points are added into a score. Some propositions are viewed to have a greater impact on the believability. Therefore, some rows in Tables 1 and 2 are counted for doubling the score, i.e. the points for a single requirement can be 2 instead of 1. The importance of some requirements over others is based on the view taken in this study.

### Table 1. Scores for movement and animation

<table>
<thead>
<tr>
<th>Requirement for NPC</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPC can find the most suitable path for its destination.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s movement is not limited to a certain area, such as one room.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s movement is not clumsy or angular.</td>
<td>2</td>
</tr>
<tr>
<td>NPCs are aware of each other and do not collide with each other.</td>
<td>1</td>
</tr>
<tr>
<td>NPC can avoid any dynamic or static obstacle in game field.</td>
<td>2</td>
</tr>
<tr>
<td>NPC has different animations for one action.</td>
<td>1</td>
</tr>
<tr>
<td>Shifting from one animation to another is fluent.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s appearance is done carefully and no unnatural features can be found in it.</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>

### Table 2. Scores for NPC’s behavior

<table>
<thead>
<tr>
<th>Requirement for NPC</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPC makes intentional mistakes.</td>
<td>2</td>
</tr>
<tr>
<td>NPC has human-like reaction times.</td>
<td>2</td>
</tr>
<tr>
<td>NPC behaves unpredictably.</td>
<td>1</td>
</tr>
<tr>
<td>NPCs are aware of each other.</td>
<td>2</td>
</tr>
</tbody>
</table>
The overall score for an NPC is made by multiplying scores from both aspects. Therefore, the overall score is always somewhere between 0 and 100. It is good to note that even if a game scores, say, fair scores of 5 from movement and animation and 5 from behavior, its overall score will be as low as 5 \* 5 = 25. Correspondingly, if a game receives an overall score of 81, it should gain very high \( 81 \) on average from both tables.

Therefore, we split the multiplied score finally into one dimension with five grades with text labels: sub-standard (score of 0-9), weak (10-29), satisfactory (30-54), good (55-79) and excellent (80-100). Labeled grades are included because the scores become intuitively more understandable, compared to the mere numeral score.

The thresholds for each labeled grade differ from each other, because when the overall score is the result of the two multiplied sub-scores, it is more likely to gain a score from somewhere in the middle than a very low or a very high score. By changing the limits of the grades or the importance of a requirement would give different results than those described in this paper.

Despite what the overall grade an NPC receives, it is easy to see whether or not its main aspects are in balance between each other. If they are, a player may place NPCs believability higher than what it really is. Correspondingly, even if overall grade of an NPC is high but the aspects scores differ much, NPCs may seem more unbelievable to a player than what the grade suggests.

The chosen policy to multiply scores results into a zero score if one of believability aspects gives a zero score. Any self-respecting game developer should not release an FPS game which does not meet even one requirement of both aspects, because it shows nothing but negligence towards NPC believability.

### 4 APPLYING FRAMEWORK

We examined three different FPS games published between 1993 and 2001 by our framework. They were *Doom* (1993), *Quake II* (1996) and Tom Glancy’s *Ghost Recon* (2001). The games were chosen because they represent the timeline of FPS game development from the player’s viewpoint. The case studies were conducted with PC-versions of the games by playing the single player mode using the medium level of the games (Doom 3/5, Quake II and Ghost Recon 2/3). Possible differences of NPCs’ believability caused by the levels of difficulty or multi-player vs. single-player modes are not included in the evaluation.

*Doom* received points as follows:

<table>
<thead>
<tr>
<th>Requirement for NPC</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPC can find most suitable path for its destination.</td>
<td>1</td>
</tr>
<tr>
<td>NPCs are aware of each other and do not collide with each other.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s appearance is done carefully and no unnatural features can be found in it.</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>

The combined overall grade for *Doom* is \( 3 \* 3 = 9 \), which is sub-standard. The scores from both aspects appear to be in balance.

*Quake II* received points as follows:

<table>
<thead>
<tr>
<th>Requirement for NPC</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPCs are aware of each other and do not collide with each other.</td>
<td>1</td>
</tr>
<tr>
<td>NPC can avoid any dynamic or static obstacle in game field.</td>
<td>2</td>
</tr>
<tr>
<td>Shifting from one animation to another is fluent.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s appearance is done carefully and no unnatural features can be found in it.</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5</strong></td>
</tr>
</tbody>
</table>

The combined overall grade for *Quake II* is \( 5 \* 5 = 25 \), which is weak. The scores from both aspects appear to be in balance. *Tom Glancy’s Ghost Recon* received points as follows:

<table>
<thead>
<tr>
<th>Requirement for NPC</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPC can find the most suitable path for its destination.</td>
<td>1</td>
</tr>
<tr>
<td>NPCs are aware of each other and do not collide with each other.</td>
<td>1</td>
</tr>
<tr>
<td>NPC can avoid any dynamic or static obstacle in game field.</td>
<td>2</td>
</tr>
<tr>
<td>NPC has different animations for one action.</td>
<td>1</td>
</tr>
<tr>
<td>Shifting from one animation to another is fluent.</td>
<td>1</td>
</tr>
<tr>
<td>NPC’s appearance is done carefully and no unnatural features can be found in it.</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>

The combined overall grade for *Ghost Recon* is \( 7 \* 6 = 42 \), which is satisfactory. Aspects are only 1 point apart from each other, so they are relatively well-balanced.
5 SUMMARY

Defining artificial intelligence has never been easy during its over 50-year-old history. Today AI research is based upon defining intelligence as intelligent behavior. Despite the fact that the first AI studies were done with board games, gaming has not been the driver in modern academic AI research. Contrary to academic AI research, game AI development has pursued to create an illusion of intelligence instead of trying to create one, ever since the 1970’s when the first arcade games were introduced.

The first two decades in computer games were mostly attempts to increase the quality of graphics of the games, instead of concentrating on what was behind the glittering surface. Ever since the first FPS games came to the market in the early 1990’s, NPCs’ believability has gained more and more attention in the development of the games. The ultimate goal is that no player could distinguish a human player from a computer controlled one.

The means to improve NPCs’ believability can be divided into three: movement, animation and behavior. Various algorithms and programming methods have been introduced and used by the game industry to improve NPCs’ believability.

In this paper, we described a framework for evaluating the user-perceived believability of NPCs. The framework is divided into two main aspects, which both can be judged independently. The overall grade which an NPC or a game receives from the evaluation comes when the scores from both main aspects are multiplied together. The grade can be anywhere between 0 and 100 and is divided into five verbal grades: sub-standard (0-9), weak (10-29), satisfactory (30-54), good (55-79) and excellent (80-100).

We applied the framework to three FPS games and the overall scores were: Doom: 9 (sub-standard), Quake II: 25 (weak) and Tom Glancy’s Ghost Recon: 42 (satisfying). Based on these results, it can be concluded that the investments of the game industry on NPCs’ believability since the 1990’s has produced results: the newer the game, the more believable the characters.

The framework is simple, but it is aimed to serve as a first step in an area of great importance: to construct a neutral and general framework for evaluating contents of digital games. Similar framework can easily constructed for different games with emphasizes altered as needed. The results obtained are two folded: first to evaluate existing games and second to influence to future games.

In the future, the evaluation of the framework should be done with a large number of game players. The parameters could be altered based on the common consensus of the players. It might well be that some of the attributes of the framework, such as “logical and human behavior”, should be elaborated further to make the framework provide more reliable results.

REFERENCES

LPI: Approximating Shortest Paths using Landmarks

Kevin Grant 1 and David Mould 2

Abstract. We present LPI, a novel approach for finding approximate shortest paths in weighted graphs. The basis of the algorithm is the presence of landmarks placed at vertices throughout the graph. Each landmark stores the shortest path from itself to all other vertices. LPI works by searching the intersection points along the landmark paths to approximate a path between two points. We show that the paths computed by LPI are quite accurate: within 1% of optimal in the majority of cases, while reducing the cost of exact search.

1 Introduction

Finding least-cost paths through weighted graphs is one of the classic problems in computer science. In many computer game applications, numerous path queries will be undertaken on the same graph. When games are the application domain, it may not be critical to report the optimal path, but performance (time and memory usage) is of enormous importance.

We propose a novel linear-time algorithm that can find high-quality approximations to optimal paths. The algorithm depends on preprocessing to create landmarks: nodes for which the distance to every other node in the graph is known [4]. An approximate path from start to destination is obtained by concatenating a partial path from one landmark to the start node with a partial path from another landmark to the destination node. We call our algorithm LPI (Landmark Pathfinding between Intersections), since it finds its path between the intersections of landmark paths.

2 Background and Previous Work

We consider the task of finding the shortest path between two vertices in the graph with v vertices and m edges; a request for a shortest path is called a query. We use S and G to refer to the start vertex and end vertex of our path, respectively, and the optimal distance between two nodes A and B is written d(A, B).

There are many algorithms for finding least-cost paths between pairs of vertices in a graph. Beginning at vertex S, the well known BFS algorithm searches through vertices by increasing distance from S, until G is found. The A* algorithm [5] is based on BFS, but uses a heuristic to estimate remaining distance from each vertex to G; vertices are expanded in increasing order of g + h, where g is the calculated distance from S to a vertex, and h is the heuristic’s estimated distance from that vertex to G. A* can provide considerable speedup over BFS, but requires the existence of a good heuristic.

Our algorithm requires solving the single-source shortest path (SSSP) problem for several vertices. The SSSP problem refers to finding the shortest path between one vertex (say S) and all other vertices in the graph, and can be solved by Dijkstra’s algorithm [3]. Dijkstra’s algorithm terminates with known values for d(S, V) for every vertex V; greedy hill-climbing from V can reconstruct the shortest path from S to V in linear time in the number of edges on the path. Dijkstra’s algorithm computes this information in O(m + n log n) time and O(n) space. Once this information is computed, finding the shortest path from S to any other vertex V requires no search, just a linear series of lookups: begin at V, and follow the previous pointers back to S, recording the path as you go. The operation requires O(k) time, where k is the length of the path. The memory required for storing the path information is O(n), storing two values for every vertex (distance and a pointer to the next node).

An algorithm with similar goals to LPI is HTAP [7], which also relies on preprocessing to accelerate runtime path planning and produces approximations to the optimal path. HTAP uses its preprocessing to create a hierarchy of abstracted graphs; runtime queries are handled by creating one query for each level of the hierarchy, and using the results of searching the low-resolution graphs to constrain subsequent higher-resolution searches. Because search takes place in a constant-width “corridor” at each level, HTAP’s time complexity is linear in the number of edges in the final path. HTAP uses O(n log n) space to store the hierarchy. However, the constraint means that the optimal path may not be found.

Similar to HTAP, the HPA* algorithm [1, 6] preprocesses a graph by subdividing it into a set of smaller subgraphs. Within each subgraph, the shortest paths are precomputed and stored. This allows path queries to be solved by searching a smaller global graph, as each subgraph can be crossed in a single step. Additional levels can be added to the hierarchy. A key difference between HTAP and HPA* is their target domains: HPA* computes over obstacle graphs, where vertices are either blocked or unblocked, traversal to blocked vertices is forbidden, and the cost of traversal between two unblocked adjacent vertices is essentially uniform across the graph. HTAP solves queries for weighted graphs. It is the latter case that we are interested in, and our algorithm will be evaluated against HTAP.

The ALT class of algorithms [4] uses landmarks to compute heuristic values for guiding an A* search. Specifically, let V and G be two vertices in the graph, and L be a landmark. By the triangle inequality, |d(V, L) − d(L, G)| is a lower bound for the distance d(V, G). The largest lower bound over all landmarks is used to estimate the cost between a vertex V and the goal vertex G; this estimate is used as the heuristic value to guide the A* search. The ALT algorithms are similar to LPI in that they both use landmarks. The primary difference between the ALT algorithm is that the landmarks are used only to guide the search, but the algorithm is free to explore any path between vertices. By contrast, the paths found by LPI are restricted to follow the paths stored at each landmark. This means that the paths of LPI are not guaranteed to be optimal, but it effectively

1 Dept. of Math and Computer Science, University of Lethbridge, Lethbridge, AB, email: kevin.grant2@uleth.ca
2 Department of Computer Science, University of Saskatchewan, Saskatoon, SK, email: mould@cs.usask.ca
eliminates the “search” employed by ALT. In the evaluation section, we give a cost comparison of LPI to ALT.

3 LPI: Approximate Pathfinding using Landmarks

A landmark \( L \) is a vertex for which a shortest path is known to all other vertices in the graph. For a path query between nodes \( S \) and \( G \), a simple and quick method for path generation is to concatenate the path from \( S \) to \( L \) (called the upswing) and the path from \( L \) to \( G \) (called the downswing). The length of the concatenated path is \( d(S, L) + d(L, G) \). Figure 1 shows some examples of the paths generated by such a strategy, where obstacles are shown in black. As the figure demonstrates, the quality of the path depends on the map topology, as well as the landmark’s location relative to a shortest path between \( A \) and \( B \).

Once the landmarks are constructed, the time and space required for finding the path to and from the landmark is linear in the length of the path. To compute the information for a landmark, we use Dijkstra’s algorithm, although any SSSP algorithm will suffice. If the graphs are unchanging, then the landmark information can be computed offline, and its running time can be amortized over all runs of the algorithm.

This idea of quickly generating paths from landmark paths will be the basis of our approach; the following subsections will improve upon this basic design. The motivation behind this approach rests on the observation that in many graphs, the shortest paths between many pairs of nodes favour a small subset of the edges. For instance, in the graph from Figure 1, the shortest path between any two nodes on opposite horizontal sides of the obstruction will traverse along the top edge of the obstruction. The same observation was used in designing the HTAP algorithm when constructing its hierarchy of graphs: nodes that survived into a higher level were “those which lie on the greatest number of optimal paths among the entire ensemble of paths” [7].

Not surprisingly, the landmark paths do not guarantee good approximations to the shortest path. The path generated in Figure 1(b) deviates a considerable amount from the optimal path. Figure 3(a) shows a near-pathological case – the path through the landmark is roughly 60 times as long as the optimal. In this section, we consider some extensions to the initial algorithm design. We refer to the final result as the LPI algorithm.

Path Overlap. For each landmark, the union of the stored paths form an implicit tree rooted at the landmark’s vertex. Rather than traversing from \( S \) to root, we need only traverse from \( S \) to the least common ancestor (LCA) of \( S \) and \( G \) in this implicit tree (the same argument applies to \( G \)). Following the path past the LCA creates overlap, as shown in Figure 2(a). By avoiding this overlap in the upswing and downswing, the quality of the approximated path can be improved substantially (Figure 2(b)).

Traversing to the LCA and traversing to the landmark have the same time complexity: both algorithms are linear in the number of vertices of the path. Traversing to the LCA requires a bi-directional traversal, starting at both \( S \) and \( G \), where at each step, we move from the point that is farthest from the landmark along the stored path towards the landmark. This algorithm ensures that both points will meet at the LCA without moving past it.

Multiple Landmarks. The quality of a path depends on the placement of the landmark relative to the path endpoints, but it is not trivial (and perhaps not even possible) to place a landmark so as to produce good paths for all endpoints.

An optimal landmark placement for one path query might be a near-worst case placement for another path query. Strategic placement of the landmarks can reduce the likelihood of bad paths. However, we can alleviate the problem simply by allowing multiple land-
marks to exist in the graph simultaneously. When searching for a path between two points, one could consider the paths generated by each landmark, and take the best one. More generally, we will store a set of landmarks throughout our graph. We use the algorithm from the previous section to compute the path generated by each landmark (up to the LCA), and then taking the minimum amongst all paths. Figure 3 shows how pathological cases are eliminated from a simple maze by introducing a second landmark.

While multiple landmarks increase the likelihood of finding a short path, they bring with them some concerns. For one, each landmark requires memory linear in the number of nodes in the graph, which can be quite expensive; each landmark in a graph of 10⁹ nodes would require space in the order of megabytes. Hence, the host architecture of the application may constrain the number of landmarks to be very few. Another concern with keeping multiple landmarks in the graph, and a further reason for minimizing their number, is running time. Finding the path associated with each landmark takes \(O(k)\) time (\(k\) being the length of the path), which means that \(r\) landmarks require \(O(rl)\) time. Note that this is an upper bound - not all paths need be expanded in some cases. For instance, in some fortunate cases, \(S\) will lie on the path from \(G\) to landmark \(L\), or vice versa, meaning that the optimal path has been found. Finding such a path will terminate the search for a better one. As well, when one solution is found, it places an upper bound on the cost of a future solution - we can terminate the expansion of a path through a particular landmark as soon as its length exceeds that of the best path found so far. An assessment of the effect of landmark count on path quality will be considered in the Evaluation section.

Combining Landmark Information. Consider the path query presented in Figure 4. Figure 4(a) shows the path generated by the left landmark. The path generated by the right landmark would be a mirror image of the left path. Neither path is better than the other, and neither is of particularly high quality.

Previously, we discussed the LCA of two vertices in the context of a single landmark. Recall that the LCA in the context of a landmark \(L\) was the location where the path from \(S\) to \(L\) meets the path from \(G\) to \(L\). With multiple landmarks, we have further intersections to consider. Consider where the upswing to the right landmark meets the downswing to the left landmark (call this point \(I\)). The path from \(S\) to \(I\) to \(G\) represents a much better approximation to the optimal path than was generated from each landmark individually (see Figure 4(b)). It turns out that we can locate these intersection points while traversing the paths to each LCA, with no effect on time complexity.

To find these intersections, we expand each upswing and downswing exactly as before, but we label each vertex that we pass with two values. In the case of an upswing expansion towards landmark \(L\), when a vertex \(V\) is reached, we label that vertex with \(d(S, V)\), since this value is readily available during this traversal (as \(d(S, L) - d(V, L)\)); we’ll refer to this label as \(d_s(V)\). In the case of a downswing expansion, when a vertex \(V\) is reached, we label that vertex with \(d(G, V)\); we’ll refer to this label as \(d_g(V)\).

As the traversal is made, we can simultaneously check for intersections between upswings and downswings. When we label vertex \(V\) on a downswing, we can check to see if \(d_s(V)\) has been given a value (and vice versa when checking an upswing). If this is the case, then the cost from traversing from \(S\) to \(G\) through \(V\) is \(d_s(V) + d_g(V)\). Remembering the intersection with the lowest cost, we approximate our path by following backwards back from this spot to \(S\) and \(G\) (this is easily accomplished in the same manner as following Dijkstra paths).

Landmark Subsets. Each landmark in our graph adds to the number of paths searched when looking for a solution. However, some landmarks are more likely candidates to produce short paths for some queries. To reduce the cost of search, we employ the same strategy as ALT: we do not necessarily search all landmarks – only a subset of the landmarks are considered. We use the heuristic of first investigating landmarks whose distance to the start or goal vertex is shortest; in practice, the landmarks suggested by this heuristic do yield high-quality paths. More sophisticated landmark selection strategies are a topic for future work.

4 Evaluation

To evaluate LPI, we use the same terrains used to test the HTAP algorithm. These include four 512 × 512 images (Mandrill, Lena, Peppers, and Terrain), one simple maze (243 × 243), one complex maze (340 × 340), and several random terrains of varying size.

We first look at the effects of landmark count on path quality. To do this, we distribute 16 landmarks on each map by dividing the map into 16 equal-sized squares and placing a landmark at the center of each square. We then compute the LPI path for 5000 randomly generated queries, using the 2, 4, 8, 16-closest landmarks.

Figure 5 shows the results for four of our test maps, with one map per graph. Each line in the graph represents the number of closest landmarks used to compute the path. The graphs show path quality: the horizontal axis is the percentage difference of the approximated path’s cost from the optimal path, while the vertical axis is the percentage of queries that fall within the suboptimality range shown on the horizontal axis.

As we increase the number of landmarks used to generate a path, the quality of the path increases, on average. However, the rate of increase in quality drops off rapidly as we add more landmarks. Doubling the number of landmarks from 2 to 4 has a much greater effect than doubling from 8 to 16. This suggests that good paths can be generated using only a few landmarks.

We next consider how the overall number of landmarks affects the path quality. To do this, we place 4, 9, 16, and 25 landmarks on each map. We then compute the LPI path for the same 5000 random queries, using the 4 closest landmarks. We plot these results using the same graphing procedures as in Figure 5. Figure 6 shows the results.

We see that increasing the number of landmarks in the map increases the likelihood of good approximations. In this case also, the marginal improvement decreases as the number of landmarks increases. This is an encouraging result, suggesting that reasonable paths can be generated with relatively few landmarks.

Next, we analyze LPI’s scalability. That is, we empirically evaluate how well the algorithm performs with increasing graph size. To test this, we generated several random images (each pixel takes a
We compare HTAP to LPI with 6 and 9 landmarks, as 6 landmarks degrade much slower as image size increases. However, the rate of degradation is slow. For instance, the number of paths found that were within 1% of optimal was 71.5% for images of size $243 \times 243$, 56.3% for images of size $1024 \times 1024$. Increasing the number of landmarks decreased this rate of degradation. Also, when we reduce the accuracy threshold, the numbers also degrade much slower as image size increases.

Finally, we compare the LPI algorithm to other algorithms for computing shortest paths in terrain maps. We include comparisons to two algorithms: ALT and HTAP, both discussed in previous sections. We choose ALT as it uses landmarks in a similar way to LPI (albeit to compute an exact path), and HTAP as it is an approximation algorithm that exploits preprocessing. Table 1 compares the quality of the paths generated. The table does not include an explicit comparison to ALT, as ALT returns an optimal path; path quality is expressed with respect to the optimal, so ALT’s path quality is implicitly compared. We compare HTAP to LPI with 6 and 9 landmarks, as 6 landmarks use roughly the same memory as the $3 \times 3$ hierarchy of HTAP for these test terrains. As with previous tests, the 4 closest landmarks are used to compute a path. Each row in the table corresponds to a test terrain, and each entry refers to the percentage of query paths that were within $x\%$ of the optimal path length, where $x\%$ refers to the column label. For example, top left entry in the table $(15,3)$ means that for image 1 (Mandril), 15.3% of the paths found by HTAP had cost that were within 1% of optimal.

LPI performs favourably in comparison to HTAP. The majority of LPI paths are typically within 1% of optimal, while the numbers are typically lower for HTAP. LPI performs particularly well in the random images, finding paths within 1% of optimal in over 45% of its test cases with 6 landmarks (by comparison, HTAP found paths of the same accuracy less than 1% of the time). When 9 landmarks are used, LPI finds paths within 1% of optimal in over 65% of cases. As regards runtime, LPI and HTAP require very similar costs, with HTAP typically performing slightly better on lower-cost paths, and LPI outperforming HTAP on higher-cost paths.

Table 2 compares the computation costs of ALT and LPI for finding the same paths reported in Table 1. When calculating computation costs, we use expanded vertices for ALT, and visited vertices for LPI. As with Table 1, each row corresponds to a terrain, and each entry refers to the percentage of query paths whose cost to find were less than $x\%$ of ALT’s cost, where $x\%$ refers to the column label. While ALT and LPI are similar in their use of landmarks, the al-

---

**Figure 5.** Path quality (percentage difference from optimal) vs. proportion of queries (percentage). The lines represent number of landmarks used.

**Figure 6.** Path quality (percentage difference from optimal) vs. proportion of queries (percentage). The lines represent number of landmarks present.

**Figure 7.** Path accuracy vs. image size.
The LPI algorithm reduces the online runtime of ALT considerably – an important property in an environment like a computer game, where processing time is scarce. Furthermore, LPI offers a dynamic time/quality tradeoff: more landmarks can be considered if processing is available, but if little processing is feasible, the results from few landmarks can still provide a path. Precomputing the landmark’s information is quite fast – less than one minute in all experiments. And unlike waypoints, the dynamic nature of the path intersections makes the computed paths less predictable and less prone to exploitation in a game setting.

There are many avenues of future research to consider. The memory requirements of landmark storage is an important issue, especially when we consider that more landmarks are required to maintain accuracy when the graph size increases. We are currently evaluating methods for reducing the amount of storage required by each landmark (e.g., storing a subset of the shortest paths at each landmark). Additionally, we have contemplated optimizations to the search component of LPI, such that we could find the best intersections sooner and terminate our search earlier. We will also consider other approaches to landmark placement, such as those considered in [4], and look at placement strategies for non-planar graphs.

ACKNOWLEDGEMENTS

We wish to thank Michael Horsch for his assistance with the HTAP algorithm, and the reviewers for their helpful comments. This research was supported by NSERC RGPIN 258424 and 299070-04.

REFERENCES


Table 1. Path length comparison between HTAP and LPI.

<table>
<thead>
<tr>
<th></th>
<th>HTAP</th>
<th>LPI (6 landmarks)</th>
<th>LPI (9 landmarks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1%</td>
<td>5%</td>
<td>10%</td>
</tr>
<tr>
<td>1</td>
<td>153</td>
<td>67.0</td>
<td>84.7</td>
</tr>
<tr>
<td>2</td>
<td>253</td>
<td>78.4</td>
<td>89.4</td>
</tr>
<tr>
<td>3</td>
<td>17.4</td>
<td>71.4</td>
<td>87.1</td>
</tr>
<tr>
<td>4</td>
<td>1.3</td>
<td>30.3</td>
<td>72.4</td>
</tr>
<tr>
<td>5</td>
<td>57.1</td>
<td>92.1</td>
<td>96.3</td>
</tr>
<tr>
<td>6</td>
<td>57.1</td>
<td>68.6</td>
<td>77.5</td>
</tr>
<tr>
<td>7</td>
<td>0.8</td>
<td>3.4</td>
<td>16.2</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.6</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Row labels: 1=Mandrill, 2=Lena, 3=Peppers, 4=Terrain, 5=Simple Maze, 6=Complex Maze, 7=Random (243x243), 8=Random (729x729).

Table 2. Computation cost comparison between LPI and ALT. Row labels are the same as for Table 1.

<table>
<thead>
<tr>
<th></th>
<th>LPI (6 landmarks)</th>
<th>LPI (9 landmarks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>1</td>
<td>38.9</td>
<td>77.9</td>
</tr>
<tr>
<td>2</td>
<td>52.4</td>
<td>73.8</td>
</tr>
<tr>
<td>3</td>
<td>61.4</td>
<td>79.8</td>
</tr>
<tr>
<td>4</td>
<td>57.7</td>
<td>76.5</td>
</tr>
<tr>
<td>5</td>
<td>51.7</td>
<td>74.9</td>
</tr>
<tr>
<td>6</td>
<td>6.5</td>
<td>26.6</td>
</tr>
<tr>
<td>7</td>
<td>18.8</td>
<td>43.7</td>
</tr>
<tr>
<td>8</td>
<td>53.4</td>
<td>73.8</td>
</tr>
</tbody>
</table>

LPI requires a fraction of the search cost incurred by ALT in most cases, improving on ALT by at least an order of magnitude in many cases. Regarding memory, the cost of ALT and LPI is roughly the same, as they both store the same landmark information.

Regarding offline costs, the LPI algorithm computes each landmark’s information in roughly 2 seconds on a 729 × 729 graph (using a 3.0 GHz Intel processor). The time to precompute the landmark information for the previous experiments was roughly 12 seconds and 18 seconds, respectively. These times would be the same for ALT, but are about an order of magnitude less than the reported time to compute the HTAP hierarchy, albeit on a faster computer.

The landmarks of LPI are in some respects similar to the concept of waypoints, another technique used in pathfinding [2]. Waypoints are points placed along good paths, and path-planning algorithms will typically search for optimal routes through a system of waypoints. The search component can be optimized by pre-computing short paths between each pair of waypoints. However, clever game players can often ascertain the location of these waypoints and exploit this knowledge to their advantage (for instance, setting up defences at waypoints). By contrast, the LPI algorithm uses the intersections of the paths from landmark to endpoints when searching for short paths. These intersections can vary amongst different pairs of start and end points, making it more difficult to exploit known paths.

5 Discussions and Future Work

We present LPI, a novel approach for finding approximate shortest paths in terrain maps. The basis of the algorithm is the presence of landmarks, placed at different vertices throughout the graph. Each landmark stores the shortest path from itself to all other vertices. LPI works by searching for intersection points along the landmark paths to approximate a path between two points. On evaluation, the paths computed by LPI are reasonably accurate – within 1% of optimal in roughly 75% of cases with 9 landmarks placed in the graphs. The LPI algorithm reduces the online runtime of ALT considerably – an important property in an environment like a computer game, where processing time is scarce. Furthermore, LPI offers a dynamic time/quality tradeoff: more landmarks can be considered if processing is available, but if little processing is feasible, the results from few landmarks can still provide a path. Precomputing the landmark’s information is quite fast – less than one minute in all experiments. And unlike waypoints, the dynamic nature of the path intersections makes the computed paths less predictable and less prone to exploitation in a game setting.

There are many avenues of future research to consider. The memory requirements of landmark storage is an important issue, especially when we consider that more landmarks are required to maintain accuracy when the graph size increases. We are currently evaluating methods for reducing the amount of storage required by each landmark (e.g., storing a subset of the shortest paths at each landmark). Additionally, we have contemplated optimizations to the search component of LPI, such that we could find the best intersections sooner and terminate our search earlier. We will also consider other approaches to landmark placement, such as those considered in [4], and look at placement strategies for non-planar graphs.
Modelling a RTS Planning Domain with Cost Conversion and Rewards

Vidal Alcázar and Daniel Borrajo and Carlos Linares

Abstract. Game playing has always been a good testbed for AI research. In particular, real time strategy (RTS) games are characterized by being complex environments which offer multiple opportunities for AI researchers. In this work, we use AI planning and apply it to a RTS game. More specifically, we have defined a domain using PDDL2.1 that can be used by a computer player to compete in a RTS game setting using automated planning. The goal is to participate in the annual ORTS competition and we have focused on one of its tasks. We have used techniques such as cost to operator conversion and rewarded soft goals to overcome the difficulties of modelling this kind of tasks.

1 Introduction

Among games, RTS is a particularly difficult type of game with many analogies to real world problems, such as those in robotics. Specifically, RTS games differ from classical games in the challenges they pose to computer players: real time reasoning, complex worlds, uncertainty, resource and technology management, concurrency, collaboration,… So far the main problem is that RTS games have been created by private companies that are mainly motivated by economical targets. Thus, most games are proprietary source projects. Also, most computer players lack real AI implementations since their objective is to provide a good gaming experience. Currently, except for pathfinding, only a handful of games use real AI techniques, as for example FEAR, where some limited automated planning has been successfully used [9].

Fortunately, some research dealing with AI in RTS games has been done in the AI science community in the last years thanks to the upcoming of new tools like Wargus, MadRTS and ORTS [10]. Examples of this research are [1], where case-based plan selection is used to design a client that fully plays a standard RTS game simulated with Wargus with some degree of success; [7], which uses MadRTS to implement a learning Hierarchical Task-Network planning architecture; and [2], which uses automated planning to solve the problem of resource gathering, again using Wargus. Other research projects that apply AI, though not focussing on RTS games, are [3, 6], just to mention a few.

The first challenge when applying AI planning technology to real-world tasks is the domain modelling. Specifying the right preconditions and effects of actions is a hard task, especially when using metrics like costs and rewards. In this paper, we present the modelling decisions that we took in order to build an operative domain model for one of the domains in the ORTS competition. ORTS (which stands for Open Real Time Strategy) is a RTS game engine that allows users to define RTS games in the form of scripts. In our case, we have designed a planning domain using as reference the third game of the annual ORTS competition, which is quite similar to their commercial counterparts. Since we are interested in the strategic level of the game, we have used automated planning using PDPLL as the domain specification language. PDDL 2.1 [11] was chosen since the domain involves temporal actions with numeric effects. Time and concurrency were handled using costs and thus only level 2 of PDDL2.1 expressiveness must be supported by the planner, allowing a wide set of planners to be used.

This paper is arranged as follows: Section 2 gives a description of the characteristics of the problem. Section 3 describes the techniques used in the definition of the RTS domain. Section 4 analyzes the results of the experimentation done in terms of quality and performance. Finally, Section 5 summarizes the conclusions drawn from the previous sections and suggests new lines of research.

2 Description of the ORTS game

The main characteristics of RTS appear in this game, though simplified so that it is easier to compare the core of the different techniques. Basically, it is a game with two players in which the objective is to annihilate the opponent. Fog of war is always active (only the part of the map seen by allied units is updated). There is a simple tech tree and the games involve both economy and military management, as mineral must be gathered to produce new buildings and units. There are three kinds of units: workers, which can attack, build new structures and gather mineral; and soldiers and tanks, which only attack. Also, there are three kinds of buildings: control centers, which produce workers and are used to deploy minerals; barracks, which produce soldiers and require a control center to be built; and factories, which produce tanks and require a control center to be built. Time is measured in discrete frames of equal duration, using a pace of 8 frames per second. The world is a $1024 \times 1024$ grid divided in $64 \times 64$ tiles that can be classified either as ground or as cliff —so that units cannot traverse tiles marked as cliffs. There is only one resource called minerals, which are spread out on the map in clusters of “patches” of a limited amount of mineral. A maximum of four workers can simultaneously harvest one mineral patch. Minerals must be delivered to any control center before they can be used. Workers may mine at a rate of 1 mineral every 4 simulation frames, carrying a maximum of 10 minerals at any time. The maps are randomly generated every match, though they always have similar characteristics. The initial state is always the same: both players start with 6 workers, a control center, and 600 minerals each, and a nearby mineral patch cluster on randomized terrain, with asymmetric start locations.
3 Domain definition in PDDL

As the problem to be solved is far more complex than the average planning problem (compare with the simpler domains typically used in the International Planning Competitions), in this case the objective is to obtain coherent plans in a reasonable time. With “coherent” we mean using as many units as possible and coordinating them to achieve a certain goal. In this section we describe how the domain has been designed justifying the modeling decisions prior to analyzing the obtained results.

3.1 Abstraction

A common way of modeling in computing, independently of the context, is abstraction. In many problems there are too many factors to deal with, making it unviable to compute them, so a way of reducing the details of the problem without losing key information is needed.

In the domain definition for the game several abstractions were used, some of them being:

- Only allied units and buildings will be handled individually. Enemy buildings will be represented by a fact enemibase independently of the existent buildings. Enemy military units will be associated as a fluent either to enemy bases (when defending them) or to allied control centers (when attacking them). Workers will not be factored in.
- Due to the characteristics of the game, tanks have been taken out. Based on their stats, it is not clear whether tanks are a better option to soldiers cost-wise, so to simplify the problem they will not be used in planning. Of course opponents are free to use them, so they will be represented as 4 soldiers to take them into account.
- The map is composed of a 64×64 grid of tiles, each one being a grid of 16×16 points (scaling up to a world of 1024×1024 effective positions). Most planners have scalability problems with such big maps, so the map has been divided in 64 equal areas to have a coarser representation of the world.
- The construction of control centers is assumed to be scripted because of the importance of having at least one and the issues related to positioning it close to mineral clusters. Taking advantage of this, minerals can be left out of the representation by associating the mine operator to control centers.

3.2 Unit consuming and intermediate actions

The easiest way of designing a domain for a game is correlating the possible actions of the units to actions in the domain in a straight way, reflecting every single action for every single unit. However, this is hardly a correct approach in RTS games, as the search space grows exponentially with the number of possible actions. Therefore we assumed a certain degree of autonomy for the units and assign them a task related to some goal. This means that, at some lower level, a behavior for the unit should be implemented for each action. This is common when planning for robots, since many high-level actions (output of a symbolic planner) must be translated to low-level behaviors that implement a more reactive control.

Another particularity is that assigning a behavior uses up that unit; this is, the unit cannot be used in another action once it has been given a behavior. This is justifiable for two reasons. On one hand, behaviors are not instantaneous actions; they take some time during which a new plan will probably be computed. Thus, assigning one behavior per planning action is good enough if replanning is done in a reasonable amount of time, provided that the state will probably not change that much for the behavior to become obsolete (such as a soldier attacking an already destroyed base or a worker gathering mineral from a depleted cluster). On the other hand, this makes planning easier, as it limits the number of units that can be chosen as a parameter in subsequent actions.

There are five of such actions, which use the position of the objects to determine their cost:

- **Attack**: It has as parameters a soldier and an enemy base. A simple implementation of this behavior would make the unit advance towards the enemy base shooting enemies as it encounters them, prioritizing military units over workers and workers over buildings.
- **Defend**: Analogous to Attack, but having as parameter an allied base instead. Enemy workers and buildings would be ignored in this behavior.
- **Harass**: Like Attack, it has as parameters a soldier and an enemy base. It is similar to attacking with the exception that flees when opposing military units, instead of engaging them in combat. Its objective is deviating the attention of the enemy, not making a direct attack.
- **Scout**: It has as parameters a worker and an unexplored area. A way of implementing this behavior could make the worker wander around the target location in a spiral while avoiding enemies.
- **Mine**: It has as parameters a worker and an allied control center. This action consists on a worker going back and forth from the control center to a mineral cluster, mining mineral and deploying it to obtain resources.

However, some of the actions can be done in a given amount of time and the units or buildings performing them can be re-assigned to do something else. In terms of the domain definition, this means that the unit or building is not used up. These intermediate actions are Buildworker, which produces a new worker from a control center, Buildsoldier, which produces a new soldier from a barracks and Buildbarracks, in which a barracks is built by a worker.

3.3 Rewarded soft goals

Coming out with a clear definition of the goal is often one of the most difficult issues when designing a planning task. RTS games have a winning condition, but a classical approach like winning is out of the question, because of its complexity. Rather, each problem generated by the playing client must have simple goals that can be computed in real time. Defining a condition-based goal that involves achieving as many soft goals as possible is a very difficult target, so a different approach has been considered: rewarded soft goals. This involves: first, the goal condition in the problem definition is getting at least a certain number of rewards by achieving soft goals (which might be even 0, meaning that there is no hard goal); and secondly, rewards are obtained by using units in actions that lead to achieving a certain soft goal (for example, using **Attack** N times with different soldiers to destroy an enemy base).

Actions are rewarded accordingly to their utility. For instance, a soldier that attacks an undefended base will get a higher reward than another one attacking a heavily defended base —expectedly resulting in being outnumbered by the defending forces. Besides, soft goals are often mutually exclusive because the units are limited and using up units to achieve a given goal implies that we cannot use them to achieve a different one. The exception to this kind of actions is the **Mine** action because mining itself does not represent a direct benefit.
by itself, being used instead as the previous step of producing a new unit which could get a reward doing another task.

Defining the reward for scouting and harassing is quite straightforward, as the more units are dedicated to it, the less useful it is to use another one—i.e., rewards are strictly diminishing. However, in the combat-oriented actions, the best rule usually consists of assigning a few more soldiers to attack/defend than the ones owned by the enemy, since fewer may mean being defeated and more would mean overkilling. As actions are related to a single unit, the reward must be computed per unit with the additional problem that the planner does not know “a priori” how many units will be attacking or defending the same point (as those would be actions performed afterwards, because of the sequentiality of plans in automated planning). Hence, our solution is based on a Gaussian function like the following:

$$\text{Reward} = a \times e^{-\frac{(x-b)^2}{2c^2}}$$

(1)

where $a$ is the maximum reward a unit can get, $b$ is the position of the maximum reward (which should correspond to the number of enemy soldiers in the opposing force plus a little value) and $c$ the width of the Gaussian, which should have a value so that $f(0)$ would be slightly greater than 0. When attacking, for example, the first soldier will contribute with almost no reward. But, if $(b - 1)$ soldiers have already been assigned, the next soldier will get a hefty amount of reward. Therefore, assigning a soldier to attack when too few or too many have been already assigned grants so little reward that the planner will avoid doing so. Another advantage is that this is a conservative approach to attacking and defending, as it is unlikely that the planner decides to attack or to defend with just a few soldiers, encouraging an “all-or-nothing” behavior that leads to successful and more believable maneuvers. Nevertheless, it is recommended to transform the function into a linear one with as few computations as possible, so all the planners support it and complex computations are avoided when evaluating nodes.

### 3.4 Metric and cost to action conversion

As mentioned before, time and concurrency issues will be represented as costs integrated with the rewards system. Units and buildings can perform different actions, but these actions cannot be performed at the same time and the order of the actions performed by the same unit or building is a relevant factor. In our design, the concept around time is that the later an action is executed, the lesser its cost. The costs due to delays in actions are just the loss of utility that occurs when that action cannot be performed immediately. This happens only with actions that require a unit to move across the map and not when producing new units. The positions in this domain are represented by a fact with two associated fluents which represent their $x$ and $y$ coordinates. In addition, another fluent is used to indicate how difficult is for a unit to move across the area designated by its position. This difficulty is based on the static obstacles present in the area and the units (both enemy and allied) that move around it. So, the cost of moving is on one hand the Manhattan distance, and on the other, the difference between the roughness factor of the areas if the value of the target position is higher.

The other way of taking into account costs is associating an individual cost to every unit and building that has been involved in an intermediate action. That cost will be added to any other cost a subsequent action may impose, so units participating in several actions will have a higher associated cost. This is more useful for the unit and building producing actions: for instance, if a barrack is used to produce a soldier, a fixed cost will be added to the cost associated to the barrack and the cost associated to the new soldier. This way, if that soldier is used to attack an enemy base, its final cost will be higher than the cost of a soldier that already existed. Besides, if that barrack produces another soldier, this second soldier shall be assigned the production cost plus the cost the barrack already had associated, leading to the notion that it is counterproductive to produce many soldiers at the same barrack. This is reflected in the game in three different ways. First, units and buildings do not queue too many actions, which anyway could not have been done before replanning. Second, the intermediate actions are distributed so a single unit or building does not perform too many things if other units or buildings can do it, achieving some degree of collaboration at unit level. And, third, it deals with the time issues of newly produced units by assigning them an initial cost that otherwise would be zero.

The metric the planner will use will be maximizing the utility of the actions, this is, maximizing the sum of rewards minus the sum of costs. Actually, the real metric is minimizing the result of the following equation for a plan containing $n$ rewarding actions:

$$\text{TotalCost} = \text{InitCost} - \sum_{i=0}^{n} (\text{rewards}_i - \text{costs}_i)$$

(2)

As it can be seen this metric is non-monotonic because the value to minimize at each step of the plan depends on the state. This is an important issue because most state-of-the-art planners cannot deal with non-monotonic metrics (being able to optimize only metrics which are modified in actions by constants and proportionally minimizing the length of the subplan containing those actions). The proposed way of overcoming this problem is using cost to action conversion. Cost conversion standardizes the assignment of values to the fluent that will serve as a metric, so conventional planners can effectively use it to improve the plans. Using this approach does not require big modifications to the domain: the value to minimize is stored in a temporal fluent until the real goal is reached. Once the goal is fulfilled, an auxiliary action can be used. This action subtracts a constant value to the temporal fluent that holds the cost and adds that same value to another fluent. The new fluent is actually the real metric of the problem. In our domain this has been implemented using the actions in Figure 1.

```
{:action TO-END
  :precondition
    (and (> (rewards) (rewards-threshold))
         (> (pre-total-cost) 0))
  :effect
    (and (decrease (pre-total-cost) (cost-increment))
         (increase (total-cost) (cost-increment))))
}

{:action END
  :precondition
    (and (> (rewards) (rewards-threshold))
         (<= (pre-total-cost) 0))
  :effect (goal_achieved))
```

Figure 1. Actions that compute the real cost of the plan.

In this domain, the goal is getting a minimum amount of rewards, represented by the precondition $(> (rewards)$
(rewards-threshold)). Once the goal has been reached, the action TO-END will be used by the planner repeatedly to decrease the fluent that temporarily holds the value to minimize and, concurrently, increase the fluent used in the metric. The values are modified by a constant (in this case represented by a fluent that does not change) called cost-increment. When pre-total-cost, the temporal fluent, has a value below 0, the action END is finally invoked, adding the fact goal-achieved, which is the actual goal in the problem definition.

Plans generated under this scheme correspond to the same sequence of actions that would have been generated in a normal domain, with a slight difference. At the end of the plan, the action TO-END appears \( N \) times, which determines the effective plan length to minimize. An example of a plan would be the following one:

0: ATTACK SOLDIER0 ENEMYBASE0 POS3-3 POS1-1
1: HARASS SOLDIER1 ENEMYBASE1 POS5-1 POS4-2
2: SCOUT WORKER0 POS5-1 POS6-2
3: TO-END
4: TO-END
5: END

As it can be seen, TO-END appears twice, meaning that the final cost is between one and two times the constant cost-increment. Other than that, after discarding these last actions, the resulting plan would be a completely conventional one with no further modifications to the domain.

### 3.5 Object reutilization

One of the main problems PDDL2.1 has when defining domains for RTS games is the limited expressivity it has to represent sets of objects of the same type. In PDDL 2.1, if there are several equivalent objects of the same type, they must be enumerated; i.e., soldier1 = soldier, soldier2 = soldier, ... Fluents cannot be used to represent them. In our domain, we need objects to create new units. If a soldier is going to be built, an object of the type soldier that is not alive needs to be produced. For \( N \) soldiers, up to \( N \) different objects may be needed. The problem is that planners handle the objects independently even if using one or another would lead to totally equivalent states. For every object that could be a possible parameter of an action that builds or produces an object, a new node is expanded and a subsequent plan may be computed. This increases the number of states by the number of equivalent facts. Even worse, this happens not only at one point in the plan but every time the action can be evaluated, so in the end the computational increase is of exponential order. This has been already studied in the literature as asymmetries [8] and particularly addressed to RTS games in [5].

We use object reutilization to solve this problem. As shown in the previous sections, actions that provide a reward use up a unit. The fact that represents the unit can be used to represent a newly produced one, having into account that plans must be correctly interpreted (because the output will show the same object used several times while it really represents different units). However, if several units are used up the same situation occurs, but this time with objects that represented old units. To overcome this, the predicate ready is used. This predicate represents a possible new unit and is a requirement in all the actions that produce new units. We have a logic that prevents having more than one unit of a type with the predicate ready assigned at the same time: the ready predicate is assigned to a fact when a unit is used up only if there was no other unit with that predicate already. This ensures that there is only one object that can be used to represent a possible new unit. This technique can be implemented in PDDL2.1 using types and existential quantifiers, as done in the effects section of the Attack operator:

```plaintext
... (not (?soldier ?src))
when (not (exists (?x - soldier)(ready ?x)))
(ready ?soldier)
...
```

### 4 Experimental Results

The main objective of this work is generating coherent plans to be used in RTS games. In this section a plan for what could be a typical problem will be analyzed in order to check whether it could be used by a computer player. Suppose we want to solve a problem with 10 soldiers, 5 workers, and 100 units of mineral in the initial state. During the execution of this plan three allied bases were attacked by four, two and zero enemy soldiers respectively and two enemy bases were defended by three and seven enemy soldiers. The planner we used is METRIC-FF [4]. The plan it obtained was:

0: DEFEND SOLDIER0 CC1 POS4-4 POS3-3
1: DEFEND SOLDIER1 CC0 POS7-2 POS4-2
2: SCOUT WORKER2 POS7-2 POS7-2
3: DEFEND SOLDIER2 CC0 POS6-5 POS4-2
4: ATTACK SOLDIER3 ENEMYBASE0 POS1-3 POS1-1
5: ATTACK SOLDIER4 ENEMYBASE0 POS2-3 POS1-1
6: DEFEND SOLDIER5 CC0 POS5-3 POS4-2
7: ATTACK SOLDIER6 ENEMYBASE0 POS1-7 POS1-1
8: ATTACK SOLDIER8 ENEMYBASE0 POS4-1 POS1-1
9: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
10: ATTACK SOLDIER0 ENEMYBASE0 POS4-2 POS1-1
11: ATTACK SOLDIER7 ENEMYBASE0 POS3-2 POS1-1
12: SCOUT WORKER1 POS5-4 POS6-3
13: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
14: ATTACK SOLDIER9 ENEMYBASE0 POS4-2 POS1-1
15: MINE WORKER4 CC2 POS1-1 POS7-5
16: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER9
17: ATTACK SOLDIER9 ENEMYBASE0 POS4-2 POS1-1
18: MINE WORKER3 CC2 POS4-6 POS7-5
19: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER9
20: DEFEND SOLDIER0 CC0 POS4-2 POS4-2
21: MINE WORKER0 CC0 POS4-2 POS4-2
22: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
23: ATTACK SOLDIER0 ENEMYBASE0 POS4-2 POS1-1
24: DEFEND SOLDIER9 CC0 POS4-2 POS4-2
25: TO-END
26: TO-END
27: END

11.14 seconds searching, evaluating 13288 states
11.45 seconds total time

Objects are used in the problem definition to represent concepts, so we need a translation back into what they represent. CC0 is the allied base attacked by 2 enemy soldiers, CC1 is not attacked, CC2 is attacked by 4 enemy soldiers, ENEMYBASE0 is defended by 3 enemy soldiers and ENEMYBASE1 is defended by 7 enemy soldiers. In the plan, 5 soldiers defend CC0, another one defends CC1, 9 soldiers
attacking ENEMYBASE0, 3 workers scout, 3 gather mineral and 5 new soldiers are produced. The results are promising: the plan is attacking with the main force the least defended base while defending the two bases that are attacked by fewer soldiers and abandoning the base that is attacked by 4 soldiers. This relates to the way by which the number of bases determine the rewards: the more there are, the less rewarding is obtained by attacking or defending them. In this case, the enemy has two bases while the player has three, so if both sides lose one, the new state is favorable to the planning player. This is computed by dividing the rewards for attacking by two and the rewards for defending by three. Workers are also effectively used: the rewards for scouting decrease as individual workers are sent to explore undiscovered areas. So, in the first half of the plan two workers are commanded to scout; however, as the plan advances the planner decides that it is more profitable to use them for mining than scouting, so new soldiers can be produced and used for attacking and defending, which is actually a very coherent decision. Besides, this way of using the soldiers leads to what is probably the most important fact in the plan: the planner uses all the available resources (100 minerals from the initial state and 150 additional minerals from mining) to produce new soldiers, which means that it really plans ahead and tries to get the most out of the resources, either units or minerals, it has at its disposal. This is exactly the behavior that was expected to get when the design was conceived, so the plans generated using the appropriate parameters can be considered as quite good.

Apart from the quality of the plan, the other factor that determines its usability is the time needed to generate it. In this case, a total of 13288 states were evaluated, taking up to 11.45 seconds. This result is not as good as the quality because in 12 seconds the state of the game can change significantly depending on how fast paced the RTS game is. In particular, the pace of game 3 in ORTS is not very fast, but more than 10 seconds may be on the verge of being unusable. This particular problem was parameterized looking for quality over performance, so this result could be improved to make the problem more manageable at the cost of losing quality. Besides, using dedicated hardware could easily allow the search to be finished in a more reasonable time.

It is also interesting to see how the complexity of the problem scales as several parameters change. Many of the parameters of the domain are set by the developer, and thus can be adjusted to get a good performance, but others depend on the game state. Perhaps one of the most relevant ones is the number of units, as one of the most typical problems in RTS games is dealing with a high number of units. In Figure 2 we can see that the time needed to generate the plan increases as the number of soldiers increases, but its impact is not that important. The last experiment, with 40 soldiers, took less than 5 seconds to be solved, which is a fairly good result taking into account that 40 soldiers is already a rather high number for a RTS game. Besides, time seems to scale linearly instead of exponentially, probably the most important feature when extrapolating this design to more complex problems. This can probably be explained by how plans are generated: in most cases, soldiers were used sequentially, meaning that they were usually selected depending on their order in parsing the problem by the planner. A similar case is when the number of workers increases, though the time needed is much higher. This was expected because of the higher number of actions involved: for a worker to get rewards, a whole sequence of actions involving mining, producing a soldier and either attacking, harassing or defending must be performed once scouting is not profitable enough.

5 Conclusions and future work

In this work we have defined a domain which can be effectively used to implement a computer player based on automated planning in a RTS game. A number of techniques have been used on the domain definition to overcome the limitations of the definition language and current planners, most of which can be easily extrapolated to different domains not related to gaming. In the future, we will implement a client using this idea to participate in the 2008 ORTS competition. Other possible lines of research include using different planners to study the interaction of the techniques with different search algorithms and heuristic functions and analyzing the possibilities of integrating these techniques in a planner to avoid having to use them explicitly in the domain definition and therefore enhancing performance.

REFERENCES

[1] Aha, Molineaux and Ponsen, Learning to win: Case-based plan selection in a real-time strategy game. Proceedings of the Sixth International Conference on Case-Based Reasoning (pp. 5-20). Chicago, IL: Springer 2005
[6] Laird et al., Using a Computer Game to Develop Advanced AI, Computer, 34 , July 2001, pp. 70-75
Towards Engaging MDPs

Olana Missura¹ and Kristian Kersting¹ and Thomas Gärtner¹

Abstract. One of the challenges that a computer game developer meets when creating a new game is setting the difficulty “right”. Not only is it a complicated task to balance all the game parameters, but the developer also needs to cater for players with very different skill levels. Providing a game with an ability to automatically scale the difficulty depending on the current player would make the games more engaging over longer time. While a few commercial games boast about having such a system, to the best of our knowledge it was not researched as a learning problem. In this paper we first give a problem definition of the automatic difficulty scaling problem we call Engagement Problem. Then, we also outline a framework based on nested Markov Decision Processes, called Engaging Markov Decision Process for solving it. Preliminary experiments in a small grid world show the effectiveness of our approach.

1 INTRODUCTION

One of the important questions that needs to be answered during a (computer) game development process is how to keep the game engaging for a wide spectrum of players and for a long time. The “interestingness” of a game depends on many different parts such as the storyline, the right amount of interaction, the attractive presentation, and others. The perceived difficulty of the game also contributes to the time that any given player spends playing it. Games that are too easy or too difficult become boring or frustrating fast. The traditional way of dealing with it is to provide a player with a way to set up the difficulty level for herself. Unfortunately this method is rarely satisfactory. Looking at the problem from the point of view of developers, it is not an easy task to map a complex gameworld into one variable. While constructed, such mapping requires additional extensive testing, producing time and money costs. Consider also the fact that generally games require several different skills to play them. It is natural to assume that each of these skills develops differently as the player progresses through the gameworld depending on her preferences and abilities. Providing the computer with an ability to adjust the game to all these skill levels automatically is more user-friendly than offering several settings for the user to be set.

A (by now classic) example of an automatic difficulty scaling in a computer game is provided by any good computer role-playing game (CRPG): As soon as a player’s character gains a new level, the monsters (CRPG) are adjusted accordingly, the “level” value being considered a universal indicator of a player’s strength. Unfortunately, even in the very numerical world of a typical CRPG, with all properties expressed in numbers and their relations in formulas, this single value represents neither the character’s strength, nor the player’s skill with the game. It is all too easy to create and develop a character who is not fit for the gameworld and its leveling up doesn’t mean it becomes any stronger, but the game follows its logic and becomes too difficult as it progresses. On the other hand, give a strong character together with the instructions on how to develop it during the game to a newbie, and the game would be too difficult for this player, even though the character can deal with it in potential.

The problem of automatic difficulty scaling can be viewed as an interaction between two agents. One of them, we call it a player, is a person (or a computer algorithm) who is playing the game. The second agent represents the game itself, tracking the player’s progress and attempting to keep her interested, to provide her with the tasks that are neither too easy, nor too difficult, but just right. We call this game agent the master. In this work we propose a general framework for the player-master interaction. The player’s task is to achieve her own goals, and the master’s task is to provide an environment for the player that keeps her challenged. In other words the master attempts to automatically adjust the difficulty of the player’s tasks. This problem can be informally formulated as follows.

Definition 1.1 (Engagement Problem). Given the player, who pursues her own (possibly unknown) goals, the master has to learn the policy to modify the player’s environment to keep her engaged over the long time.

In this paper, we propose a Markov Decision Process-based framework for solving the Engagement Problem. As this is the first attack of the problem, we provide the empirical proof of the principle in the experiments within a tiny grid world.

We proceed as follows: After describing related work in Section 2, we present the framework for the player-master interaction called Engaging MDPs in Section 3. Before concluding and listing open questions, we will present the results of our preliminary experiments that show the effectiveness of our approach.

2 RELATED WORK

Systems with multiple interacting agents are being researched in different contexts, the main distinguishing property being the amount of cooperation and the relations between the agents’ goals.

Littman in [7] considers the multi-agent reinforcement learning problem as a Markov game. A Markov game is defined by a set of states, S, and a collection of action sets, A₁, . . . , Aₖ, where Aᵢ is a set of actions of agent i. State transitions are controlled by the current state and one action from each agent: T : S × A₁ × · · · × Aₖ → PD(S), where PD(S) is the set of all discrete probability distributions over the set S. Each agent has an associated reward function, Rᵢ : S × A₁ × · · · × Aₖ → R ⊂ R, for agent i, and attempts to maximize its expected sum of discounted rewards, E(∑ₖj=0 γᵢjriₙ+j), where riₙ+j is the reward received j steps into the future by agent

¹ Fraunhofer-Institut fuer Intelligente Analyse- und Informationsysteme IAIS, Germany, email: {olina.missura, kristian.kersting, thomas.gaertner}@iais.fraunhofer.de
i and γ is the discount factor. Littman constrains a Markov game in question to the two-player zero-sum game, i.e. there are exactly two agents and they have diametrically opposed goals,

\[ R_i(s, a_1, a_2) = -R_2(s, a_1, a_2) \quad \forall s \in S, a_1 \in A_1, a_2 \in A_2. \]

This constraint allows the author to use a single reward function for both agents (one of them tries to maximize it, the other to minimize) and to find the agents’ optimal policies by applying a modified Q-learning algorithm [13]. Unfortunately this framework appears unsuitable for the Engagement Problem [11]. The main assumption of a Markov game considered in [13] is that the agents are strictly non-cooperative and have conflicting goals, whereas in our problem the master does not pursue its goal at the expense of the player, instead it provides a type of assistance by producing the tasks that are interesting for the player (neither boring, nor overwhelming).

The question of assistance is also investigated from the decision-theoretic perspective [3][6]. Here, the goal is to learn the user’s intentions and to use this knowledge to help him. The assistance consists of replacing the user’s actions (or suggesting the actions to the user) so that the new behaviour imitates that of the user as good as possible. The assistant’s goal is to minimize the expected cost of the user’s task. To achieve this goal however the assistant first has to learn about the user’s task. One approach to this problem is described in [13]. The authors propose to constrain the user’s policies using prior domain knowledge in the form of hierarchical and relational constraints. This knowledge is compiled into an underlying Dynamic Bayesian network and with its help the distribution of user’s goals is inferred given the sequence of their atomic actions. (For a definition and a tutorial on use of Dynamic Bayesian networks for learning see the work of Zoubin Ghahramani [3].) For an alternative way also using Partially Observed Markov Decision Processes see [11]. While decision-theoretic assistance focuses on suggesting or replacing user’s action, the master’s goal in our work is to learn from the player’s experience what kind of tasks fit her the best, which differs significantly from assistance problem and requires a different approach.

A similar problem is considered in a system of interacting agents called teacher and follower(s) [1]. The follower algorithms concentrate on learning how to play best against (typically) a stationary opponent. Follower algorithms (mostly) ignore the payoffs of other agents. Teacher algorithms use the available information about other agents to “teach” them a “good” way of playing the game. The authors introduce the notion of guilt, which numerically encodes the deviations of an agent from the “good” strategy that the teacher attempts to enforce. The teacher utility function depends on the agents’ rewards and their guilt values. Before the learning begins the teacher needs to calculate (or acquire in some other way) the target actions sequence that represents the “good” strategy. The teacher-follower approach concentrates on forcing the follower to modify its strategy, while in our problem the master is required to change the environment to adapt to whatever the player considers a “good” strategy.

A possible approach to solve the Engagement Problem [11] is considered in the work of R. Herbich and T. Graepel [4]. In this case difficulty adjustment happens through creating and matching teams of players in such a way that the skill ratings of resulting teams are approximately the same. The authors propose and evaluate a novel approach to estimate the skill ratings based on the outcomes of multiple games. In this context the engaging environment is created by carefully choosing the “right” opponents of the player.

The Engagement Problem [11] in its most general formulation as a learning problem has not been addressed in the literature so far. In the following, we propose a framework based on Markov Decision Processes for solving it.

3 ENGAGING MDP

Let us first briefly review a Markov Decision Process framework, which forms the basis of our work. A Markov Decision Process (MDP) describes a quite general system of an agent interacting with an environment. The environment is described by a set of states \( S \), while the agent has at its disposal a set of actions \( A \). The interaction is described by two functions: A transfer function \( T : S \times A \rightarrow PD(S) \), where \( PD(S) \) is the set of all discrete probability distributions over the set \( S \), defines the effects of the agent’s actions on the environment, and a reward function \( R : S \rightarrow \mathbb{R} \) defines the effects of the environment’s states on the agent.

The agent and the environment interact continuously over a sequence of time steps. At time \( t \) the agent is in a particular state \( s_t \in S \) and executes an action \( a_t \in A \) based on its knowledge of the environment; the environment presents the agent with the next state \( s_{t+1} \) according to \( T(s_t, a_t) \) and rewards it with \( R(s_t, a_t) \) according to the reward function \( R \). The goal of the agent is to maximize the total amount of reward it receives in the long run. The agent’s actions are controlled by its so called policy \( \pi : S \rightarrow A \). Given the policy \( \pi \) each state acquires a specific value \( V^\pi(s) = E_{\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t)] | s_0 = s \) which is the expected reward the agent will get if it starts from this state and follows the policy \( \pi \). Values indicate the long-term desirability of states under the given policy.

A process described above is called an MDP if it satisfies the Markov property [10]:

\[
Pr\{s_{t+1} = s' | s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\} = Pr\{s_{t+1} = s' | s_t, a_t\}
\]

That means that the environment’s response at \( t + 1 \) should depend only on its state and the actions executed at time step \( t \), and not on the full history of the process so far. Due to the Markov property, there exist efficient techniques for solving MDPs (for the definitions and detailed descriptions see the book of Sutton and Barto [10]).

The MDP framework is abstract and flexible enough to be applied to many different learning problems. Here we show how to adapt it to solve the Engagement Problem [11].

In our particular case we extend it to introduce two agents, who operate in the same environment, but each of them has its own set of actions \( A_i, i = 1, 2 \). Their interaction has an episodical and alternating structure. Alternating in the sense that they do not act simultaneously, but rather there exist two sets of states, \( S_m, S_p \), such that \( S = S_m \cup S_p \) and \( S_m \cap S_p = \emptyset \), and the agents act only when they find themselves in their respective states.

Consider the followig situation (where for the sake of simplicity we omit the rewards):

\[
\begin{align*}
&\ldots, s_{i-1}^m \xrightarrow{a_{i-1}^m} s_i^m \xrightarrow{a_i^m} s_{i+1}^m \xrightarrow{a_{i+1}^m} \ldots \\
&\ldots, s_{i-1}^p \xrightarrow{a_{i-1}^p} s_i^p \xrightarrow{a_i^p} s_{i+1}^p \xrightarrow{a_{i+1}^p} \ldots
\end{align*}
\]

The master is in the state \( s_{i-1}^m \in S_m \). It performs zero or more of his actions \( \{a_m\} \) in such a way as to modify (or not) the difficulty level of the next assignment. This set of actions transforms the state \( s_{i-1}^m \) into the state \( s_i^m \). Next this state becomes observable.
In the ideal case it depends on the reward function of the current environment (note that no “difficulty level” is involved in defining a player’s policy state-value function \( V^{\pi} \)) to calculate the master’s reward function. To construct the reward function \( R^m \) of the master, where the master acquires its own learning curve of the player) to calculate the master’s reward.

4 EXPERIMENTS

The goal of the experiments described in this section was to demonstrate the effectiveness of the framework outlined above. To this aim the experiments were kept as simple as possible. As a sample environment we have chosen a rectangular maze 3 × 3, containing a treasure in one of the cells. The player always starts in the top left corner and attempts to find the treasure. As the test player we took a Q-learning agent [12] with the following parameters:

- Learning rate \( \alpha = 0.1 \);
- Discount factor \( \gamma = 0.9 \);
- Step \( \delta = 1 \);
- Exploration rate \( \epsilon = 0.1 \).

The master in its turn can move the treasure. In the initial setup the master has only two actions: “Put the treasure in the top right corner” and “Put the treasure in the bottom left corner” (Figure 2(a)). In the second experiment we extend this set of actions to allow the master to put the treasure also in the center of the maze (Figure 2(b)).

To complete the master’s MDP we need to construct an approximation of a player’s policy \( \pi' \) and define the master’s reward function \( R^m \). To construct \( \pi' \) the master keeps record of the history of the player’s actions in the current episode: \( H = \{(s_i, a_i)\} \), where \( s_i \in S_p \) is a state encountered by the player during the current episode, and \( a_i \in A_p \) is an action taken by the player in \( s_i \). For every \( s \in S_p \) we define \( A' = \{a \in A_p | (s, a) \in H\} \). Then the policy approximation \( \pi'(s, a) \) is defined as the number of times that \( a_s \) occurs in \( A' \), normalised by the length of \( A' \).
In this paper we addressed the problem of automatic difficulty scaling in a computer game and formalized it as the Engagement Problem, see Definition[1,7]. We also presented the Engaging MDP framework. Its advantage is that it provides a clear mathematical foundation for solving the Engagement problem. The results of our initial experiments demonstrate that the Engaging MDP framework can be used with success to produce an interesting behaviour. Since this is work in progress, there are indeed more experiments that need to be performed and more questions that need to be researched. Among them are the following:

How suitable is an MDP for solving the Engagement Problem? Involving people in any environment automatically turns it into a partially observable one. In the context of adjusting a computer game to a particular player there exist some parameters that are hidden from the master. One of them is a skill level of the player: Even the player herself can usually make only an educated guess on how well she will perform. Furthermore, above we assumed that the master observes not only the states and actions of the player, but also its reward function $R_p$. This assumption is unrealistic when dealing with people and even artificial agents, since in the ideal case $R_p$ estimates the player’s enjoyment of the current task, and as such is unknown to anyone but the player (and even the player most probably is unable to define it precisely). Taking this into account we should formulate the master’s task as a Partially Observable MDP (for a definition and details see [5]). Nevertheless our model of Engaging MDPs can be viewed as the approximation technique for solving the Engagement Problem[7,7).

How to construct the master’s reward function? In order for Engaging MDP to be a good approximation, another question that needs further research is the specific function used in the master’s reward and its influence on the master’s performance. Recall that we constrained the master’s problem by assuming that it knows the player’s reward function $R_p$. This assumption is unrealistic when dealing with people or even artificial agents, since in the ideal case $R_p$ estimates the player’s enjoyment of the current task, and as such is unknown to anyone but the player (and even the player most probably is unable to define it precisely). Taking this into account we should formulate the master’s task as a Partially Observable MDP (for a definition and details see [5]). Nevertheless our model of Engaging MDPs can be viewed as the approximation technique for solving the Engagement Problem[7,7).

How realistic is the episodic setting? When formulating Engaging MDP framework we assumed an episodic setting for the master-player interaction. This assumption is also unrealistic, because in the ideal case the master’s presence should be inobtrusive and its learning should be online. Nevertheless, the episodic setting allows for an approximation of the ideal situation by letting the master to gather experience during (artificially created, i.e. the player does not need to know about them) episodes. In future work we intend to research the question of incorporating online learning in our framework.

The authors hope that this paper will inspire and motivate researchers in the AI and games but also in the reinforcement learning communities to join investigating the exciting new research question of engaging agents.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful and insightful comments. Olana Missura was funded by the MPG-FhG Machine Learning Collaboration (http://lif.fml.tuebingen.mpg.de/) and Kristian Kersting by the Fraunhofer ATTRACT fellowship STREAM.

REFERENCES