Stephan Schiffel
Michael Thielscher
Julian Togelius (Eds.)

The IJCAI-15 Workshop on General Game Playing

General Intelligence in Game-Playing Agents, GIGA'15

Buenos Aires, Argentina, July 2015
Proceedings

Preface

Artificial Intelligence (AI) researchers have for decades worked on building game-playing agents capable of matching wits with the strongest humans in the world, resulting in several success stories for board games like chess and checkers and computer games such as StarCraft, Pac-Man and Unreal Tournament. The success of such systems has been partly due to years of relentless knowledge-engineering effort on behalf of the program developers, manually adding application-dependent knowledge to their game-playing agents. The various algorithmic enhancements used are often highly tailored towards the game at hand.

Research into general game playing (GGP) aims at taking this approach to the next level: to build intelligent software agents that can, given the rules of any game, automatically learn a strategy for playing that game at an expert level without any human intervention. In contrast to software systems designed to play one specific game, systems capable of playing arbitrary unseen games cannot be provided with game-specific domain knowledge a priori. Instead, they must be endowed with high-level abilities to learn strategies and perform abstract reasoning. Successful realization of such programs poses many interesting research challenges for a wide variety of artificial-intelligence sub-areas including (but not limited to):

- applications
- computational creativity
- computational game theory
- evaluation and analysis
- game design
- imperfect-information games
- knowledge representation
- machine learning
- multi-agent systems
- opponent modeling
- planning
- reasoning
- search

These are the proceedings of GIGA'15, the first ever fourth workshop on General Intelligence in Game-Playing Agents, following the inaugural GIGA Workshop at IJCAI'09 in Pasadena (USA) and the follow-up events at IJCAI'11 in Barcelona (Spain) and IJCAI'13 in Beijing (China). This workshop series has been established to become the major forum for discussing, presenting and promoting research on General Game Playing. It is intended to bring together researchers from the above sub-fields of AI to discuss how best to address the challenges and further advance the state-of-the-art of general game-playing systems and generic artificial intelligence.

These proceedings contain the 10 papers that have been selected for presentation at this workshop. All submissions were reviewed by a distinguished international program committee. The accepted papers cover a multitude of topics such as general video game playing, advanced simulation-based methods, heuristics, parallelization, and learning.

We thank all the authors for responding to the call for papers with their high quality submissions, and the program committee members and other reviewers for their valuable feedback and comments. We also thank IJCAI for all their help and support.

We welcome all our delegates and hope that all will enjoy the workshop and through it find inspiration for continuing their work on the many facets of General Game Playing!

July 2015

Stephan Schiffel
Michael Thielscher
Julian Togelius

Organization

Workshop Chairs

Stephan Schiffel, Reykjavík University, Iceland
Michael Thielscher, University of New South Wales, Australia
Julian Togelius, New York University, USA

Program Committee

Yngvi	Björnsson	Reykjavík University
Tristan	Cazenave	Université Paris-Dauphine
Michael	Genesereth	Stanford University
Lukasz	Kaiser	Université Paris Diderot
Simon	Lucas	University of Essex
Jacek	Mańdziuk	Warsaw University of Technology
Diego	Perez	University of Essex
Ji	Ruan	Auckland University of Technology
Abdallah	Saffidine	University of New South Wales
Spyridon	Samothrakis	University of Essex
Tom	Schaul	New Yourk University
Sam	Schreiber	Google Inc.
Nathan	Sturtevant	University of Denver
Mark	Winands	Maastricht University

Additional Reviewers

Sumedh Ghaisas
Chiara Sironi
Maciej Świechowski

Table of Contents

On the Cross-Domain Reusability of Neural Modules for General Video Game Playing

Alex Braylan **Mark Hollenbeck** **Elliot Meyerson** **Risto Miikkulainen**
{braylan,mhollen,ekm,risto}@cs.utexas.edu
The University of Texas at Austin

Abstract

We consider a general approach to knowledge transfer in which an agent learning with a neural network adapts how it reuses existing networks as it learns in a new domain. Networks trained for a new domain are able to improve performance by selectively routing activation through previously learned neural structure, regardless of how or for what it was learned. We present a neuroevolution implementation of the approach with application to reinforcement learning domains. This approach is more general than previous approaches to transfer for reinforcement learning. It is domain-agnostic and requires no prior assumptions about the nature of task relatedness or mappings. We analyze the method's performance and applicability in high-dimensional Atari 2600 general video game playing.

1 Introduction

The ability to generally apply any and all available previously learned knowledge to new tasks is a hallmark of general intelligence. *Transfer learning* is the process of reusing knowledge from previously learned *source* tasks to bootstrap learning of *target* tasks. For reinforcement learning (RL) agents, transfer is particularly important, as previous experience can help to efficiently explore new environments. Knowledge acquired during previous tasks also contains information about the agent's task-independent decision making and learning dynamics, and thus can be useful even if the tasks seem completely unrelated.

Existing approaches to transfer learning for reinforcement learning have successfully demonstrated transfer of varying kinds of knowledge [Taylor and Stone, 2009], but they tend to make two fundamental assumptions that restrict their generality: (1) some sort of a priori human-defined understanding of how tasks are related, (2) separability of knowledge extraction and target learning. The first assumption limits the applicability of the approach by restricting its use only to cases where the agent has been provided with this additional relational knowledge, or, if it can be learned, cases where task mappings are useful. The second assumption implies further expectations about what knowledge will be useful and how it should be incorporated *before* learning on the target task begins, preventing the agent from adapting the way it uses source knowledge as it gains information about the target domain.

We consider General Reuse Of Modules (GReuseOM), a general neural network approach to transfer learning that avoids both of these assumptions, by augmenting the learning process to allow learning networks to selectively route through existing neural modules (source networks) as they simultaneously develop new structure for the target task. Unlike previous work, which has dealt with mapping task variables between source and target, GReuseOM is task-independent, in that no knowledge about the structure of the source task or even knowledge about where the network came from is required for it to be reused. Instead of using mappings between task-spaces to facilitate transfer, it searches directly for mappings in the solution space, that is, connections between existing source networks and the target network. GReuseOM is motivated by studies that have shown in both naturally occurring complex networks [Milo *et al.*, 2002] and artificial neural networks [Swarup and Ray, 2006] that certain network structures repeat and can be useful across domains, without any context for how exactly this structure should be used. We are further motivated by the idea that neural resources in the human brain are reused for countless purposes in varying complex ways [Anderson, 2010].

In this paper, we present an implementation of GReuseOM based on the Enforced Subpopulations (ESP) neuroevolution framework [Gomez and Miikkulainen, 1997]. We validate our approach first in a simple boolean logic domain, then scale up to the Atari 2600 general game playing domain. In both domains, we find that GReuseOM-ESP improves learning overall, and tends to be most useful when source and target networks are more complex. In the Atari domain, we show that the effectiveness of transfer coincides with an intuitive high-level understanding of game dynamics. This demonstrates that even without traditional transfer learning assumptions, successful knowledge transfer via general reuse of existing neural modules is possible and useful for RL. In principle, our approach and implementation naturally scale to transfer from an arbitrary number of source tasks, which points towards a future class of GReuseOM agents that accumulate and reuse knowledge throughout their lifetimes across a variety of diverse domains.

The remainder of this paper is organized as follows: Sec-

tion 2 provides background on transfer learning and related work, Section 3 describes our approach in detail, Section 4 analyzes results from experiments we have run with this approach, and Section 5 discusses the implications of these results and motivations for future work.

2 Background

Transfer learning encompasses machine learning techniques that involve reusing knowledge across different domains and tasks. In this section we review existing transfer learning methodologies and discuss their advantages and shortcomings to motivate our approach. We take the following two definitions from [Sinno and Yang, 2010]. A *domain* is an environment in which learning takes place, characterized by the input and output space. A *task* is a particular function from input to output to be learned. In sequential-decision domains, a task is characterized by the values of sensory-action sequences corresponding to the pursuit of a given goal. A taxonomy of types of knowledge that may be transferred are also enumerated in [Sinno and Yang, 2010]. As our approach reuses the structure of existing neural networks, it falls under 'feature representation transfer'.

2.1 Transfer Learning for RL

Reinforcement learning (RL) domains are often formulated as Markov decision processes in which the state space comprises all possible observations, and the probability of an observation depends on the previous observation and an action taken by a learning agent. However, many real world RL domains are non-Markovian, including many Atari 2600 games.

Five dimensions for characterizing the generality and autonomy of algorithms for transfer learning in RL are given in [Taylor and Stone, 2009]: (1) restrictions on how source and target task can differ; (2) whether or not *mappings* between source and target state and action variables are available to assist transfer; (3) the form of the knowledge transferred; (4) restrictions on what classes of learning algorithms can be used in the source and/or target tasks; (5) whether or not the algorithm autonomously selects which sources to reuse.

Some of the most general existing approaches to transfer for RL automatically learn task mappings, so they need not be provided beforehand, e.g., [Taylor *et al.*, 2007; 2008; Talvitie and Singh, 2007]. These approaches are general enough to apply to any reinforcement learning task, but as the state and action spaces become large they become intractable due to combinatorial blowup in the number of possible mappings. These approaches also rely on the assumption that knowledge for transfer can be extracted based on mappings between state and action variables, which may miss useful internal structure these mappings cannot capture.

2.2 General Neural Structure Transfer

There are existing algorithms similar to our approach in that they enable general reuse of existing neural structure. They can apply to a wide range of domains and tasks in that they automatically select source knowledge and avoid inter-task mappings. Knowledge-Based Cascade Correlation [Shultz and Rivest, 2001] uses a technique based on cascade correlation to build increasingly complex networks by inserting source networks chosen by how much they reduce error. Knowledge Based Cascade Correlation is restricted in that it is only designed for supervised learning, as the source selection depends heavily on an immediate error calculation. Also, connectivity between source and target networks is limited to the input and output layer of the source. Subgraph Mining with Structured Representations [Swarup and Ray, 2006] creates sparse networks out of primitives, or commonly used sub-networks, mined from a library of source networks. The subgraph mining approach depends on a computationally expensive graph mining algorithm, and it tends to favor exploitation over innovation and small primitives rather than larger networks as sources.

Our approach is more general in that it can be applied to unsupervised and reinforcement learning tasks, and makes fewer a priori assumptions about what kind of sources and mappings should work best. Although we only consider an evolutionary approach in this paper, GReuseOM should be extensible to any neural network-based learning algorithm.

3 Approach

This section introduces the general idea behind GReuseOM, then provides an overview of the ESP neuroevolution framework, before describing our particular implementation: GReuseOM-ESP.

3.1 General Reuse Of Modules (GReuseOM)

The underlying idea is that an agent learning a neural network for a target task can selectively reuse knowledge from existing neural modules (source networks) while simultaneously developing new structure unique to a target task. This attempts to balance reuse and innovation in an integrated architecture. Both source networks and new hidden nodes are termed *recruits*. Recruits are added to the target network during the learning process. Recruits are adaptively incorporated into the target network as it learns connection parameters from the target to the recruit and from the recruit to the target. All internal structure of source networks is *frozen* to allow learning of connection parameters to remain consistent across recruits. This forces the target network to transfer learned knowledge, rather than simply overwrite it. Connections to and from source networks can, in the most general case, connect to any nodes in the source and target, minimizing assumptions about what knowledge will be useful.

A GReuseOM network or *reuse network* is a 3-tuple $\mathcal{G} = (M, S, T)$ where M is a traditional neural network (feedforward or recurrent) containing the new nodes and connections unique to the target task, with input and output nodes corresponding to inputs and outputs defined by the target domain; S is a (possibly empty) set of pointers to recruited source networks $\mathcal{S}_1, ..., \mathcal{S}_k$; and T is a set of weighted *transfer connections* between nodes in M and nodes in source networks, that is, for any connection $((u, v), w) \in T$, $(u \in M \wedge v \in \mathcal{S}_i) \vee (u \in \mathcal{S}_i \wedge v \in M)$ for some $0 \leq i \leq k$. This construction strictly extends traditional neural networks so that each $\mathcal{S}_i$ can be a traditional neural network or a reuse network of its own. When $\mathcal{G}$ is evaluated, we evaluate only the network

induced by directed paths from inputs of M to outputs of M, including those which pass through some $\mathcal{S}_i$ via connections in T. Before each evaluation of $\mathcal{G}$, all recruited source network inputs are set to 0, since at any given time the agent is focused only on performing the current target task. The parameters to be learned are the usual parameters of M, along with the contents of S and T. The internal parameters of each $\mathcal{S}_i$ are frozen in that they cannot be rewritten through $\mathcal{G}$.

The motivation for this architecture is that if the solution to a source task contains *any* information relevant to solving a target task, then the neural network constructed for the source task will contain *some* structure (subnetwork or module) that will be useful for a target network. This has been shown to be true in naturally occurring complex networks [Anderson, 2010], as well as cross-domain artificial neural networks [Swarup and Ray, 2006]. Unlike in the subgraph mining approach to neural structure transfer [Swarup and Ray, 2006], this general formalism makes no assumptions as to what subnetworks actually will be useful. One perspective that can be taken with this approach is that a lifelong learning agent maintains a system of interconnected neural modules that it can potentially reuse at any time for a new task. Even if existing modules are unlabeled, they may still be useful, simply due to the fact that they contain knowledge of how the agent can successfully learn. Furthermore, recent advances in reservoir computing [Lukoševičius and Jaeger, 2009] have demonstrated the power of using large amounts of frozen neural structure to facilitate learning of complex and chaotic tasks.

The above formalism is general enough to allow for an arbitrary number of source networks and arbitrary connectivity between source and target. In this paper, to validate the approach and simplify analysis, we use at most one source network and only allow connections from target input to source hidden layer and source output layer to target output. This is sufficient to show that the implementation can successfully reuse hidden source features, and analyze the cases in which transfer is most useful. Future refinements are discussed in Section 5. The current implementation, described below, is a neuroevolution approach based on ESP.

3.2 Enforced Subpopulations (ESP)

Enforced Sub-Populations (ESP) [Gomez and Miikkulainen, 1997] is a neuroevolution technique in which different elements of a neural network are evolved in separate *subpopulations* rather than evolving the whole network in a single population. ESP has been shown to perform well an a variety of reinforcement learning tasks, e.g., [Gomez and Miikkulainen, 1997; 2003; Gomez and Schmidhuber, 2005; Miikkulainen *et al.*, 2012; Schmidhuber *et al.*, 2007]. In standard ESP, each hidden neuron is evolved in its own subpopulation. Recombination occurs only between members of the same subpopulation, and mutants in a subpopulation derive only from members of that subpopulation. The genome of each individual in a subpopulation is a vector of weights corresponding to the weights of connections from and to that neuron, including node bias. In each generation, networks to be evaluated are randomly constructed by inserting one neuron from each subpopulation. Each individual that participated in the network receives the fitness achieved by that network.

When fitness converges, i.e., does not improve over several consecutive generations, ESP makes use of *burst phases*. In initial burst phases each subpopulation is repopulated by mutations of the single best neuron ever occuring in that subpopulation, so that it reverts to searching a δ-neighborhood around the best solution found so far. If a second consecutive burst phase is reached, i.e., no improvements were made since the previous burst phase, a new neuron with a new subpopulation may be added [Gomez, 2003].

3.3 GReuseOM-ESP

We extend the idea of enforced sub-populations to transfer learning via GReuseOM networks. For each reused source network $\mathcal{S}_i$ the transfer connections in T between $\mathcal{S}_i$ and M evolve in a distinct subpopulation. At the same time new hidden nodes can be added to M and evolve within their own subpopulations in the manner of standard ESP. In this way, the integrated evolutionary process simultaneously searches the space for how to reuse each potential source network and how to innovate with each new node. Specifically, the GReuseOM-ESP architecture (Figure 1) is composed of the following elements:

- A pool of potential source networks. In the experiments in this paper, each target network reuses at most one source at a time.
- *Transfer genomes* defining a list of transfer connections between the source and target networks. Each potential source network in the pool has its own subpopulation for evolving transfer genomes between it and the target network. Each connection in T is contained in some transfer genome. In our experiments, the transfer connections included are those such that the target's inputs are fully connected to the source's hidden layer, and the source's outputs are fully connected into the target's outputs. Therefore, the transfer genome only encodes the weights of these cross-network connections.
- A burst mechanism that determines when innovation is necessary based on a recent history of performance improvement. New hidden recruits (source networks or new single nodes) added during the burst phase evolve within their own subpopulations in the manner of classic ESP.

All hidden and output neurons use a hyperbolic tangent activation function. Networks include a single hidden layer, and can include self loops on hidden nodes; they are otherwise feedforward. The particulars of the genetic algorithm in our implementation used to evolve each subpopulation mirror those described in [Gomez, 2003]. This algorithm has been shown to work well within the ESP framework, though any evolutionary algorithm could potentially be substituted in its place.

4 Experiments

We evaluate GReuseOM-ESP on two domains: a simple n-bit parity domain mirroring that used to evaluate knowledge

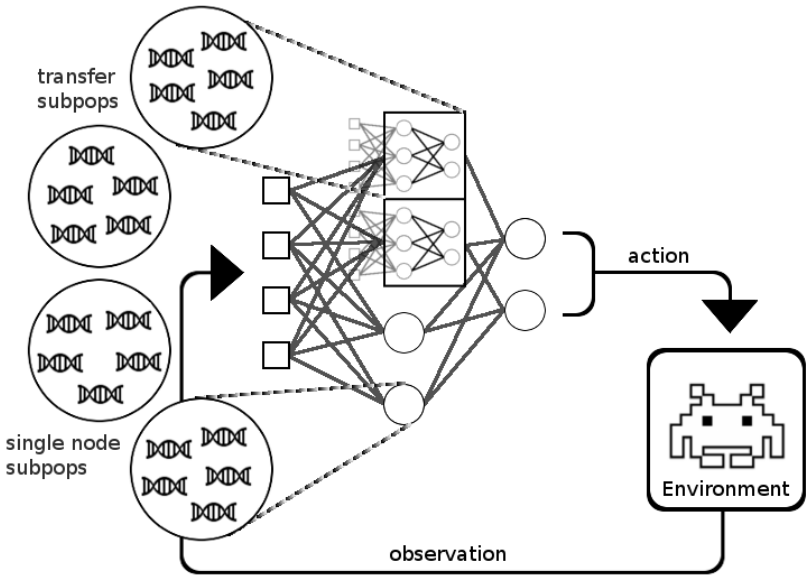


Figure 1: The GReuseOM-ESP architecture. Reused subnetworks of sources are boxed. Edges between input and source and between source and output denote full connectivity between these layers. The genome in each subpopulation encodes weight information for the connections from and to the corresponding recruit.

transfer in [Swarup and Ray, 2006], and the more complex Atari 2600 video game playing domain. In both domains, we first train *scratch* networks that do not reuse existing networks, that is, S is the empty set. We then reuse each scratch network in training GReuseOM networks for different tasks. We compare performance between scratch and transfer, and between source-target setups. Results demonstrate the ability of GReusOM-ESP to selectively reuse source structure.

4.1 N-Bit Parity

GReuseOM-ESP was initially evaluated under the boolean logic domain using N-bit parity. The N-bit parity problem has a long-standing history serving as a benchmark for basic neural network performance. The N-bit parity function is the mapping defined on N-bit binary vectors that returns 1 if the sum of the N binary values in the vector is odd, and 0 otherwise. This function is deceptively difficult for neural networks to learn since a change in any single input bit will alter the output. Although N-bit parity is not fully *cross-domain* in the stronger sense for which our approach applies, the input feature space does differ as N differs, and it is useful for validation of the approach and connection with previous work.

Performance is measured in number of generations to find a network that solves N-bit parity within $\epsilon = 0.1$ mean squared error. In this experiment, networks were trained from scratch with ESP for $N = [2, 3, 4]$. Then, each of these networks was used as a source network for each N-bit parity target domain with $N = [3, 4, 5]$. ESP, without transfer, was used as a control condition for each target task. A total of 10 trials were completed for each condition.

In this experiment, transfer learning was able to outperform learning from scratch for all three target tasks when using some source task. For 3-bit and 4-bit parity, transfer learning always outperformed learning from scratch for all three possible sources. For the more complex 5-bit parity target task, transfer from the 4-bit network outperformed learning from scratch, while transfer from the simpler tasks did not. This may be due to the significantly greater complexity required for 5-bit parity over 2- or 3-bit parity. The limited frozen structure may become a burden to innovation after the initial stages of evolution. The more complex 4-bit parity networks have more structure to select from, and thus may assist in innovation over a longer time frame.

	Source	None	2-bit par	3-bit	4-bit
Target	3-bit par	309.5	202	167.5	**158**
	4-bit par	339	**192.5**	308	311
	5-bit par	626	780	720.5	**542**

Table 1: Median number of generations for task completion for all N-bit parity source-target setups.

4.2 Atari 2600 Game Playing

Our next experiment evaluated game playing performance in the Atari 2600 game platform using the Arcade Learning Environment (ALE) simulator [Bellemare *et al.*, 2013]. This domain is particularly popular for evaluating RL techniques, as it exhibits sufficient complexity to challenge modern approaches, contains non-markovian properties, and entertained a generation of human video game players. We used GReuseOM-ESP to train agents to play eight games (Asterix, Bowling, Boxing, Breakout, Freeway, Pong, Space Invaders, and Seaquest) both from scratch and using transferred knowledge from existing game-playing source networks. Neuroevolution techniques are quite competitive in the Atari 2600 domain [Hausknecht *et al.*, 2013], and ESP in particular has

yielded state-of-the-art performance for several games [Braylan *et al.*, 2015].

Each source network was trained from scratch on a game using standard ESP (GReuseOM-ESP with an empty reuse set). Each source network was then used by a target network for an evolutionary run for each other game. Each run lasted 200 generations with 100 evaluations per generation. Each individual i achieves some score $i(g)$ in its game g. Let $min(g)$ be the min over all max scores achieved in a single generation by any run of g. Let the *fitness* of i be $i(g) - min(g)$. This ensures that fitness is always positive (in both boxing and pong, raw scores can be negative). The fitness of an evolutionary run at a given generation is the highest fitness achieved by an individual by that generation.

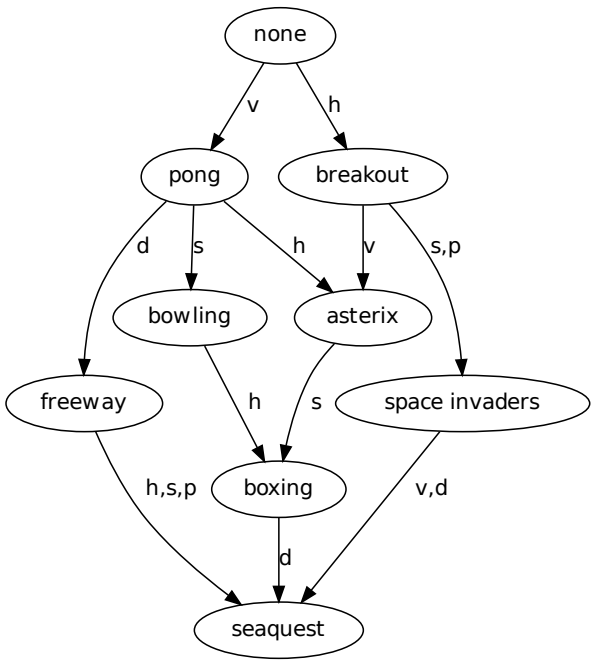
We ran a total of 176 trials split across all possible setups: training using each other game as a source, and training from scratch. We use the ϵ-repeat action approach as suggested in [Hausknecht and Stone, 2015] to make the environment stochastic in order to disable the algorithm from finding loopholes in the deterministic nature of the simulator. We use the recommended $\epsilon = 0.25$ [1]. Parameters were selected based on their success with standard ESP.

To interface with ALE, the output layer of each network consists of a 3x3 substrate representing the 9 directional movements of the Atari joystick in addition to a single node representing the Fire button. The input layer consists of a series of object representations manually generated as previously described in [Hausknecht *et al.*, 2013], where the location of each object on the screen is represented in an 8x10 input substrate corresponding to the object's class. The number of object classes for the games used in our experiments vary between one and four. Although object representations are used in these experiments, pixel-level vision could also be learned from scratch below the neuroevolution process, e.g., via convolutional networks, as in [Koutník *et al.*, 2014].

Domain Characterization

Each game can be characterized by generic binary features that determine the requirements for successful game play, in order to place the games within a unified framework. We use binary features based on the existence of the following: (1) horizontal movement (joystick left/right), (2) vertical movement (joystick up/down), (3) shooting (fire button); (4) delayed rewards; and (5) the requirement of long-term planning. Intuitively, more complex games will possess more of these qualities. The partial ordering of games by complexity defined by these features is shown in Figure 2. The assignment of features (1), (2) and (3) is completely defined based on game interface [Bellemare *et al.*, 2013]. Freeway and Seaquest are said to have *delayed rewards* because a high score can only be achieved by long sequences of rewardless behavior. Only Space Invaders and Seaquest were deemed to require long-term planning [Mnih *et al.*, 2013], since the long-range dynamics of these games penalize reflexive strategies, and as such, agents in these games can perform well with a low decision-making frequency [Braylan *et al.*, 2015]. Aside from their intuitiveness, these features are validated below based on their ability to characterize games by complexity and predict transferibility. For a simple metric of complexity, let $cmplx(g)$ be the number of the above features game g exhibits.

[1] https://github.com/mgbellemare/Arcade-Learning-Environment/tree/dev



	vert	horiz	shoot	delay	plan
pong	■				
breakout		■			
asterix	■	■			
bowling	■		■		
freeway	■			■	
boxing	■	■	■		
space invaders		■	■		■
seaquest	■	■	■	■	■

Figure 2: Vector of features for each game (indicated in black) and lattice of games ordered by features. Every path from *none* to g contains along its edges each complexity feature of g a exactly once. Features: v = vertical movement, h = horizontal movement, s = shooting, d = delayed reward, p = long-term planning.

Atari 2600 Results

There are many possible approaches to evaluating success of transfer [Taylor and Stone, 2009]. For comparing performance *across* games, we focus on *time to threshold*. To minimize threshold bias, for each game we chose the threshold to be the min of the max fitness achieved across all trials. Given this threshold, the average time to threshold in terms of generations may be vastly different, depending on the average learning curve of each game. These learning curves are quite irregular, as illustrated in Figure 4. For each game we measure time in terms of percent of average time to threshold, and the *success rate* is the proportion of trials that have achieved the threshold by that time.

Figure 3 plots success over time for different groups of trials. The leftmost plot compares the success rate of all transfer

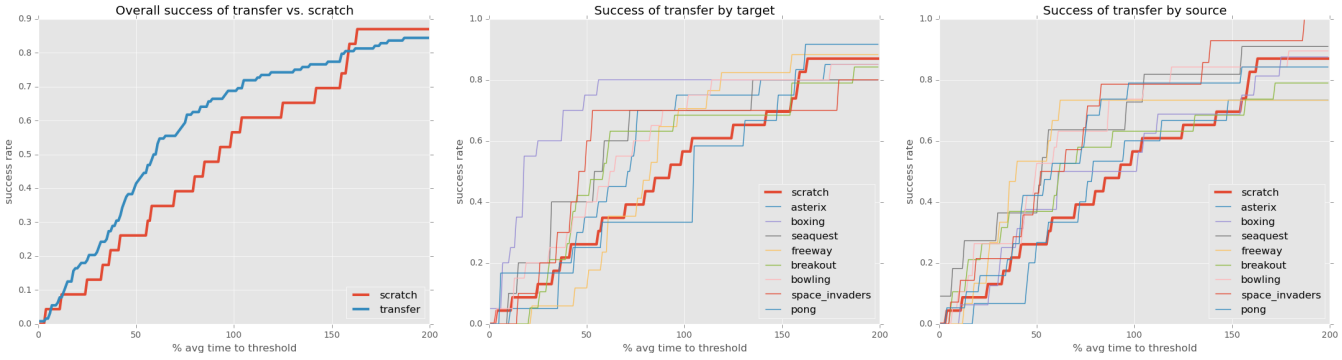


Figure 3: Success rate (proportion of trials that have reached the target threshold) by percent of average number of generations to threshold of target game with trials (allowing comparisons across games with different average times to threshold) grouped by (1) scratch vs. transfer, (2) target game, (3) source game.

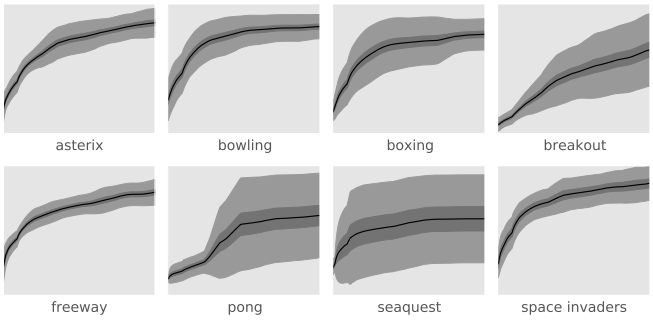


Figure 4: Distributions of fitness for each game by generation over all trials. Mean (black), standard error (dark gray) and standard deviation (light gray) are shown at each generation.

game g	cmplx(g)	deg(g)	deg$^-(g)$	deg$^+(g)$
seaquest	5	8	4	4
space invaders	3	7	4	3
boxing	3	6	4	2
bowling	2	5	3	2
asterix	2	5	2	3
freeway	2	4	2	2
pong	1	4	1	3
breakout	1	1	0	1

Table 2: A total ordering of games by complexity score and degree (total, in (-), and out (+)) in the transferability digraph with edge cutoff 0.5 (Figure 5(a)).

trials to scratch trials. It shows what we would expect from transfer overall: networks that reuse frozen structure from previous games are able to take advantage of that structure to bootstrap learning. This works initially, but eventually scratch catches up, as it becomes more difficult to innovate with a single frozen structure. When trials are grouped by target (middle pane), we can see that some games are better targets for transfer than others. As demonstrated in Figure 3, more complex games (with respect to our game features) are generally better targets than less complex. It is less clear what we can draw from grouping trials by source (right pane). There is a tighter spread than with targets, though there may still be a tendency towards more complex games being better sources. This may be counter-intuitive, as we might expect simpler games to be easier to reuse. However, more complex games have networks with more complex structure from which a target network can, through the evolutionary process, select some useful subnetwork that fits its needs. Similarly, a complex domain will be more likely to be a good target, since it requires a wider variety of structure to be successful, so sources have a higher chance of satisfying *some* of that requirement.

For comparing performance *within* a target game, we need not resort to threshold normalization, and can instead focus on raw max fitness. For reference, average and best fitness for both transfer and scratch are given in Table 3. Note that previously published approaches to Atari game-playing use fully deterministic environments, making direct score comparisons difficult (see [Braylan *et al.*, 2015] for a comparison of ESP to other approaches in deterministic environments).

The *transfer effectiveness* of a source-target setup is the log ratio between its average max fitness and the average max fitness of that game from scratch. The digraphs in Figure 5 each contain the directed edge from g_1 to g_2 only when transfer effectiveness is above a specified threshold. These graphs indicate that the more complex games serve a more useful role in transfer than less complex. Consider the total ordering of games by $cmplx(g)$ given in Table 2. This ordering corresponds exactly to that induced by the degree sequence (by both total degree and in-degree) [Diestel, 2005] of the graph with edge cutoff 0.5. However, for out-degree, the correlation with respect to the ordering is less clear. This reflects Figure 3, in which there is more spread in success when grouped by target (in-degree) vs. source (out-degree).

We see that we can predict the transfer effectiveness by the feature characterizations we provided. The feature characterizations allow us to consider all trials in the same feature space. A linear regression model trained on a random half of the setups yielded weight coefficients for the source and target features that successfully predicted the transfer effectiveness of setups in the test set (Figure 6). The slope was found to be statistically significant with a p-value of 0.01. The most

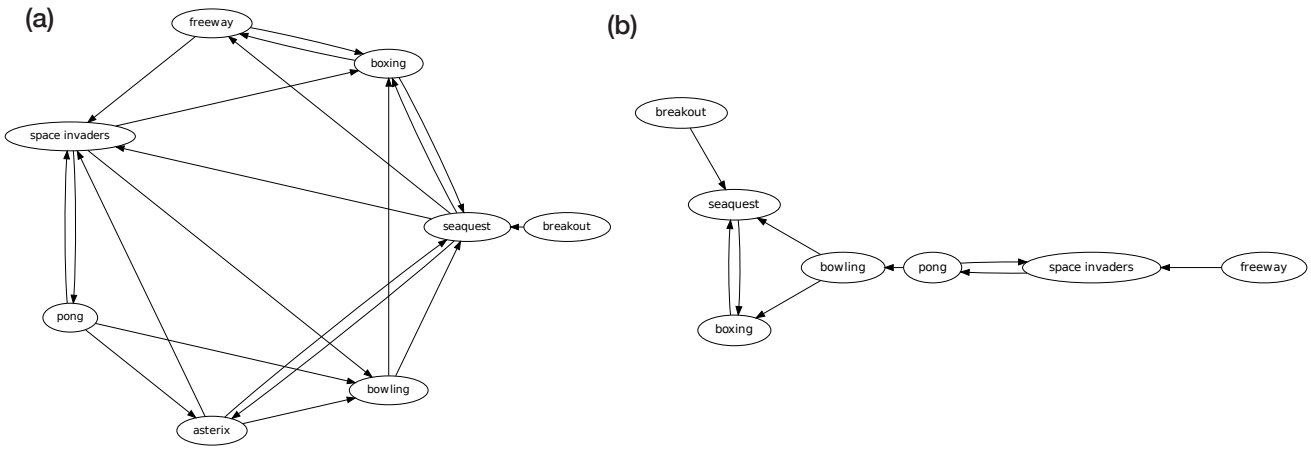


Figure 5: Transferability graphs illustrating the most successful source-target pairs. Each graph includes a directed edge from g_1 to $g_2 \iff$ the transfer effectiveness (defined above) for g_2 reusing g_1 is greater than (a) 0.5, and (b) 1.0, respectively.

game	$min(g)$	best$_t$	best$_s$	avg$_t$	avg$_s$
seaquest	160	1510	300	475.0	262.0
space invaders	310	1520	1320	1076.0	1160.0
boxing	-12	111	107	98.6	104.1
bowling	30	237	231	219.9	201.9
asterix	650	3030	2150	1989.0	2016.7
freeway	21	13	11	10.7	10.7
pong	-21	42	42	21.8	20.3
breakout	0	51	37	25.4	31.3

Table 3: For both transfer (t) and scratch (s) runs, average fitness and best fitness of GReusOM-ESP.

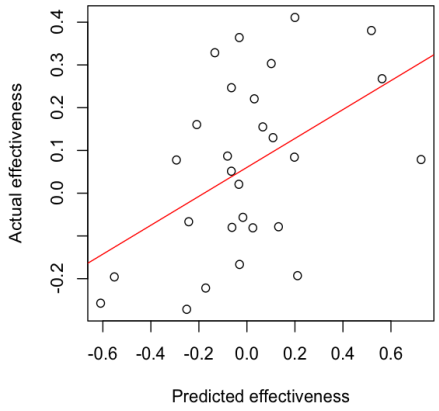


Figure 6: Feature-based linear prediction versus actual transfer effectiveness on out-of-sample setups

significant features were vertical movement and long-term planning in the source domain, with respective coefficients of 0.73 and 0.89. The ability to use the game features to predict transfer effectiveness can be used to inform source selection. It is also encouraging that the effectiveness of transfer with GReuseOM-ESP correlates with a high-level intuition of inter-game dynamics.

5 Discussion

Our results show that GReuseOM-ESP, an evolutionary algorithm for general transfer of neural network structure, can improve learning in both boolean logic and Atari game playing by reusing previously developed knowledge. However, we find that the improvement in learning performance in the target domain depends heavily on the source network. Some source-target pairs do not consistently outperform training from scratch, indicating negative transfer from that source. This highlights the importance of source selection in transfer learning.

Specifically with the Atari game playing domain, we observe an issue of source knowledge *quality*. Some of the source networks that were trained from scratch do relatively well on games whereas others do not. One problem is that the measure of knowledge in source networks is ill-defined. As alluded to in [Taylor *et al.*, 2007], there could be an optimal point in a source's training at which to transfer knowledge to a target, after which the source network has encoded knowledge too specific to its own task, which does not generalize as well to other tasks, and makes useful knowledge difficult to extract. Future analysis will investigate topological regularities of source networks and transfer connections, to further address what and how knowledge is successfully reused.

Another future area of work will involve increasing the flexibility in the combined architecture by 1) relaxing the requirement for all transfer connections to be input-to-hidden and output-to-output, and 2) allowing deeper architectures for the source and target networks. This will promote reuse of subnetworks of varying depth and flexible positioning of modules. However, as networks become large and plentiful, maintaining full connectivity between layers will become intractable, and enforcing sparsity will be necessary.

Having shown that our algorithm works with certain target-source pairs, a next step will involve pooling multiple candidate sources and testing GReuseOM-ESP's ability to ex-

ploit the most useful ones. GReuseOM-ESP extends naturally to learning transfer connections for multiple sources simultaneously. By starting with limited connectivity and adding connections to sources that show promise (while removing connections from ones that are not helping), adaptive multi-source selection may be integrated into the evolutionary process. Methods for adapting this connectivity online have yet to be developed.

Although our initial experiments only scratched the surface, they are encouraging in that they show general transfer of neural structure is possible and useful. They have also helped us characterize the conditions under which transfer may be useful. It will be interesting to investigate whether the same principles extend to other general video game playing domains, such as [Perez *et al.*, 2015; Schaul, 2013]. This should help us better understand how subsymbolic knowledge can be recycled indefinitely across diverse domains.

6 Conclusion

We consider a framework for general transfer learning using neural networks. This approach minimizes a priori assumptions of task relatedness and enables a flexible approach to adaptive learning across many domains. In both the Atari 2600 and N-bit parity domains, we show that a specific implementation, GReuseOM-ESP is able to successfully boost learning by reusing neural structure across disparate tasks. The success of transfer is shown to correlate with intuitive notions of task dynamics and complexity. Our results indicate that general neural reuse – a staple of biological systems – can effectively assist agents in increasingly complex environments.

Acknowledgments This research was supported in part by NSF grant DBI-0939454, NIH grant R01-GM105042, and an NPSC fellowship sponsored by NSA.

References

[Anderson, 2010] M. L. Anderson. Neural reuse: A fundamental organizational principle of the brain. *Behavioral and Brain Sciences*, 33:245–266, 2010.

[Bellemare *et al.*, 2013] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *JAIR*, 47:253–279, 2013.

[Braylan *et al.*, 2015] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *AAAI Workshop on Learning for General Competency in Video Games*, 2015.

[Diestel, 2005] R. Diestel. *Graph Theory*. Springer, 2005. p. 278.

[Gomez and Miikkulainen, 1997] F. J. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, (5):317–342, 1997.

[Gomez and Miikkulainen, 2003] F. J. Gomez and R. Miikkulainen. Active guidance for a finless rocket using neuroevolution. In *Proc. of GECCO '03*, pages 2084–2095, 2003.

[Gomez and Schmidhuber, 2005] F. J. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory pomdps. In *Proc. of GECCO '05*, pages 491–498, 2005.

[Gomez, 2003] F. J. Gomez. Robust non-linear control through neuroevolution. Technical report, UT Austin, 2003.

[Hausknecht and Stone, 2015] M. Hausknecht and P. Stone. The impact of determinism on learning atari 2600 games. In *AAAI Workshop on Learning for General Competency in Video Games*, 2015.

[Hausknecht *et al.*, 2013] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general atari game playing. In *Comp. Intel. and AI in Games*, 2013.

[Koutník *et al.*, 2014] J. Koutník, J. Schmidhuber, and F. J. Gomez. Online evolution of deep convolutional network for vision-based reinforcement learning. In *From Animals to Animats 13*. 2014.

[Lukoševičius and Jaeger, 2009] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[Miikkulainen *et al.*, 2012] R. Miikkulainen, E. Feasley, L. Johnson, I. Karpov, P. Rajagopalan, A. Rawal, and W. Tansey. Multiagent learning through neuroevolution. In *Advances in Computational Intelligence*, pages 24–46. 2012.

[Milo *et al.*, 2002] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[Mnih *et al.*, 2013] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv:1312.5602, 2013.

[Perez *et al.*, 2015] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C. Lim, and T. Thompson. The 2014 general video game playing competition. *IEEE Trans. on Computational Intelligence and AI in Games*, (99), 2015.

[Schaul, 2013] T. Schaul. A video game description language for model-based or interactive learning. In *Proc. of IEEE Computational Intelligence in Games (CIG '13)*, pages 1–8, Aug 2013.

[Schmidhuber *et al.*, 2007] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. J. Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

[Shultz and Rivest, 2001] T. R. Shultz and F. Rivest. Knowledge-based cascade-correlation: Using knowledge to speed learning. *Connection Science*, 13(1):43–72, 2001.

[Sinno and Yang, 2010] P. J. Sinno and Q. Yang. A survey on transfer learning. *Knowl. and Data Eng.*, (10):1345–1359, 2010.

[Swarup and Ray, 2006] S. Swarup and S. R. Ray. Cross-domain knowledge transfer using structured representations. In *Proc. of AAAI*, pages 506–511, 2006.

[Talvitie and Singh, 2007] E. Talvitie and S. Singh. An experts algorithm for transfer learning. In *Proc. of IJCAI '07*, pages 1065–1070, 2007.

[Taylor and Stone, 2009] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, pages 1633–1685, 2009.

[Taylor *et al.*, 2007] M. E. Taylor, S. Whiteson, and P. Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *Proc. of AAMAS '07*, pages 37:1–37:8, 2007.

[Taylor *et al.*, 2008] M. E. Taylor, G. Kuhlmann, and P. Stone. Autonomous transfer for reinforcement learning. In *Proc. of AAMAS '08*, pages 283–290, 2008.

Width-based Planning for General Video-Game Playing

Tomas Geffner
Universidad de Buenos Aires
Buenos Aires, ARGENTINA
tomas.geffner@gmail.com

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, SPAIN
hector.geffner@upf.edu

Abstract

Iterated Width is a simple search algorithm that assumes that states can be characterized in terms of a set of boolean features or atoms. In particular, IW(1) consists of a standard breadth-first search with one variation: a newly generated state is pruned if it does not make a new atom true. Thus, while a breadth-first search runs in time exponential in the number of atoms, IW(1) runs in linear time. Variations of the algorithm have been shown to yield state-of-the-art results in classical planning and more recently in the Atari video games. In this paper, we use the algorithm for selecting actions in the games of the general video-game AI competition (GVG-AI) which, unlike classical planning problems and the Atari games, are stochastic. We evaluate a variation of the algorithm over 30 games under different time windows using the number of wins as the performance measure. We find that IW(1) does better than the sample MCTS and OLMCTS controllers for all time windows with the performance gap growing with the window size. The exception are the puzzle-like games where all the algorithms do poorly. For such problems, we show that much better results can be obtained with the IW(2) algorithm which is like IW(1) except that states are pruned in the breadth-first search when they fail to make true a new *pair* of atoms.

1 Introduction

In its early years, AI researchers used computers for exploring intuitions about intelligence and for writing programs displaying intelligent behavior. Since the 80s, however, there has been a shift from this early paradigm of writing programs for specific, sometimes ill-defined problems to developing solvers for well-defined mathematical models like constraint satisfaction problems, STRIPS planning, SAT, Bayesian networks, partially observable Markov decision processes, logic programs, and general game playing [Geffner, 2014]. Unlike the early AI programs, solvers are general as they must deal with any instance that fits the underlying model. The generality captures a key component of intelligence and raises a crisp computational challenge, as these models are all computationally intractable. Indeed, for these solvers to scale up, they must be able to exploit the structure of the problems automatically. The generality and scalability of solvers is tested experimentally through benchmarks and in most cases through competitions.

The general video-game AI playing competition (GVG-AI) is one of the latest developments in this trend toward empirically tested and scalable solvers [Perez *et al.*, 2015]. The competition provides a setting for evaluating action selection mechanisms on a class of Markov decision processes (MDPs) that represent video-games. As in other competitions, some of the games are known and some are new. The information available to the decision mechanisms are the states, the rewards, and the status of the game as resulting from a simulator. The games are defined in a compact manner through a convenient high-level language called VGDL [Schaul, 2013] but the resulting descriptions are not available to the solvers which have to select an action from a small pool of actions every 40 milliseconds. The competition is closely related to the existing MDP planning competition [Younes *et al.*, 2005; Coles *et al.*, 2012], except that the MDPs express video games, the MDPs are specified in a different language that is not available to the solvers, and the time windows for action selection are very short precluding much exploration before decisions.

Like in the MDP competition, the algorithms that do best in the GVG-AI setting tend to be based on variations of Monte-Carlo Tree Search [Chaslot *et al.*, 2008]. While Monte-Carlo planning methods evaluate each of the applicable actions by performing a number of stochastic simulations starting with each of the actions, in Monte-Carlo Tree Search, the average rewards obtained from such simulations are used to incrementally build a tree of the possible executions, which may deliver good decisions over short time windows, while ensuring optimal decisions asymptotically. Since its first success in the game of Go, where leaves of the tree are initialized with values obtained from a given informed base policy [Gelly and Silver, 2007], variations of the basic MCTS algorithm called UCT [Kocsis and Szepesvári, 2006] have been successfully used in a number of contexts, including the MDP and GVG-AI competitions.

MCTS methods, however, do not do well in problems with large spaces, sparse rewards, and non-informed base poli-

cies where they generate random actions and bootstrap very slowly. Indeed, plain MCTS methods make no attempt at exploiting the structure of such problems. This is in contrast with the methods developed for example in classical planning, SAT, and CSP solving, where good, general methods have been devised for exploiting structure successfully in the form of automatically derived heuristic functions or effective forms of inference [Geffner and Bonet, 2013; Biere *et al.*, 2009; Rossi *et al.*, 2006].

The aim of this paper is to gain further insights on the computational challenges raised by the GVG-AI framework by using a different type of algorithm whose origin is precisely in classical planning where it has been shown to be very effective over large spaces [Lipovetzky and Geffner, 2012]. The complete algorithm, called Iterated Width or IW, is a sequence of calls IW(1), IW(2), ..., where IW(k) is a breadth-first search in which a new state is pruned right away when it is not the first state in the search to make true some subset of k atoms or boolean features. In classical planning, the algorithm has been shown to be effective for decomposing a problem into subproblems and for solving the subproblems. More recently, the algorithm IW(k) with the parameter k fixed to 1 [Lipovetzky *et al.*, 2015] has been shown to outperform MCTS in the Atari games within the Arcade Learning Environment [Bellemare *et al.*, 2013]. In this work, we consider variations of IW(1) and IW(2) within the GVG-AI framework where the games can be stochastic. The results are no less interesting. A key difference between IW and most other classical planning algorithms is that the latter require a compact encoding of the problems in a PDDL language or similar for either inferring heuristic values or translating them into SAT. IW, on the other hand, works with simulators, very much like MCTS and brute-force search methods, but unlike them IW makes an attempt at exploiting the *factored structure of states* in a way that has been shown to pay off in many settings.

The paper is organized as follows. First, we review the IW algorithm, the GVG-AI framework, and the way IW(1) is used. We look then at the empirical results, consider a version of the IW(2) algorithm, and end with a summary and discussion.

2 Basic Algorithms: IW and IW(1)

The iterated width algorithm (IW) has been introduced as a classical planning algorithm that takes a planning problem as an input and computes an action sequence that solves the problem as the output [Lipovetzky and Geffner, 2012]. The algorithm however applies to a broader range of problems which can be characterized in terms of a finite set of states where each state s has a *structure* given by a set of features or variables that can take a finite set of values. A state assigns values to all the variables. In classical planning, the state variables are boolean, while in ALE, for example, the state is a vector of 1024 bits organized into 128 words, which can be regarded as composed of 1024 boolean variables or as 128 variables that can take up to 2^8 values each. Width-based algorithms are sensitive to these choices, and in particular, to the resulting set of atoms. An *atom* $X = x$ represents the assignment of value x to variable X, while a *tuple of atoms* is a set of atoms involving different variables. The *size* of a tuple is the number of atoms that it contains. A state s makes an atom $X = x$ true if the value of X in s is x, and makes a tuple of atoms t true if it makes each atom in t true. Likewise, the state s makes the atom false if it doesn't make it true, and makes a tuple of atoms false if it makes false one of the atoms in the tuple.

IW is a sequence of calls IW(i) for $i = 1, 2, \ldots$ where IW(i) is a plain *breadth-first search* with one change: right after a new state s is generated in the search, the state is pruned if s does not make true a new tuple of at most i atoms. That is, the state s is *not* pruned in the breadth-first search only if there is a tuple t of size no greater than i such that t is true in s and false in all the states generated in the search before s.

As an illustration, IW(i) for $i = 1$, i.e., IW(1), is a breadth-first search where a newly generated state s is pruned when there is no *new atom* made true by s. Similarly, IW(2) is a breadth-first search where a newly generated state s is pruned when there is no *new atom pair* made true by s, and so on.

A key property of the algorithm is that while the number of states is *exponential* in the number of atoms, IW(i) runs in time that is exponential in i only. In particular, IW(1) is linear in the number of atoms, while IW(2) is quadratic. Furthermore, Lipovetzky and Geffner define a general *width* measure for problems P and prove that IW(i) solves P when the width of P is no greater than i. Moreover in such a case, IW(i) solves P optimally (i.e., it finds a shortest solution). For example, *any* blocks world problem where the goal is to have one block x on top of another block y can be shown to have width 2 no matter the number of blocks or configuration. As a result, all such problems can be solved in quadratic time by IW(2) although the number of states is exponential. On the other hand, the same problem with joint goals, as when a tower of blocks is to be built, does not have a bounded width. The serialized iterated algorithm (SIW) proposed by Lipovetzky and Geffner uses IW sequentially for achieving the joint goal, one subgoal at a time. The algorithm is effective on most classical planning benchmarks where there are general and effective ways for serializing goals into subgoals, and where the width of the resulting subproblems is bounded and low. At the same time, the algorithm IW(i) with i set to the number of problem variables is equivalent to breadth-first search and hence complete for all problems provided that different states disagree on the truth of some atom or feature.

The algorithm IW(1) has been used recently for playing Atari video-games *on-line* in the Arcade Learning Environment [Bellemare *et al.*, 2013] where it was shown to achieve state-of-the-art performance [Lipovetzky *et al.*, 2015]. For this, the atoms $X_i = x$ were defined so that there is one variable X_i for each of the 128 words in the state vector, and one atom $X_i = x$ for each of its 2^8 possible values. IW runs in time linear in the number of atoms, which in this representation is $128 \times 256 = 2^{15}$. The alternative representation where the variables X_i are associated with each of the 1024 bits in the state vector results into a smaller set of $1024 \times 2 = 2^{11}$ atoms, making IW(1) run faster but with poorer results.

In the Atari games, there is no crisp goal to achieve but rewards to be collected, and planning is done *on-line* by se-

lecting actions from the *current state* using the simulator in a lookahead search. This search is performed with the IW(1) algorithm with little modification, keeping the (discounted) total reward accumulated in each non-pruned path in the breadth-first search, and selecting the first action of the path with most reward. A similar idea will be used in the GVG-AI setting.

Last, for using the IW(i) algorithms it is not necessary for the states s to explicitly encode the value x_i of a set of variables X_i. An alternative is to associate the variables X_i with a set of *features* ϕ_i so that the atoms made true by the state s are of the form $X_i = \phi_i(s)$. Indeed, the algorithm IW(i) can be understood as the algorithm IW(1) with a set of boolean features ϕ_k, each of which checks whether a tuple t_k of size no greater than i is true in s.

3 The GVG-AI Framework

The games supported in the GVG-AI framework are MDPs that represent video games [Perez *et al.*, 2015]. The videogames consist of objects of different categories that live in a rectangular grid and have dynamic properties. There are four actions for moving an avatar in the four directions and a fifth "use" action whose effects are game and state dependent but which basically makes use of the resources that the avatar may have. The categories of objects includes avatar, non-playing character (NPC), movable and static objects, resources, portal, and from–avatar (like bullets fired by avatar). Changes occur due to the actions performed, object collisions, and the internal dynamics of objects. The games are defined in a compact manner through a convenient high level language called VGDL [Schaul, 2013] that is mostly declarative except for the changes resulting from collisions that are expressed in Java code. The game descriptions are not available to the solvers which have to select an action every 40 milliseconds by interacting with the simulator. The simulator provides information about the state, the rewards, and the status of the game (won/lost/on-going) while applying the actions selected. The range of problems that can be expressed elegantly in VGDL is very broad and includes from simple shooter games to Sokoban puzzles

An important part of the game description is the object type hierarchy tree (sprite set). For example, the tree for the game Butterfly has nodes *cocoon* and *animal*, the latter with children *avatar* and *butterfly*. We call the nodes in these hierarchies, *stypes* (sprite types). The stype of an object is associated with its dynamic and behaviors and is readable from the state observation. Stypes are important in our use of the IW algorithms as the set of total stypes in a game is finite and usually small (smaller than 20 in general), while the set of objects and the stype of an object may change dynamically.

The GVG-AI setting is similar to the one used in the MDP planning competition [Younes *et al.*, 2005; Coles *et al.*, 2012], the main difference being that the MDPs are described in a different language and that the game descriptions are not available to the solvers.

4 IW Algorithms in GVG-AI

For using the IW algorithms in the GVG-AI setting, two issues need be addressed: first the definition of the boolean features or atoms; second, the stochasticity of the games. We consider these two issues next.

Since every object has a unique ID id and a number of dynamic features ϕ_i, we could associate problem variables $X[id, i]$ with values $\phi_i(id, s)$ representing the value of feature ϕ_i of object id in the state s. There are however two problems with this. First, the set of objects is dynamic. Second, the resulting number of atoms ends up being too large even for a linear time algorithm like IW(1) given that the time window for decisions is very small: 40 milliseconds. Indeed, running IW(1) to completion in such a representation may require up to two orders-of-magnitude more time in some games.

In order to avoid dealing with either a *dynamic* set of variables or a set of atoms that is *too large*, we have chosen to define the set of boolean features as the ones representing just whether an object of a certain *stype* is in a given grid cell. If there are N grid cells and T stypes in the object hierarchy, the number of atoms $at(cell, stype)$ would be $N \times T$, no matter the number of actual objects. In addition, since the actions control the avatar, we treat the avatar stypes (stypes beneath the avatar node in the hierarchy if any, else the stype avatar itself) in a different way. For avatar subtypes we consider the atoms $avatar(cell, angle, atype)$ where $angle$ is the possible avatar orientations (between 1 and 4 depending on the game), and $atype$ is the avatar stype (1 or 2; e.g. avatar-with-key and avatar-with-no-key). With this provision, the total number of atoms is in the order of $N \times T + N \times O \times A$ where O and A represent the avatar orientations and stypes respectively. For a grid with 100 cells and a hierarchy of 10 stypes, this means a number of atoms no greater than 1800. The IW(1) lookahead search will not expand then more than 1800 nodes.

The second issue to be addressed is the stochasticity of the games. In principle, IW algorithms can be used off the box oblivious to the fact that calling the simulator twice from the same state and with the same action may result in different successor states. Indeed, IW(i) would never call the simulator twice in that fashion. This is because IW(i) will not apply the same action twice in the same node, and the search tree generated by IW(i) can never have two nodes representing the same state (the tuples made true by the second node must have been made true by the first node). Yet ignoring stochasticity altogether and considering only the first outcome produced by the simulator can be too risky. For example, if the avatar has a killing monster to its right that moves randomly, it would be wise to move away from the monster. However, IW(i) may find the action of moving right safe, if the simulator returns a state where the monster also happened to move right. Thus to temper the risky optimism that follows from ignoring all but the state transitions returned first by the simulator when other transitions are possible, we add to IW(i) a simple safety check. Before running IW(i) for selecting an action in a state s, we take M samples of the successor state of each action a applicable in s, and count the number of times $D(a, s)$ where the avatar dies as a result of that one

step (game lost). The actions a that are then regarded as *safe* in s are all the actions applicable in s that have a minimum count $D(a, s)$. This minimum count does not have to be zero, although hopefully it will be smaller than M. Equivalently, if $D(a, s) < D(a', s)$ for two applicable actions a and a' in s, action a' will be declared not safe in s.

The applicable actions that are labeled as *not safe* in the current state s are then treated as if they were not applicable in s. That is, when deciding which action to apply in s, actions that are not safe in s are deemed not applicable in s when performing the looking ahead from s using the IW(i) algorithm. We call this pruning, *safety prepruning* or just *prepruning*. This pruning is shallow and fast, but helps to tame optimism when risk is immediate. The IW(i) algorithms below use this simple form of prepruning. For comparison, a plain breadth-first search is also used in the experiments with and without prepruning. In all cases, the number M of samples is set to 10.

This way of dealing with stochastic state transitions is myopic but practical. Actually, it's common to design closed-loop controllers for stochastic systems using simplified deterministic models, letting the feedback loop take care of the modelling error. Similar "determinization" techniques have been used for controlling MDPs even without risk detection. Indeed, the on-line MDP planner called FF-replan [Yoon *et al.*, 2007] performed very well in the first two MDP competitions making the assumption that the uncertain state transitions are under the control of the planning agent. This converts the MDP into a classical planning problem that can be solved very efficiently. The resulting plans are executed and monitored, and when a state is observed that is not the one predicted by the simplified model, a new call to the classical planner is made from the observed state (replanning) and the process iterates until reaching the goal.

In our use of the IW algorithms, the determinization is not made by the planning agent but by the simulator, replanning is done at every step, and the prepruning trick is added to deal with immediate risk.

5 Experimental Results

For testing the IW algorithms in the GVG-setting, we used the software available in the competition site along with the 3 sets of 10 games each available.[1] For comparison, we consider the vanilla Monte Carlo Tree Search (MCTS) and the Open Loop MCTS (OLMCTS) controllers provided by the organizers [Perez *et al.*, 2015], a breadth-first search lookahead (BrFS), a simple 1-step lookahead (1-Look), and random action selection (RND). Moreover, for BrFs, we consider a variant with prepruning and one without. The 1-lookahead algorithm uses the $M = 10$ samples not only to prune unsafe actions but to choose the safe action with the most reward. Initially we evaluate the IW(1) algorithm but then consider IW(2) and an additional variant. In order to learn how time affects the performance of the various algorithms, we we consider three time windows for action selection: 40 milliseconds, 300 milliseconds, and 1 second. The experiments were run on an AMD Opteron 6300@2.4Ghz with 8GB of RAM.

[1] The competition site is `http://www.gvgai.net`.

Tables 1–3 show the performance of the algorithms for each of the game sets. The tables include BrFS (with prepruning), MCTS, OLMCTS, IW(1), 1-Look, and RND. We focus on a crisp performance measure: the number of games won by each algorithm. Since there are 5 levels in each game, and for each level we run 5 simulations, the maximum number of wins per game is 25. The total number of wins in each set of games is shown in the bottom row, which is bounded by the total number of games played in the set (250).

For the first set of games, shown in Table 1, we can see that IW(1) wins more games than MCTS and OLMCTS for each of the three time windows. For 40 msecs, the number of wins for MCTS, OLMCTS, and IW(1) are 89, 93, and 145 respectively, and the gap in performance grows with time: with 260 msecs more, MCTS and OLMCTS win 20 and 17 more games respectively, while IW(1) wins 36 more. Then, MCTS and OLMCTS do not win additional games when the window is extended to 1 second, while IW(1) wins 10 games more then. The total number of games won by each of these three algorithms when the time window is 1 second is thus 108, 108, and 191 respectively. Surprisingly, breadth-first search with prepruning does also very well in these games; indeed, better than MCTS and OLMCTS for all time windows, although not as well as IW(1), which also takes better advantage of longer times. BrFS with prepruning wins 125 games with 40 msecs, while normal BrFS without the pruning (not shown), wins 107. The two algorithms win 137 and 123 games respectively with 1 second. 1-step lookahead and random action selection end up winning 67 and 20 games respectively.

The pattern for the second set of games, shown in Table 2 is similar: IW(1) does best for all time windows and in this case by larger margins. For 40 msecs, MCTS, OLMCTS, and IW(1) win 68, 70, and 110 games respectively, while the numbers are 77, 90, and 183 for 1 second. Again, BrFS with prepruning does slightly better than MCTS for all time windows, although for this set it is tied with OLMCTS. BrFS without prepruning, on the other hand, does slightly worse. 1-step lookahead does almost as well as MCTS winning 62 of the games, while random selection wins 28.

Table 3 for the third set of games is however different. These games are more puzzle-like, like Sokoban, and as a result all the algorithms do poorly, including IW(1), that actually does worse than MCTS, OLMCTS, and even BrFS. The number of wins by MCTS and OLMCTS reach the 59 and 60 games, while IW(1) does not get beyond 45. Moreover, in these games, IW(1) does not get better with time. Of course, these games are more challenging, and indeed, none of the algorithms manages to solve 25% of the games for any time window. In the previous two sets of games, IW(1) managed to win more than 72% of the games with 1 second.

IW(2) and Variations

A key question is what makes the puzzle-like problems in the third set actual puzzles and thus more challenging. It's definitely not the size of the state space; indeed, the famous blocks world has a state space that is exponential in the number of blocks, and yet it's not a puzzle (unless one is looking for provably shortest solutions). A second question is why on

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Aliens	24	**25**	**25**	**25**	25	25	25	25	25	25	25	25	10	5
Boulderdash	0	0	**2**	1	5	1	0	**7**	5	2	0	**17**	1	0
Butterflies	24	24	**25**	**25**	25	25	25	25	25	25	25	25	22	9
Chase	3	0	0	**6**	5	1	1	**17**	8	0	0	**14**	1	0
Frogs	17	4	5	**20**	15	6	8	**25**	17	6	8	**25**	12	0
Missile Command	15	13	13	**17**	19	22	20	**25**	16	23	21	**25**	8	5
Portals	15	2	3	**17**	14	3	0	**20**	14	1	1	**22**	5	0
Sokoban	5	**7**	6	2	8	**10**	9	3	9	**11**	10	1	0	0
Survive Zombies	**13**	10	12	12	9	13	**14**	**14**	8	12	**14**	**14**	7	1
Zelda	9	4	2	**20**	9	3	8	**20**	10	3	4	**23**	1	0
Total	125	89	93	**145**	134	109	110	**181**	137	108	108	**191**	67	20

Table 1: Performance over first set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Camel Race	**1**	0	**1**	0	1	3	0	**24**	1	0	3	**25**	0	1
Digdug	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
Firestorms	**13**	2	2	11	14	7	6	**25**	12	3	4	**23**	10	0
Infection	22	21	19	**23**	21	19	**22**	21	20	22	20	**25**	19	22
Firecaster	0	0	0	0	0	0	**1**	0	0	0	0	**3**	0	0
Overload	10	4	7	**19**	17	3	5	**23**	17	5	5	**25**	0	0
Pacman	**1**	0	0	**1**	1	1	4	**14**	2	2	7	**22**	0	0
Seaquest	13	**16**	14	15	11	17	**22**	9	4	20	**24**	18	12	0
Whackamole	23	23	**24**	20	22	23	**25**	21	24	22	**25**	19	21	5
Eggomania	1	2	3	**15**	0	0	2	**22**	3	3	2	**23**	0	0
Total	84	68	70	**104**	87	73	87	**159**	84	77	90	**183**	62	28

Table 2: Performance over second set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

such problems IW(1) does poorly, and moreover, doesn't get better with time. And finally, how can such problems be addressed computationally using generic methods.

Fortunately, for these questions, the theory behind the IW(i) algorithms helps [Lipovetzky and Geffner, 2012]. From this perspective, a problem is easy either if it has a low width, or if it can be easily serialized into problems with low width. This is certainly true for blocks world problems but not for problems like Sokoban. Indeed, while IW(1) runs in time that is linear in the number of features, it is *not* a complete algorithm in general; it is provably complete (and actually optimal) for width 1 problems only. Problems like Sokoban have a higher width even when a single stone needs to be placed in the goal, in the presence of other stones. One way to deal with such problems is by moving to a IW(i) algorithm with a higher width-parameter i.

Table 4 shows the results of the IW(2) algorithm for the third set of puzzle-like problems. The algorithm runs in time that is quadratic in the number of atoms and thus is less efficient than IW(1) but as it can be seen from the table, by 1 second, it has won 81 of the games, many more than those won in 1 second by MCTS and OLMCTS: 60 and 58. The three algorithms have a similar performance for 40 milliseconds but, while MCTS and OLMCTS do not improve much then, the opposite is true for IW(2). The atoms used by IW(2) are the same as those used by IW(1) but these atoms are used in *pairs* for pruning states in the breadth-first search. Given the special role of the avatar in the GVG-AI games, we also considered a variant of IW(2) that we call IW(3/2), which pays attention to *some* atom pairs only: those where one of the atoms is an avatar atom, i.e., an atom of the form $avatar(cell, angle, atype)$. As it can be seen in the table, IW(3/2) does better than IW(2), as while the resulting search considers a smaller set of nodes, it gets deeper in the search tree faster. The same is true for IW(1) in the first two sets of "easier" problems where it does better than both IW(2) and IW(3/2) for the same reasons (see Tables 5 and 6). The IW(i) algorithms are all breadth-first search algorithms that perform a more aggressive pruning the smaller the width-parameter i. More aggressive pruning means that the breadth-first search gets deeper sooner, while less aggressive pruning means that more paths are explored. There is thus a tradeoff: on easy games and small time windows, IW(1) is the best choice, but for harder problems and more time IW(3/2) and IW(2) will do better. As mentioned above, neither of these algorithms by themselves, nor MCTS or OLMCTS, will do well in simple serializable problems with joint goals, like when a tower of blocks needs to be built, even if the individual goals are rewarded. The reason is that just one serialization will do the job; namely, the one that achieves the goals bottom up. Methods for obtaining such serializations automatically using width-based algorithms are discussed in [Lipovetzky and Geffner, 2012], yet such methods, like other classical planning methods do not work with simulations and require a factored representation of the action effects and goals.

5.1 Profiles in Time

The performance of the algorithms over all the games is summarized in the curves shown in Figure 1. For each algorithm, the curve shows the total number of games won in the Y axis as a function of the three time windows shown in the X axis. The flat line shows the performance of 1-step lookahead. The curves for MCTS, OLMCTS, and BrFS show a higher number of wins for 40 milliseconds that however becomes prac-

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Bait	3	2	2	**7**	1	2	2	**5**	1	3	2	**5**	2	2
Bolo Adventures	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0
Brain Man	**1**	**1**	**1**	0	**2**	**2**	**2**	0	1	0	**3**	0	0	0
Chips Challenge	4	**5**	4	0	**6**	3	4	0	**7**	5	5	2	4	0
Modality	**8**	6	5	6	**9**	6	8	5	5	**6**	**6**	5	5	2
Painters	**25**	21	24	**25**	**25**	**25**	24	**25**	25	25	25	25	6	8
Real Portals	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Real Sokoban	0	0	0	0	0	0	**9**	0	0	0	0	0	0	0
The Citadel	2	**5**	1	1	3	**4**	2	3	4	**7**	4	2	1	0
Zen Puzzle	5	**9**	5	6	6	15	**16**	6	6	**13**	**13**	5	2	2
Total	48	**49**	42	45	52	57	**59**	44	49	**60**	58	44	20	14

Table 3: Performance over third set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

Time	40ms			300ms			1s		
Game	IW1	W2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Bait	**7**	5	5	5	5	**14**	5	10	**19**
Bolo Adv	0	0	0	0	0	0	0	0	0
Brain Man	0	1	**2**	0	0	**5**	0	3	**8**
Chips Ch	0	**2**	0	0	**2**	**2**	2	4	**5**
Modality	**6**	5	**6**	5	15	**20**	5	**20**	**20**
Painters	25	25	25	25	25	25	25	25	25
Real Port	0	0	0	0	0	0	0	0	0
R Sokoban	0	0	0	0	0	0	0	0	0
Citadel	1	**2**	0	3	3	**7**	2	12	**17**
Zen Puzz	**6**	**6**	5	6	6	**8**	5	7	**16**
Total	45	**46**	43	44	56	**81**	44	81	**100**

Table 4: IW(1), IW(2) and IW(3/2) in third set of games

Time	40ms			300ms			1s		
Game	IW1	IW2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Aliens	**25**	13	**25**	25	25	25	25	25	25
Boulder	1	0	**2**	**7**	0	6	**17**	0	6
Butterf	**25**	21	24	**25**	22	**25**	25	25	25
Chase	**6**	1	5	**17**	3	14	14	4	**16**
Frogs	**20**	0	17	**25**	0	21	**25**	0	**25**
Missile	**17**	10	13	**25**	18	20	**25**	20	24
Portals	**17**	10	11	**20**	12	16	**22**	13	16
Sokoban	2	2	**6**	3	12	**15**	1	15	**20**
Survive	**12**	8	9	**14**	6	9	**14**	10	11
Zelda	**20**	9	16	**20**	16	18	**23**	18	17
Total	**145**	74	128	**181**	114	169	**191**	130	185

Table 5: IW(1), IW(2) and IW(3/2) in first set of games

Time	40ms			300ms			1s		
Game	IW1	IW2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Camel Race	0	**1**	0	**24**	0	5	**25**	0	5
Digdug	0	0	0	0	0	0	0	0	0
Firestorms	**11**	4	9	**25**	6	15	**23**	6	14
Infection	**23**	16	21	21	17	**24**	**25**	21	22
Firecaster	0	0	0	0	0	0	**3**	0	0
Overload	**19**	12	17	23	19	**24**	**25**	23	**25**
Pacman	**1**	0	**1**	**14**	0	3	**22**	0	6
Seaquest	**15**	11	9	9	9	**10**	**18**	9	5
Whackamole	20	20	**22**	**21**	**21**	11	19	24	**25**
Eggomania	**15**	0	0	**22**	0	14	**23**	1	20
Total	**104**	64	79	**159**	72	117	**183**	84	122

Table 6: IW(1), IW(2) and IW(3/2) in second set of games

tically flat after 300 milliseconds. The curve for IW(1) starts higher than these curves and increases until reaching its peak at 1 second where the gap with the previous algorithms is very large. The curve for IW(2), on the other hand, starts lower but overtakes MCTS, OLMCTS, and BrFS after 300 milliseconds. Finally, the curve for IW(3/2) is similar to the curve for IW(1) although the two algorithms do best on different game sets: IW(1) on the easier games, and IW(3/2) on the last set.

6 Summary and Discussion

We have evaluated the IW(1) algorithm in the games of the GVG-AI framework and shown that it appears to outperform breadth-first search and plain Monte-Carlo methods like MCTS and OLMCTS. The exception are the puzzle-like games where all the algorithms do rather poorly. The theory behind the IW(i) algorithms provides useful hints for understanding why. The algorithm IW(2) does indeed much better over such problems but needs more time. The width parameter i in IW(i) captures a tradeoff: the higher the number, the higher the quality of the decision when given more time.

IW algorithms operate on state spaces where the relevant information about states can be encoded through a number of boolean features. For the games in the GVG-AI setting, these boolean features have been identified with atoms of the form $at(cell, stype)$ and $avatar(cell, angle, atype)$ where the stypes (atypes) correspond to the types of objects (avatars) in the sprite set. IW(1) preserves new states that bring new atoms and hence runs in time linear in this set of atoms, while IW(2) preserves new states that bring new pair of atoms, and runs in quadratic time. The algorithm IW(3/2) achieves a further tradeoff by considering only some pairs: those that include an avatar atom $avatar(cell, angle, atype)$.

IW algorithms are oblivious to the stochasticity of the domain as they only consider one successor state s' for every applicable action a in a state s: the one returned by the stochastic simulator the first time it's called with a and s. Since this may be too risky sometimes, before running IW(i) (and BrFS) from a state s in the GVG-AI games, actions that are deemed not safe in s after a 1-lookahead step that uses $M = 10$ samples, are deemed as not applicable in s. This way of dealing with uncertainty has similarity with forms of optimistic planning used for dealing with MDPs [Yoon *et al.*, 2007].

A key difference between IW algorithms and MCTS is that the former make an attempt at exploiting the structure of the problems and states. This is why IW algorithms can do so well in classical planning problems that often involve huge state spaces. On the other hand, in problems with large spaces and sparse rewards, uninformed MCTS methods act randomly and bootstrap slowly. Similarly, a key difference

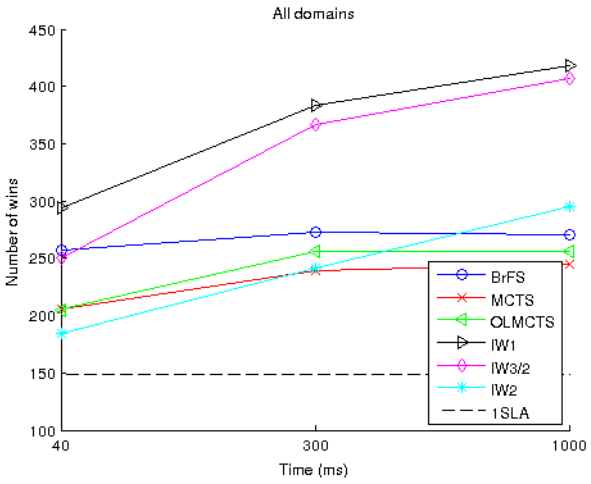


Figure 1: Total number of games won by each algorithm over the three sets of games as function of time window

between IW and standard classical planners is that the latter require an explicit, compact encoding of the action effects and goals.

These relations suggest a very concrete and crisp challenge; namely, how to plan effectively with a simulator when there is no explicit encoding of the action effects and goals in problems that are easy to serialize into simple problems but where not every serialization would work. The simplest example illustrating this challenge is no puzzle at all: it's the classical blocks world where the goal is to build a given tower of blocks. Getting rewards for achieving parts of the tower does not make the problem easier as the agent must realize that it has to work bottom up. Modern classical planners that have explicit descriptions of the action effects and goals do not run into this problem as they can easily infer such goal orderings. For simulation-based approaches, however, that is a challenge.

Acknowledgements: We thank the team behind the GVG-AI framework for a wonderful environment and set of problems, and Ivan Jimenez at UPF for help running the experiments. We also thank Nir Lipovetzky and Miquel Ramirez for the feedback.

References

[Bellemare *et al.*, 2013] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47(47):253–279, 2013.

[Biere *et al.*, 2009] A Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*. IOS Press, 2009.

[Chaslot *et al.*, 2008] GMJ Chaslot, M.H.M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3):343, 2008.

[Coles *et al.*, 2012] A. Coles, A.and Coles, A. Garcia Olaya, S. Jimenez, C. Linares, S. Sanner, and S. Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1):83–88, 2012.

[Geffner and Bonet, 2013] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers, 2013.

[Geffner, 2014] H. Geffner. Artificial Intelligence: From programs to solvers. *AI Communications*, 27(1):45–51, 2014.

[Gelly and Silver, 2007] S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *Proc. ICML*, pages 273–280, 2007.

[Kocsis and Szepesvári, 2006] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Proc. ECML-2006*, pages 282–293. Springer, 2006.

[Lipovetzky and Geffner, 2012] Nir Lipovetzky and Héctor Geffner. Width and serialization of classical planning problems. In *Proc. ECAI*, pages 540–545, 2012.

[Lipovetzky *et al.*, 2015] N. Lipovetzky, M. Ramirez, and H. Geffner. Classical planning algorithms on the atari video games. In *Proc. of 2015 AAAI Workshop on Learning for General Competency in Video Games*, 2015.

[Perez *et al.*, 2015] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couetoux, J. Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computional Intelligence and AI in Games*, 2015. Forthcoming. At julian.togelius.com/Perez20152014.pdf.

[Rossi *et al.*, 2006] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[Schaul, 2013] T Schaul. A video game description language for model-based or interactive learning. In *Proc. IEEE Computational Intelligence in Games (CIG-2013)*, pages 1–8, 2013.

[Yoon *et al.*, 2007] S. Yoon, A. Fern, and R. Givan. FF-replan: A baseline for probabilistic planning. In *Proc. ICAPS-07*, pages 352–359, 2007.

[Younes *et al.*, 2005] H. Younes, M. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res.(JAIR)*, 24:851–887, 2005.

Game Description Language for Real-time Games

Jakub Kowalski
Institute of Computer Science
University of Wrocław, Poland
jakub.kowalski@cs.uni.wroc.pl

Andrzej Kisielewicz
Institute of Mathematics
University of Wrocław, Poland
andrzej.kisielewicz@math.uni.wroc.pl

Abstract

We present a simple extension of the Game Description Language GDL that makes it possible to describe a large variety of games involving a real-time factor. Apart from the formal syntax and semantics of our extension (acronymed rtGDL), we provide also a series of illustrative examples. In addition, we show that by combining rtGDL with the GDL-II extension one may obtain a simple and elegant universal game description language that enables the formalization of the rules of arbitrary n-player games allowing randomness, imperfect information and time-dependent events.

1 Introduction

The aim of *General Game Playing* (GGP) is to develop a system that can play a variety of games with previously unknown rules. Unlike standard AI game playing, where designing an agent requires special knowledge about the game, in GGP the key is to create a universal algorithm performing well in different situations and environments.

The beginning of the GGP field dates from 1968 with the work of Pitrat [Pitrat, 1968] concerning the class of arbitrary chess-like board games. In 2005 General Game Playing was identified as a new Grand Challenge of Artificial Intelligence and from this year the annual International General Game Playing Competition (IGGPC) has taken place to foster and monitor progress in this research area [Genesereth *et al.*, 2005].

For the purpose of GGP, a new language called the *Game Description Language* (GDL) [Love *et al.*, 2006] was developed. GDL has enough power to describe all turn-based, finite and deterministic n-player games with full information.Playing a game given by such a description requires not only developing a move searching algorithm, but also implementing a reasoning approach to understand the game rules in the sense of computing legal moves, the state update function, and the goal function. In 2010, GDL-II, the extension to GDL removing deterministic and full information restrictions was proposed in [Thielscher, 2010].

Currently, the various aspects of General Game Playing are a popular research topic. A successful method of learning neural networks to play 29 from 49 tested Atari 2600 games at or above the human level was presented in [Mnih *et al.*, 2015]. The recently launched General Video Game AI Competition, focusing on learning how to play simple video games by simulation and experience (rather then learning knowledge from the rules), is summarized in [Perez *et al.*, 2015].

The classic, GDL-based branch of GGP is constantly aimed for improving performance of the players [Björnsson and Schiffel, 2013; Kowalski and Szykuła, 2013], and developing better playing algorithms in both perfect information [Finnsson, 2012; Tak *et al.*, 2012] and imperfect information domain [Geißer *et al.*, 2014]. For a survey of General Game Playing and International GGP Competition, its history, achievements and actual research challenges we refer to [Genesereth and Thielscher, 2014] and [Genesereth and Björnsson, 2013].

In this paper, we follow the general idea of [Thielscher, 2010] to extend substantially the class of described games with minimal modifications to the language and communication protocol. We extend GDL to a Real-time Game Description Language (rtGDL), which can represent games involving time-dependent events, where the exact moment of an action is relevant. This concerns most current video games (such as RTS, FPS and RPG) for which computers are programmed to play these games using a specially designed algorithm [Ontanón *et al.*, 2013; Hingston, 2010].

We add only two new keywords, and an additional argument for the other three standard GDL relations. What we have achieved is that no real number arithmetic is required inside the rule engine, which unlike some other proposed extensions, e.g. [Thielscher and Zhang, 2010], preserve the pure declarative character of GDL. We present the formal syntax and semantics of this language, and also sketch a construction of the union of rtGDL and GDL-II, which yields the most universal game description language in the GDL family.

We assume basic familiarity with GGP and GDL structure. The paper is organized as follows. In the next section we introduce rtGDL as an extension of GDL. Section 3 provides formal semantics of Real-time GDL, while Section 4 covers the subject of the execution model. In the next section, the extension is illustrated by a series of examples. Section 6 contains a sketch of joining rtGDL with GDL-II yielding a language capable of describing games with both imperfect information and real-time factor. Concluding remarks are given in Section 7.

2 From GDL to rtGDL

Game Description Language [Love *et al.*, 2006], developed for the purpose of the General Game Playing Competition [Genesereth *et al.*, 2005] is a formal language to describe rules of any turn-based, finite, deterministic, n-player game with simultaneous moves and perfect information. GDL is a high-level, strictly declarative language using logic programming-like syntax based on the Knowledge Interchange Format (KIF [Genesereth and Fikes, 1992]). Every game description contains declarations of player roles, the initial game state, legal moves, state transition function, terminating conditions and the goal function.

GDL does not provide any predefined functions: neither arithmetic expressions nor game-domain specific structures like board or card deck. Every function and declaration must be defined explicitly from scratch, and the only keywords used to define the game are presented in Table 1. This ensures that, given the game rules, any additional, background knowledge is not required for the player.

`role(R)`	`R` is a player
`init(F)`	fact `F` is true in the initial state
`true(F)`	fact `F` is true in the current state
`legal(R,M)`	in the current state `R` can perform move `M`
`does(R,M)`	`R` performed move `M` in the previous state
`next(F)`	`F` will be true in the next state
`terminal`	current state is terminal
`goal(R,N)`	player `R` score is `N`

Table 1: GDL keywords

A surprisingly simple, but very powerful extension of GDL, called GDL-II (for Game Description Language with Incomplete Information) proposed by Thielscher [Thielscher, 2010; 2011a; 2011b], removes the restrictions that games have to be deterministic and with full information. This extension adds two additional keywords, and it slightly changes the communication protocol.

The extension of GDL presented in this paper is intended to remove another important restriction of GDL games that they need to be turn-based. We preserve the purely declarative style, so that the state computing inference engine can remain unchanged. In general, dealing with real-time games may involve real number arithmetic, which cannot be encoded in pure GDL in a simple way. The proposed approach requires real arithmetic on the level of search engine reasoning only. On the level of the rule engine, numbers are treated like standard terms without additional semantics (in a manner similar to natural numbers in a `goal` relation).

`init(T,F)`	`F` is true in the initial state for the time `T`
`true(T,F)`	`F` is true in the current state for the time `T`
`next(T,F)`	`F` is true in the next state for the time `T`
`infinity`	stands for ∞ when used as a time value
`expired(F)`	holds when `F` becomes obsolete

Table 2: The rtGDL keywords: the top three are modified GDL keywords, while the last two are new keywords

Changes between rtGDL and GDL keywords are summarized in Table 2. Keywords that are not listed: `role`, `legal`, `does`, `terminal` and `goal` remain unchanged. Parameter `T`$\in \mathbb{R}^+ \cup \{\infty\}$ linked with the currently holding fact `F` encodes the *lifetime* of this fact, that is the time for which this fact will be true. After the time is over, the state update is performed and the additional relation `expired(F)` indicates that `F` have just expired and allows changes based on this knowledge.

The `expired` keyword can be seen as just a syntactic sugar for the expression `true(0,F)`, however its introduction simplifies games rules (there is no more need for conditioning time value) and provides unambiguous semantics for obsolete facts. Thus, syntactic restrictions guarantees that 0 time value never occurs in `true` relation and `expired` is used instead.

State updates are performed after a player makes a move or when some fact becomes obsolete. Before such an update is computed, all times assigned to the set of holding facts are properly updated according to the time since the last update. The detailed semantics of rtGDL language is provided in the next two sections. Before going into formal details, the reader may want to see first examples in Section 5.

2.1 Formal Syntax Restrictions

Descriptions of rtGDL games use the standard syntax of logic programs, including negation and inequality (denoted as `distinct`). For the sake of readability, we will use Prolog convention rather than standard GDL code convention, that is, clauses are written in infix form, variables are uppercase letters, and function names start with lowercase letters.

As in the case of the standard GDL, certain syntactic restrictions have to be fulfilled to ensure that the game specification has an unambiguous interpretation, and all derivations are finite. First, exactly the same restrictions as in GDL and GDL-II are imposed on the keywords (a new keyword `expired` is restricted in the same way as `true`).

- `role` only appears in the head of ground atomic sentences;
- `init` only appears in the head of clauses and does not depend on `true`, `legal`, `does`, `next`, `terminal`, `goal` or `expired`;
- `true` only appears in the body of clauses;
- `does` only appears in the body of clauses and does not depend on `legal`, `terminal` or `goal`;
- `next` only appears in the head of clauses;
- `expired` only appears in the body of clauses.

Also, we adopt the convention that to be considered as *valid*, rtGDL game description must be *stratified* [Apt *et al.*, 1988], *allowed* [Lloyd and Topor, 1986], and satisfy the general *recursion restriction* (see [Schiffel and Thielscher, 2014, Definition 3.]). These restrictions ensure that game rules can be effectively and unambiguously interpreted by a state transition system.

Despite introducing time values, which in principle can be arbitrary real numbers, all derivations remain finite and decidable. This is due to the fact, that in each step there are

only a finite number of time values, which appear in the form of ground constants. In the next section we describe a unique game model that is obtained from a valid rtGDL game description using the concept of a unique stable model as in [Schiffel and Thielscher, 2014].

3 Semantics: A Game model for rtGDL

We show that rtGDL may be seen as a straightforward extension of GDL. As in GDL, any game description defines a finite set of domain-dependent function symbols and constants, which determines a (usually infinite) set of ground symbolic expressions Σ. These ground terms can represent players, moves or parts of the game state. However, unlike in the standard GDL, the game states are not simply subsets of Σ. Every individual feature of the game state has assigned a *lifetime*, which is a real number value or the infinity symbol. Such feature holds for this period of time, and after that (unless other events occur) it disappears from the state. Moreover, initially declared real numbers which occur in Σ are not the only ones. Any real number, given by the outside environment as updated lifetime, can appear as a constant later during the game.

Therefore, we define an rtGDL game state to be a finite subset of $\mathbb{R}_+^\infty \times \Gamma$, where $\mathbb{R}_+^\infty = (0, \infty]$ (including ∞), and $\Gamma \subseteq \Sigma(\mathbb{R}_+^\infty)$ where $\Sigma(\mathbb{R}_+^\infty)$ is a set of ground terms of the game, with the set of constants extended by $\mathbb{R}_+^\infty$.

Now, in introducing the elements of the game model, we follow the constructions for GDL and GDL-II given in [Schiffel and Thielscher, 2014]

- $R \subseteq \Sigma$ (the *roles*);
- $s_0 \subseteq \mathbb{R}_+^\infty \times \Sigma$ (the *initial position*);
- $t \subseteq \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *terminal positions*);
- $l \subseteq R \times \Gamma \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *legality relation*);
- $u : \mathbb{R}_+^\infty \times (R \nrightarrow \Gamma) \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma) \mapsto \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *update function*);
- $g \subseteq R \times \mathbb{N} \times \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$ (the *goal relation*).

(where $\mathcal{P}_{\text{fin}}(A)$ denotes the finite subsets of A) The legality relation $l(r, m, S)$, states that in the position S, a player r can perform move m. Given a time Δt since the last state update, and a partial function of the joint moves performed by the players $M : (R \nrightarrow \Gamma)$, the state update function $u(t, M, S)$ determines the updated position. Finally, the relation $g(r, n, S)$ defines the payoff of player r in the game state S.

3.1 State transition system

Let $\mathcal{G}$ be a valid game description of an n-player rtGDL game. The players' roles are defined by the derivable instances of `role(R)`. A *game state* $S = \{(t_1, f_1), \ldots, (t_k, f_k)\}$ is encoded as a set of facts with assigned lifetimes. At the beginning of the game, the state is composed of the derivable instances of `init(T,F)`. We use the keyword `true` to extend the game description $\mathcal{G}$ by the facts resulting from S, which form the set

$$S^{\texttt{true}} \stackrel{\text{def}}{=} \{\texttt{true}(t_1, f_1), \ldots, \texttt{true}(t_k, f_k)\}.$$

Instances of `goal(R,N)` derivable from $\mathcal{G} \cup S^{\texttt{true}}$ assign to every player `R` its goal value `N` in this state. If `terminal` is derivable from $\mathcal{G} \cup S^{\texttt{true}}$, then the state S is a terminal position. Computing legal actions, requires derivable instances of `legal(R,M)`, for every role `R` and move `M`.

The state of the game remains unchanged until an *event* occurs. There are two types of events, one caused by players sending their moves, and the other resulting from expiration of some facts in $S^{\texttt{true}}$. Let Δt be the time to the first occurrence of an event.

When a subset of players $r_{i_1}, \ldots, r_{i_l}$, where $\forall j \; i_j \in \{1, \ldots, n\}$, perform moves $m_{i_1}, \ldots, m_{i_l}$, then

$$M^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_{i_1}, m_{i_1}), \ldots, \texttt{does}(r_{i_l}, m_{i_l})\}.$$

Let us notice that $M^{\texttt{does}} = \emptyset$ if no player sends a move.

Performing state update requires updating lifetime of every holding fact, and information about facts that expired. These two sets are defined in a following way:

$$\mu(S, \Delta t) = \{\texttt{true}(t_i - \Delta t, f_i) : \texttt{true}(t_i, f_i) \in S^{\texttt{true}} \wedge t_i > \Delta t\}$$

$$S^{\text{exp}}_{\Delta t} \stackrel{\text{def}}{=} \{\texttt{expired}(f_i) : \texttt{true}(t_i, f_i) \in S^{\texttt{true}} \wedge t_i \leq \Delta t\}$$

where $\mu(S, \Delta t)$ updates the facts lifetimes and $S^{\text{exp}}_{\Delta t}$ contains expired facts. The updated position is composed of instances of `next(T,F)` derivable from $\mathcal{G} \cup M^{\texttt{does}} \cup \mu(S, \Delta t) \cup S^{\text{exp}}_{\Delta t}$.

The semantics of rtGDL, arising from the above description is presented in the definition below. We follow [Schiffel and Thielscher, 2014] using the unique finite stable model $\text{SM}[\mathcal{G}]$ rather then the derivability relation as in [Thielscher, 2010]. For details concerning the existence of stable models for logic programming the reader is referred to [Gelfond and Lifschitz, 1988].

Definition 1 *Let $\mathcal{G}$ be a valid rtGDL specification, whose signature determines the set of ground terms Σ, and $\Gamma \subseteq \Sigma(\mathbb{R}_+^\infty)$. The semantics of $\mathcal{G}$ is the state transition system (R, s_0, t, l, u, g) given by*

- $R = \{r \in \Sigma : \texttt{role}(r) \in \text{SM}[\mathcal{G}]\}$;
- $s_0 = \{(t, f) \in \mathbb{R}_+^\infty \times \Sigma : \texttt{init}(t, f) \in \text{SM}[\mathcal{G}]\}$;
- $t = \{S \in 2^{\mathbb{R}_+^\infty \times \Gamma} : \texttt{terminal} \in \text{SM}[\mathcal{G} \cup S^{true}]\}$;
- $l = \{(r, m, S) : \texttt{legal}(r, m) \in \text{SM}[\mathcal{G} \cup S^{true}]\}$, *for all $r \in R$, $m \in \Gamma$ and $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$*;
- $u(\Delta t, M, S) = \{(t, f) : \texttt{next}(t, f) \in \text{SM}[\mathcal{G} \cup M^{does} \cup \mu(S, \Delta t) \cup S^{exp}_{\Delta t}]\}$, *for all $M : (R \nrightarrow \Gamma)$, $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$, and minimal Δt such that $M^{does} \cup S^{exp}_{\Delta t} \neq \emptyset$*;
- $g = \{(r, n, S) : \texttt{goal}(r, n) \in \text{SM}[\mathcal{G} \cup S^{true}]\}$, *for all $r \in R$, $n \in \mathbb{N}$ and $S \in \mathcal{P}_{\text{fin}}(\mathbb{R}_+^\infty \times \Gamma)$.*

This defines a formal semantics for an abstract game model of rtGDL. The calculation of function μ takes place outside the state computation and is a part of the execution model, which is the subject of the next section

4 Execution Model

An execution model of rtGDL is designed to handle real-time events, which makes it a bit more complex than the standard execution model of GDL. After the initial state s_0 is computed, the timer is turned on and the game starts. The initial state is treated as the current one, until some event occurs which stops the timer. This event can be triggered by players who sent their moves, or it can be scheduled, known in advance, expiration of some state features. At the moment of the event, the state update is performed according to the game model semantics with Δt equal to time indicate by the timer. The timer is then turned on again, and the whole process repeats until a game reaches a terminal position. The final scores of the players are defined by the goal relation.

It should be noticed, that scheduled updates caused by fact expirations are silent, i.e. there is no message to the players indicating that the update took place. Such messages are unnecessary due to the fact that players have all the informations to perform such updates by themselves, based on control times sent by the Game Manager.

The straightforward implementation of a Game Manager according to this model looks as follows:

1. Send to the players the rtGDL code containing game description with information about their roles and binding timelimits. Set $S := S_0$. Turn the timer on.
2. After the time for the players to familiarize with game has passed, inform the players of the game beginning.
3. Restart the timer. Calculate t_u which is the minimal time for some position feature from S to become obsolete.
4. Wait until some players send their moves or the timer is equal to t_u. Set $M :=$ 'players moves', $\Delta t :=$ 'timer indications'.
5. If $M \neq \emptyset$ inform the players about the performed moves M and the current game time (since the beginning of the game).
6. Perform state update by setting $S := u(\Delta t, M, S)$.
7. If S is not a terminal state go to step 3. Otherwise, compute the payoff for every player accordingly to g relation and finish the game.

The game flow is unequivocally dictated according to the Game Manager's timer. In a case when a move sent by a player is not legal in the current state, update is not performed. There are no restrictions on the number of moves the players can send, and all of them should be considered by the Game Manager accordingly to their arrival times.

4.1 Communication Protocol

There are slight differences between the GDL and rtGDL communicates, but the main idea and architecture remains the same. For the detailed description of the GDL communication protocol we refer to [Genesereth *et al.*, 2005].

In rtGDL protocol there are no new types of messages, and all player replies remain unchanged. In the Game Manager message of type `START`, the semantic of `PLAYCLOCK` parameter has changed. Now it is used as a multiplier of fact lifetimes, i.e. to get the number of seconds before a fact becomes obsolete, its current lifetime has to by multiply by `PLAYCLOCK`. With `PLAYCLOCK` 20, fact `true(1.5,control(white))` becomes expired after 30 seconds, while fact `true(0.5,cooldown)` after 10 seconds. This preserves a very desirable possibility of controlling the game pace without changes in the game code. The semantic of the `STARTCLOCK` parameter remains unchanged.

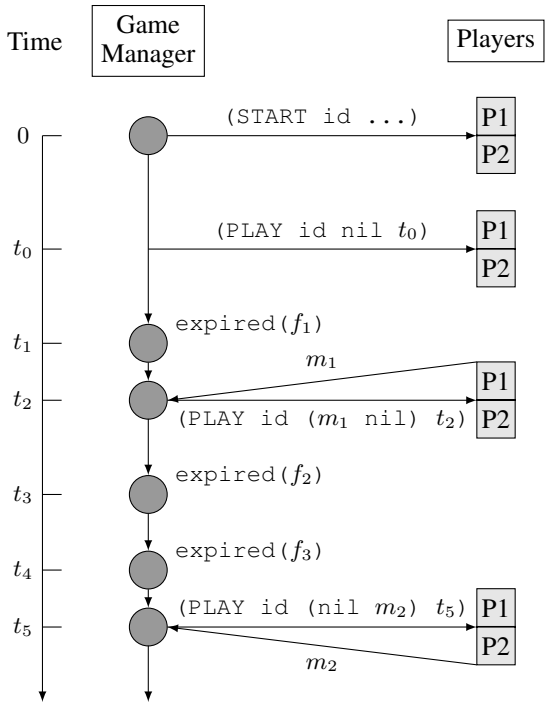


Figure 1: Visualization of communication during some rtGDL game. Circles in the Game Manager's column indicates a state update.

The `PLAY` and `STOP` messages are extended, and now their last, third, parameter is a game time as `PLAYCLOCK` multiplyer, i.e. the number of seconds (as a floating point number) from the beginning of the game to the moment when the message was sent by the Game Manager divided by the current `PLAYCLOCK` value.

The second argument of `PLAY`/`STOP` message is a list of actions taken by all players. This list can be set to `NIL` in two special cases: when this is the initial game message indicating the end of the preparation phase, or it is an answer to a player who sent an illegal move (other players should not receive this message). In a real-time gaming the Game Manager is receiving players' moves sequentially rather than simultaneously, which results that the lists of performed actions sent in `PLAY`/`STOP` messages are only partially filled. If a player on a position k made no move, his move is simply denoted as `NIL` which allows that the length of the list remains constant and equal to number of the players in the game. (In fact, the lists may be replaced by *role–move* pairs.)

An example of communication between Game Manager and players during exemplary rtGDL game with two players is presented in Figure 1.

5 Examples

To illustrate the expressiveness of Real-time Game Description Language, we provide a series of short examples describing real-time elements appearing in many games.

Example 1. Turn-based games

First, we demonstrate how to translate a turn-based GDL game into rtGDL. For this purpose we use the standard Tic-tac-toe game. The following code describes the most important parts of the suitable rtGDL description:

```
role(xplayer).
role(oplayer).
init(infinity,cell(1,1,blank)).
...
init(infinity,cell(3,3,blank)).
init(1.0,control(xplayer))

next(1.0,control(S))
    ⇐ does(R,M)
    ∧ role(S)
    ∧ distinct(R,S).

next(infinity,cell(M,N,R))
    ⇐ does(R,mark(M,N))
    ∧ true(infinity,cell(M,N,blank)).
next(infinity,cell(M,N,C))
    ⇐ true(infinity,cell(M,N,C)).
    ∧ distinct(C,blank).
next(infinity,cell(M,N,blank))
    ⇐ true(infinity,cell(M,N,blank))
    ∧ does(R,mark(J,K))
    ∧ (distinct(J,M) ∨ distinct(K,N)).

legal(R,mark(M,N))
    ⇐ true(infinity,cell(M,N,blank))
    ∧ true(T,control(R)).

terminal ⇐ expired(control(R)).

goal(R,0) ⇐ expired(control(R)).
goal(S,100)
    ⇐ expired(control(R))
    ∧ role(S)
    ∧ distinct(R,S).

terminal ⇐ ¬boardopen.
terminal ⇐ line(R).
goal(R,100) ⇐ line(R)).
...
```

Information about the board is stored as a set of forever holding facts, while the time for making a move is determined by the lifetime of the `control` fact. State update is performed only in two cases, after the current player performs a move – then the board should be accordingly updated and control switched to the other player; or after the current player exceeds timeout, which cause expiration of `control` and immediate end of the game.

Other terminal and goal conditions, omitted in the above listing, remain as in the standard Tic-Tac-toe, with `boardopen` holding when there is some blank space left on the board, and `line(R)` holding when player `R` composed a line of his symbols. It is worth to note that in the presented game translation (which is not the only one possible) there is no need for the special `NOOP` move by a player without control.

Example 2. Chess clock

Instead of allocating a constant time for every move, rtGDL makes it possible to implement a chess clock, and count accumulated move time for each player.

```
role(white).
role(black).
init(infinity,timer(R,120.0)) ⇐ role(R).
init(infinity,control(white)).
init(120.0,clock).

next(infinity,timer(R,T))
    ⇐ does(R,M)
    ∧ true(T,clock).
next(infinity,timer(S,T))
    ⇐ true(infinity,timer(S,T))
    ∧ does(R,M)
    ∧ distinct(R,S).

next(T,clock)
    ⇐ true(infinity,timer(S,T))
    ∧ does(R,M)
    ∧ distinct(R,S).

terminal ⇐ expired(clock).
goal(R,0)
    ⇐ expired(clock)
    ∧ true(infinity,control(R)).
```

The remaining times for players are stored in the `timer` relation, while the passing time is counted using the lifetime of the `clock`. After every turn, the remaining `clock` time is copied to the `timer` of the player with control, and it is set as the remaining time of the other player.

Example 3. Self-appearing and disappearing objects

The following rules specify an entity that appears and disappears in constant amounts of time.

```
init(1.0,appear(cheshireCat))

next(3.0,disappear(C)) ⇐ expired(appear(C)).
next(T,disappear(C)) ⇐ true(T,disappear(C)).

next(1.0,appear(C)) ⇐ expired(disappear(C)).
next(T,appear(C)) ⇐ true(T,appear(C)).
```

Example 4. Respawning objects

Slightly more complicated case describes respawning objects, e.g. items lying in the game area, reappearing some time after a player takes them, which is very popular concept in FPS games.

```
init(infinity,onMap(2,3,yellowArmor,5.5))
init(infinity,onMap(11,11,redArmor,10.0))
```

```
3
4  next(T,toRespawn(X,Y,A,T))
5      ⇐ true(infinity,onMap(X,Y,A,T))
6      ∧ playerAtPosition(X,Y).
7  next(T,toRespawn(X,Y,A,S))
8      ⇐ true(T,toRespawn(X,Y,A,S)).
9
10 next(infinity,onMap(X,Y,A,T))
11     ⇐ true(infinity,onMap(X,Y,A,T))
12     ∧ ¬playerAtPosition(X,Y).
13 next(infinity,onMap(X,Y,A,T))
14     ⇐ expired(toRespawn(X,Y,A,T))
```

Example 5. Self-moving objects

Another example contains an object traveling through the discrete space grid with fixed speed and refresh rate (`plus` is the $+$ relation defined on a finite subset of natural numbers).

```
1  speed(xwing,2,2,1)
2  init(infinity,space(35,86,40,xwing))
3  init(0.5,refresh)
4
5  next(0.5,refresh) ⇐ expired(refresh)
6  next(T,refresh) ⇐ true(T,refresh)
7
8  next(infinity,space(F,G,H,S))
9      ⇐ true(infinity,space(X,Y,Z,S))
10     ∧ expired(refresh)
11     ∧ speed(S,A,B,C)
12     ∧ plus(X,A,F)
13     ∧ plus(Y,B,G)
14     ∧ plus(Z,C,H).
15 next(infinity,space(X,Y,Z,S))
16     ⇐ true(infinity,space(X,Y,Z,S))
17     ∧ ¬expired(refresh).
```

Example 6. Player ordering time-taking events

In many games, after giving an order, the player is free to make other actions, while the execution of the order needs some time to complete. As an example of such situation we present a partial code of an RTS game where the player can order to build a barracks or a blacksmith (`greaterEqual` is the $\geq$ relation defined on a finite subset of natural numbers).

```
1  cost(barracks,160,6.0).
2  cost(blacksmith,140,7.0).
3  init(infinity,gold(500)).
4
5  legal(player,build(B))
6      ⇐ true(infinity,gold(G))
7      ∧ cost(B,F,T)
8      ∧ greaterEqual(G,F).
9  next(T,underConstruction(B))
10     ⇐ does(player,build(B))
11     ∧ cost(B,G,T).
12 next(T,underConstruction(B))
13     ⇐ true(T,underConstruction(B)).
14
15 next(infinity,constructed(B))
16     ⇐ expired(underConstruction(B)).
17 next(infinity,constructed(B))
18     ⇐ true(infinity,constructed(B)).
```

6 Real-time GDL with Imperfect Information

Following the extension from GDL to GDL-II proposed by Thielscher in [Thielscher, 2010], we can also drop the deterministic and perfect information restrictions. Below we sketch the rules of another GDL extension, called rtGDL-II, which is a combination of Real-time GDL and GDL with Incomplete Information, and in consequence, the most universal language in GDL family.

6.1 Nondeterminism

The nondeterminism introduced by GDL-II rules relies on the special `random` role operated by the Game Manager. Real-time GDL can be extended in a similar way, however in this case the randomness may be introduced at two dimensions. We may describe not only *which* action is to be drawn (like in GDL-II), but also *when* it should be drawn.

The first case can be solved in a straightforward way, by assuming that if in some state the `random` role has a non-empty set of legal moves, then immediately one of these moves should be made with uniform probability and the state update should be performed. Such usage of randomness is suitable for generating discrete random variables, e.g. rolling a dice or shuffling a deck of cards.

For the latter case we may adopt the following solution. If the `random` player's move drawn by the Game Manager is a relation with arity greater then 0, and the first argument of this relation is a non-negative real number t, then the state update should be performed after a random time t' obtained using the continuous uniform distribution $\mathcal{U}(0,t)$.

As an example consider four legal actions of `random`: `wait`, `move(right)`, `shot(0,pistol)`, `shot(2,blaster)`. Every action has a 25% chance to be chosen. If one of the first three actions will is to be made, then the Game Manager should perform state update immediately. In the remaining case, a random value t' generated with distribution $\mathcal{U}(0,2)$ should be drawn. After the time t', if the move is still legal, the state update should be performed. In the case when the move is no longer legal (due to the players' actions or some other events) no state update is performed, and the action is wasted.

6.2 Imperfect Information

In orded to introduce imperfect information we follow directly Thiescher's approach. The relation `sees` serves as a container for players' percepts, and the Game Manager as a feedback for the player sends his percepts instead of the joint move.

However, as the game is not turn-based any more, the Game Manager cannot simply send everyone messages after obtaining some player's move, because this will give everyone a clue that someone made a move. We may want to avoid this in games where just knowing that other player made an action is a crucial piece of information.

Therfore we may adopt the following solution. For every player two sets of percepts are computed. One containing percepts after updating a state according to the game rules, and the other with the percepts from the same moment, but with the assumption that the state update did not occur. The latter is possible, because apart from the players' actions the Game Manager can predict the state in any future moment. A message containing the updated percept is send to the player only in the case when this two sets differ.

7 Conclusion

With introduction of GDL-II, it was claimed that the GDL language can be considered complete [Schiffel and Thielscher, 2014], and additional elements can only serve for simplifying description or will be forcing setting extensions far beyond the concept of General Game Playing (e.g. open-world games or physical games like in General Video Game Playing [Perez *et al.*, 2015]).

The real-time extension presented in this paper preserves the core idea of GDL – a purely logical way of reasoning about the state and a Game Manager based execution model. At the same time it is a larger extension than GDL-II, allowing a game to have an infinite number of states and the players to have an infinite number of actions. By introducing time based events and by giving relevance to the move ordering it makes it possible to describe properly many real world situations. A good example is the game of chicken, which in continuous-time space takes a deeper meaning.

Extending general games with real-time events allows to the modeling of elements of the popular computer games, which are currently used as a test-bed for dedicated AI players, e.g. Unreal Tournament 2004 [Hingston, 2010], TORCS [Loiacono *et al.*, 2010] or Super Mario Bros [Togelius *et al.*, 2013]. In some cases like in Starcraft [Ontanón *et al.*, 2013], where real-world physics is minimized, it theoretically allows modeling the entire game.

Beyond the usage in General Game Playing, the correlations between the GDL and game theory [Thielscher, 2011b] and Multiagent Systems [Schiffel and Thielscher, 2010] are often pointed out. A goal for GDL is to become a universal description language which can describe as large class of game-like problems as possible, at the same time remaining compact, high-level and machine-processable. We presented the next step into such a generalization of problems description, which brings the class of games covered by the GDL family closer to real-time game theory (e.g. Differential Games [Isaacs, 1999], continuous time repeated games [Bergin and MacLeod, 1993], extensive games in continuous time [Simon and Stinchcombe, 1989]) and Real Time Multiagent Systems [Julian and Botti, 2004].

In this paper we have introduced a new language rtGDL with possible extension to rtGDL-II, and addressed a new ambitious challenge for General Game Playing systems. Taking into account response time of message arrival, creates for the players an "act-wait dilemma", i.e. given more computation time, the game tree could be explored better, however this may cause some promising paths to no longer be available due to the in-game events or the other players actions. This scenario requires developing new general-case solutions, like for example usage of real-time Monte Carlo Tree Search [Pepels *et al.*, 2014]. It makes real-time general gaming a very hard, but interesting area of further General Game Playing research.

Acknowledgements

We thank Steve Draper for reading the first draft of the paper and many valuable remarks.

This work was supported by Polish National Science Centre grants No 2014/13/N/ST6/01817 and No 2012/07/B/ST1/03318.

References

[Apt *et al.*, 1988] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148. 1988.

[Bergin and MacLeod, 1993] James Bergin and W. Bentley MacLeod. Continuous time repeated games. *International Economic Review*, pages 21–37, 1993.

[Björnsson and Schiffel, 2013] Yngvi Björnsson and Stephan Schiffel. Comparison of GDL Reasoners. In *Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, pages 55–62, 2013.

[Finnsson, 2012] Hilmar Finnsson. Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1550–1556, 2012.

[Geißer *et al.*, 2014] Florian Geißer, Thomas Keller, and Robert Mattmüller. Past, Present, and Future: An Optimal Online Algorithm for Single-Player GDL-II Games. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pages 357–362, 2014.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[Genesereth and Björnsson, 2013] Michael Genesereth and Yngvi Björnsson. The International General Game Playing Competition. *AI Magazine*, 34(2):107–111, 2013.

[Genesereth and Fikes, 1992] Michael Genesereth and Richard E. Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Technical Report LG-1992-01, Stanford Logic Group, 1992.

[Genesereth and Thielscher, 2014] Michael Genesereth and Michael Thielscher. *General Game Playing*. Morgan & Claypool, 2014.

[Genesereth *et al.*, 2005] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26:62–72, 2005.

[Hingston, 2010] Philip Hingston. A new design for a turing test for bots. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 345–350. IEEE, 2010.

[Isaacs, 1999] Rufus Isaacs. *Differential games: a mathematical theory with applications to warfare and pursuit, control and optimization*. Courier Corporation, 1999.

[Julian and Botti, 2004] Vicente Julian and Vicent Botti. Developing real-time multi-agent systems. *Integrated Computer-Aided Engineering*, 11(2):135–149, 2004.

[Kowalski and Szykuła, 2013] Jakub Kowalski and Marek Szykuła. Game Description Language Compiler Construction. In *AI 2013: Advances in Artificial Intelligence*, volume 8272 of *LNCS*, pages 234–245. 2013.

[Lloyd and Topor, 1986] John W. Lloyd and R. W. Topor. A basis for deductive database systems II. *The Journal of Logic Programming*, 3(1):55–67, 1986.

[Loiacono *et al.*, 2010] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A Pelta, Martin V Butz, Thies D Lonneker, Luigi Cardamone, Diego Perez, Yago Sáez, et al. The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131–147, 2010.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General Game Playing: Game Description Language Specification. Technical Report LG-2006-01, Stanford Logic Group, 2006.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Ontanón *et al.*, 2013] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in Starcraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4):293–311, 2013.

[Pepels *et al.*, 2014] Tom Pepels, Mark HM Winands, and Marc Lanctot. Real-time Monte Carlo Tree Search in Ms Pac-Man. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(3):245–257, 2014.

[Perez *et al.*, 2015] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, S Lucas, Adrien Couëtoux, Jerry Lee, C Lim, and Tommy Thompson. The 2014 General Video Game Playing Competition. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2015. To appear.

[Pitrat, 1968] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress*, pages 1570–1574, 1968.

[Schiffel and Thielscher, 2010] Stephan Schiffel and Michael Thielscher. A Multiagent Semantics for the Game Description Language. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 44–55. 2010.

[Schiffel and Thielscher, 2014] Stephan Schiffel and Michael Thielscher. Representing and Reasoning About the Rules of General Games With Imperfect Information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

[Simon and Stinchcombe, 1989] Leo K. Simon and Maxwell B. Stinchcombe. Extensive Form Games in Continuous Time: Pure Strategies. *Econometrica*, 57(5):1171–1214, 1989.

[Tak *et al.*, 2012] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Björnsson. N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.

[Thielscher and Zhang, 2010] Michael Thielscher and Dongmo Zhang. From general game descriptions to a market specification language for general trading agents. In *Agent-Mediated Electronic Commerce. Designing Trading Strategies and Mechanisms for Electronic Markets*, pages 259–274. 2010.

[Thielscher, 2010] Michael Thielscher. A General Game Description Language for Incomplete Information Games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 994–999, 2010.

[Thielscher, 2011a] Michael Thielscher. GDL-II. *Künstliche Intelligenz*, 25:63–66, 2011.

[Thielscher, 2011b] Michael Thielscher. The General Game Playing Description Language is Universal. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1107–1112, 2011.

[Togelius *et al.*, 2013] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N. Yannakakis. The Mario AI Championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

Discounting and Pruning for Nested Playouts in General Game Playing

Michael Schofield, Tristan Cazenave, Abdallah Saffidine, Michael Thielscher
School of Computer Science and Engineering
The University of New South Wales
{mschofield, abdallahs, mit}@cse.unsw.edu.au
LAMSADE - Université Paris-Dauphine
cazenave@lamsade.dauphine.fr

Abstract

In the field of General Game Playing (GGP) there is much emphasis on the use of the Monte Carlo playout as an evaluation function when searching intractable game spaces. This facilitates the use of statistical techniques like MCTS and UCT[1], but it requires significant processing overhead. We seek to improve the quality of information extracted from the Monte Carlo playout in three ways. Firstly by nesting the evaluation function inside another evaluation function, secondly by measuring and utilizing the depth of the playout, and thirdly by incorporating pruning strategies that eliminate unnecessary searches and avoid traps[2]. We present a formalism for a Nested Evaluation Function along with a move selection strategy that incorporates a form of discounting and pruning based on playout depth. We show experimental data on a variety of two-player games from past GGP competitions and compare the performance of a Nested Player against a standard, optimised UCT player.

1 Introduction

General Game Playing (GGP) is concerned with the design of AI systems able to take the rules of any game described in a formal language and to play that game efficiently and effectively [Genesereth *et al.*, 2005]. This area of research is growing with much emphasis on improving the AI systems ability to play games with intractable search spaces by extracting as much "insight" from the game as possible.

We focus our attention on the use of the Monte Carlo technique in GGP. This technique is domain independent, that is, the player does not need to construct a different evaluation function from each game. As such it provides a particularly suitable foundation for GGP systems to play previously unknown games without human intervention. The technique is intuitive and simple to implement. Moreover, it can provide an estimated probability distribution for the game outcomes and hence form the basis of some more advanced statistical techniques. But it also has limitations. The principle limitations are that it has a high cost and the results derived from Monte Carlo playouts[3] are predicated on the assumption that all of the roles in the real game select their move randomly. They do not.

This raises a fundamental question: how do we take a simple random technique and improve the quality of the information it produces without a similar increase in cost, specifically in the context of GGP?

In this paper we offer a way to improve cost-effectiveness by improving the quality of the information extracted from the playouts. We achieve this by implementing three different techniques in concert. Firstly we implement a nested playout such that the higher level playouts are not random but heuristically guided, thereby improving the quality of the terminal value. Secondly we consider the depth of the playouts as a measure of the "value of information" in much the same way as discounting is used in other forms of modeling. Thirdly we prune the search to eliminate wasted effort, with special emphasis on avoiding traps, ie. moves that look promising, but eventually fail.

1.1 Clarification

We use several terms that seem similar, but are not. So we offer these clarifications:

- Nested Player - Any GGP player that uses another set of players to calculate its move evaluation function;
- Nested Playout - Any playout using a nested player(s); and
- Monte Carlo Playout - A playout where all move selections are made randomly.

We use the term "search" to describe a special instance of a "tree search". That is, a tree search limited to a depth of one. In other words, we are choosing a move from the list of legal moves without constructing a game tree. And so, all of our playouts begin at depth = 1, with no expansion, no exploration, and no exploitation[4]. This simple form of search is shown in Figure 1.

[1] Monte Carlo Tree Search and Upper Confidence bound applied to Trees.

[2] Any move that looks promising but is not, especially when it takes a long time to reveal the truth.

[3] even UCT exploration uses decision values derived originally from Monte Carlo playouts.

[4] each phases in various forms of Monte Carlo Tree Searches

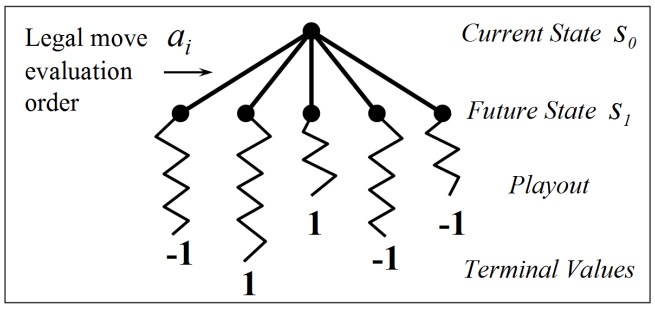


Figure 1: A search using playout terminal values to choose the next legal move a_i.

1.2 Background

The Nested Monte Carlo Search (NMCS) algorithm was proposed as an alternative to single-player Monte Carlo Tree Search (MCTS) for single-player games in [Cazenave, 2009]. NMCS lends itself to many optimizations and improvements, and it has been successful in many single-agent problems [Cazenave, 2010; Akiyama *et al.*, 2010]. In particular, it lead to a new record solution to the Morpion Solitaire mathematical puzzle.

The NMCS technique inspired the Nested Rollout Policy Adaptation algorithm which enabled further record establishing performances in similar domains [Rosin, 2011]. [Méhat and Cazenave, 2010] compare NMCS and UCT for single player games with mixed results, they explore variants of UCT and NMCS and conclude that neither one is a clear winner.

[Pepels *et al.*, 2014] have shown in the context of MCTS that more information than a binary outcome could be extracted from a random playout, even when very little domain knowledge is available. In particular, the outcome of a short playout might be more informative than that of a longer one because fewer random actions have taken place.

The idea of nesting searches of a certain type has been used to distribute the Proof Number Search algorithm over a cluster [Saffidine *et al.*, 2011].

[Chaslot *et al.*, 2009] took advantage of the nesting concept in their Meta-MCTS approach to opening book generation. The book was built through MCTS using self-play games by standard MCTS players in terms of playouts.

Yet another type of nesting was explored for the RecPIMC (Recursive Perfect Information Monte Carlo) search as a way to alleviate the strategy fusion and non-local dependencies problems exhibited by PIMC in imperfect information games [Furtak and Buro, 2013].

Finally we have the [Baier and Winands, 2013] work combining or "nesting" a MiniMax subsearch inside an MCTS with the intent of avoiding traps in tactical situations[5]. They offer three ways to augment the MCTS process with localised MiniMax searches specifically designed to minimise wasted effort.

[5]as distinct from strategic traps, see [Ramanujan *et al.*, 2010]

1.3 Contribution

The NMCS algorithm has been used in a variety of single agent domains and the concepts of discounting and pruning are not new.

We extend the use of NMCS to the general case, implementing it with two-player turn-taking games with win/lose outcomes, and implementing discounting and pruning in a practical way. In this paper, we describe how NMCS can be adapted for these games and propose heuristics that improve its performance.

Our main contributions are as follows:

- A framework for implementing a Nested Playout in two-player win/lose games;
- An improvement to the quality of information produced by the Nested Playout via a discounting heuristic;
- Pruning techniques that improve the cost-effectiveness of the Nested Playout;
- Implementation of a Player using Nested Playouts; and
- Experimental data evaluating a Nested Player for commonly played two-player, win/lose games.

The resulting player will compete favourably with a UCT player that has been optimised for best performance[6]. Experimental results show when the Nested Player is superior to the standard UCT Player and offer insights into why this is so.

2 Nested Playouts

We create a nested playout by wrapping one player around another, simpler, version of itself. At its core we still use a random move selection policy[7]. The level of nesting can be increased by wrapping yet another version around the outside. The only limitation is that the computational costs increase exponentially as the level of nesting increases, the benefit is that the quality of the information from the terminal value of the nested playout also improves.

2.1 Formalism

In our formalism for the Nested Playout we start by adopting a notation for finite games in extensive form by extending the definitions given in [Thielscher, 2011] following the style set out in [Schofield and Thielscher, 2015].

The Game

Let $G = \langle S, R, A, v, \delta \rangle$ be a game determined by a GDL description:

- S is a set of states and R is a set of roles in the game, additionally we use $s_0 \in S$ for the initial state, T for terminal states, and $D = S \backslash T$ for decision states;
- A is a set of moves in the game, and $A(s, r) \subseteq A$ is a set of legal moves, for role $r \in R$ in state $s \in S$;
- $v : T \times R \to \mathbb{R}$ is the payoff function on termination; and
- $\delta : D \times A^{|R|} \to S$ is the joint move successor function.

[6]Optimised for each different game tested so as to play that game as well as it could

[7]based on traditional Monte Carlo playouts

Move Selection Policy
In order for a game to be played out to termination we require a move selection policy π for each role, being an element of the set of all move selection policies Π:

- $\pi \in \Pi : D \times R \rightarrow \phi(A)$ is a move selection policy expressed as a probability distribution across A;
- $\vec{\pi} : \langle \pi_1, \ldots, \pi_{|R|} \rangle$ is a move selection policy tuple; and[8]
- $play : D \times \Pi^{|R|} \rightarrow T$ is the playout of a game to termination according to the given move selection policies.

Move Evaluation Function
Move selection requires an evaluation function $eval()$. We play out the game according to a move selection policy and use the terminal value as a measure of utility:

- $\langle \vec{a}_{-r}, a_r \rangle = \langle a_1 \ldots a_r \ldots a_{|R|} \rangle$ is a move vector containing a specific move a_r for role $r \in R$[9];
- $eval : D \times \Pi^{|R|} \times R \times \mathbb{N} \rightarrow \mathbb{R}$;
- $eval(d, \vec{\pi}, r, n) = \frac{1}{n} \sum_1^n v(play(d, \vec{\pi}), r)$ evaluates the node $d \in D$ using the policies in $\vec{\pi}$ and n playouts;
- $eval(\delta(d, \langle \vec{a}_{-r}, a_{ri} \rangle), \vec{\pi}, r, n)$ is the evaluation of move $a_{ri} \in A(d, r)$ by role r; and
- $a_r = argmax_{a_{ri}}[eval(\delta(d, \langle \vec{a}_{-r}, a_{ri} \rangle), \vec{\pi}, r, n)]$ is the selection process for making a choice.

Nested Evaluation
From the definitions above we take the move selection policy $\pi : D \times R \rightarrow \phi(A)$ of the **parent** player as the maximisation of the evaluation function $eval()$ using move selection policies of the **child** player. This is the basis for a nested playout.

Definition 1. Let the move selection policy for a nested evaluation be defined as:

- $\pi' : a_r = argmax_{a_{ri}}[eval(\delta(d, \langle \vec{a}_{-r}, a_{ri} \rangle), \vec{\pi}, r, n)]$; and
- $\pi'' : a_r = argmax_{a_{ri}}[eval(\delta(d, \langle \vec{a}_{-r}, a_{ri} \rangle), \vec{\pi'}, r, n)]$.

Note that if π were the random move policy, then π' would be a simple Monte Carlo playout; therefore we adopt the following nomenclature for our players:

- NMC(0) is the random player;
- NMC(1) is a simple Monte Carlo player with its move policy based on Monte Carlo playouts; and
- NMC(2) is a nested player using NMC(1) playouts to set its move selection policy, etc..

For example; an NMC(2) player would evaluate each of the 22 opening moves in Breakthrough by playing out a full game using an NMC(1) player for each roles. The two NMC(1) players would, in their turn, use Monte Carlo playouts for each move choice for the length of the game. It is easy to see how the computational cost grows exponential.

[8] we could also say $\vec{\pi}_r$ is the policy tuple used by role r to model all roles behaviour. Our treatment does not use this refinement as all role models are the same, so it is omitted.

[9] this is simplified as we are evaluating turn taking games and the other moves are noop.

3 Heuristic Improvements

The challenge is to improve the cost-effectiveness of the nested playout, and playouts in general. We use the following logic to achieve this;

- A nested playout improves the quality of the evaluation function, but increases the computational cost;
- Using the playout depth for discounting improves the evaluation function quality without increasing cost;
- Using the playout depth facilitates the use of search pruning; and
- Search pruning reduces the cost of the evaluation function without reducing its quality.

3.1 Discounting

When using Monte Carlo playouts, it is common practice to consider only the terminal value $v(t, r)$, from the playout. Here we follow the lead of [Finnsson and Björnsson, 2008] and also consider n, the depth to termination.

Using The Playout Depth
Let $G = \langle S, R, A, v, do \rangle$ be a GDL game described in the previous section, then:

- Without loss of generality, we set the range of the terminal values to $-1 \leq v(t, r) \leq 1$;
- With no prior knowledge, the expected value for a_{ri} is $E(a_{ri}) = e$, ie. all moves have the same expected value;
- Let the estimated branching factor for the sampled branch be $bf \geq 1$, and the estimated depth (number of moves) of the branch be n and $|T| = bf^n$;
- After a single playout revealing a terminal value $v(t, r)$ the expected value of making the same move is $E(a_{ri}) = [(bf^n - 1) \times e + v(t, r)]/bf^n$; and
- The increment is $\delta E(a_{ri}) = (1/bf)^n \times (v(t, r) - e)$;

In many games the branching factor bf may be high, say 30, but as we repeatedly playout out a game there are only a few moves in each round that offer a realistic chance of victory.

In these informed playouts the effective branching factor becomes small, say 2, and the term $(1/bf)^n$ begins to look like a discount factor. Hence we use the term Discounting when considering the depth n of the playout.

Definition 2. Let the move selection policy for our Nested Playout be defined by maximising the increment in the expected value $\delta E(a_{ri}) = (1/bf)^n \times (v(t, r) - e)$.

In operation this is simplified to:

- Maximise $v(t, r)$;
- Settling ties;
  - Case: $max(v(t, r)) < e$, Maximise n;
  - Case: $max(v(t, r)) = e$, Do nothing;
  - Case: $max(v(t, r)) > e$, Minimise n;
- Settle all ties randomly.

Note: In the uninformed case $e = 0$; ie. a drawn game.

The discounting heuristics turns a win/loss game into a game with a wide range of outcomes by having the Max

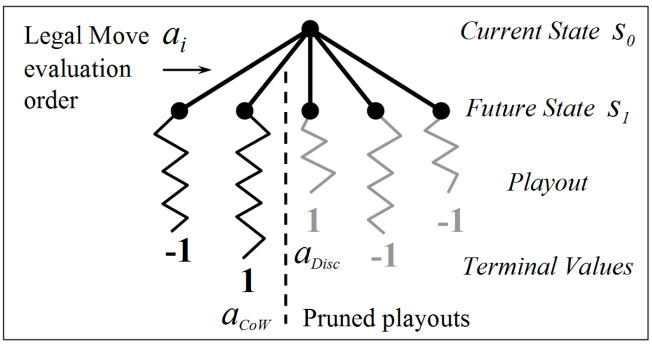


Figure 2: A decision where different heuristics lead different choices, CoW chooses early and prunes a_{Disc}.

player preferring short wins to long wins, and long losses to short losses. The intuition here is to win as quickly as possible so as to minimize our opponent's chance of finding an escape, and to lose as slowly as possible so as to maximize our chance of finding an escape.

A convenient implementation trick is to set a value of $score = v(t,r)/n$ so that a win in depth n moves is $1/n$ and a loss in n moves is $-1/n$. This way, the ordering between game outcomes corresponds exactly to the ordering between the scores considered as real numbers.

Note that our discounting heuristic should not be confused with Steinhauer's discount rate proposed for MCTS [Steinhauer, 2010]. We discount on the length of the playout whereas the latter discounts on how long ago it was performed.

3.2 Pruning

From the classical alpha-beta search [Knuth and Moore, 1975] to the more recent Score-Bounded MCTS adaptation [Cazenave and Saffidine, 2010], two-player games usually lend themselves quite well to pruning strategies. Adapting Nested Playouts to two-player games also gives pruning opportunities that were absent in the single-player case.

Cut on Win

The first pruning strategy, Cut on Win (CoW), hangs on two features of two-player win/loss games. First, we know the precise value of the best outcome of the game. Second, these values are reached regularly in playouts.

The CoW strategy consists of evaluating the moves randomly and selecting the first move that returns the maximum value from its evaluation(s), thus:

- The set of move is evaluated randomly; and
- When $eval()$ returns the maximum value then a choice is made and the remaining evaluations abandoned.

Proposition 1. *Assume that whenever a state is evaluated using a nested playout, move choices are randomly presented in the same order.*

Then NMC(n) with no discounting and no pruning returns the same move as NMC(n) with no discounting and CoW.

Note that when discounting is used, NMC(n) with CoW is not guaranteed to pick the same move as NMC(n). As a counter-example, Figure 2 presents a decision in which NMC(n) with CoW and discounted NMC(n) would select different moves.

Pruning on Depth

The second pruning strategy, Prune on Depth (POD) exploits the use of discounting and the richer outcome structure. Remember the convenient trick of using $score = v(t,r)/n$ to evaluate the game outcomes. With Pruning on Depth we maximise $v(t,r)$ then minimise n thus:

- The set of move is evaluated randomly;
- n is the playout depth;
- $n_{min} = min(n)$ is the minimum depth of any playout to return the maximum score; and
- When $depth(eval()) = n_{min}$ the playout is terminated, returning the goal value of -1.

Put simply, we prune states deeper than the shallowest win.

Proposition 2. *Assume that whenever a state is evaluated using a nested playout, move choices are randomly presented in the same order.*

Then NMC(n) with discounting and no pruning search returns the same move as NMC(n) with discounting and PoD.

Cascade Pruning

The PoD heuristic is like shallow pruning in the alpha-beta algorithm. We now propose a third pruning strategy which is like deep alpha-beta pruning, *Cascade Pruning*.

Whereas in PoD the playout length would only be compared to the length of sibling playouts, in cascade pruning the bound on the length is shared between the parents and the children. That is, the pruning information is cascaded downward from one nesting level to the next. This provides for additional pruning opportunities compared to PoD, but the safety with respect to discounted searches is lost.

The loss of safety can be illustrated in the Breakthrough Challenge in Figure 4. A Monte Carlo playout will often arrive at a favorable outcome via a less than optimal path. For example; Black can counter b6-a7 with b8-a7, but a Monte Carlo playout might make some trivial moves first. If Cascade Pruning is implemented those trivial moves can cause an early termination of the playout returning a loss instead of a safe defense. This impacts the choices made by the higher level player.

3.3 The Trade-off of Unsafe Pruning

Unlike the classical alpha-beta pruning algorithm, the CoW and Cascade Pruning techniques described previously are *unsafe* when using discounting: they may lead to a better move being overlooked.

Unsafe pruning methods are common in the game search community, for instance null move and forward pruning [Smith and Nau, 1994], and the attractiveness of a new method depends on the speed versus accuracy trade-off.

The quality of the choices made by the NMC(3) player is built on both the quality of the choices made by the embedded NMC(2) player and the number of sub-searches that can be performed in a given amount of time. Both aspects can be affected by the pruning policy.

3.4 Embedded MiniMax Search

A close examination of an NMC(n) playout reveals a depth limited MiniMax search being conducted at every step of the playout, very similar to the MCTS-MR proposed by [Baier and Winands, 2013].

Consider an NMC(3) playout being used in a game of Breakthrough for the initial move. The game is at the initial state, $n = 0$. The NMC(3) playouts start at each of the games states of $n = 1$; the embedded NMC(2) playouts start at each of the game states of $n = 2$; and all of the embedded NMC(1) playouts start at each of the game states of $n = 3$.

A similar phenomenon occurs at the bottom of the playout. A depth limited MiniMax at the termination ensures the most correct terminal value is being returned.

4 Experimental Results

4.1 Discounting in TicTacToe

To illustrate discounting we look at the game of TicTacToe in Figure 3. The X player has an opportunity to make the decisive move a_3 = `(does x (mark 3 1))`.

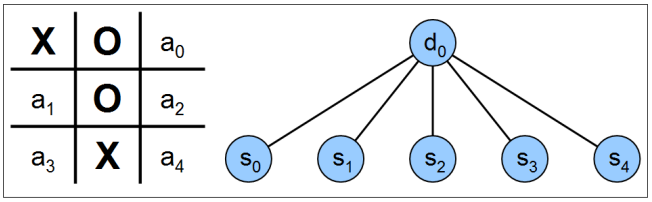


Figure 3: A partially played game of TicTacToe where the X player has the decisive move a_3.

We show the move selection choices made by the NMC(3) player for a 1000 game sample. Terminal values are in the range $-1 \leq v(t,r) \leq 1$ and the values in brackets are (min, max) values.

No Discounting

a_i	$v(t,x)$	$Freq$
a_0	(0, 0)	0
a_1	(0, 1)	176
a_2	(0, 1)	123
a_3	(1, 1)	575
a_4	(0, 1)	126

Discounting

a_i	$v(t,x)$	n	$Freq$
a_0	(0, 0)	(4, 4)	0
a_1	(0, 0)	(4, 4)	0
a_2	(0, 1)	(4, 4)	0
a_3	(1, 1)	(2, 2)	1000
a_4	(0, 1)	(4, 4)	0

Table 1: Results from 1000 games played by a NMC(3) player illustrating the effect that Discounting on depth n has on the frequency of each choice.

Note that, without discounting, moves a_1, a_2 and a_4 are selected occasionally in a tie break. Whereas, with discounting the value of $depth = n$ decides equal values of $v(t,x)$. Also, the embedded NMC(2,Disc) selecting moves for the O player is forcing a draw after move a_1.

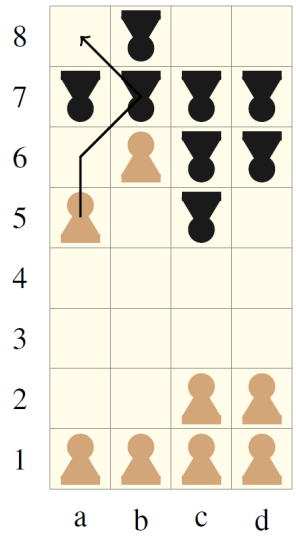


Figure 4: Partially played game of breakthrough. White has a winning move, a5-a6, but also has two incorrect moves b6-a7, b6-c7. The challenge is to avoid the incorrect moves.

4.2 Breakthrough Challenge

As an illustrative example, we look at the game of Breakthrough being played in Figure 4 [Gudmundsson and Björnsson, 2013].[10] This position was used as a test case exhibiting a trap that is difficult to avoid for a plain UCT player. The correct move for White is a5-a6, however b6-a7 and b6-c7 initially look strong.

To better understand the effect of the various heuristics described in the previous sections, we run multiple NMC(3) searches with different parameter settings, starting from the position displayed in Figure 4. For each parameter setting, we record the number of states visited during the search and the likelihood of selecting the correct initial move and present the results in Table 2.

Each entry in the table was an average of 900 games, and the 95% confidence interval on standard error of the mean is reported for each entry. A setting of the type CoW(1) is to be understood as Cut on Win pruning heuristic was activated at nesting level 1 but not at any higher level.

A number of facts can be observed in Table 2:

- If the discounting heuristic is disabled, then the best move is selected with probability around 0.11, but it is identified with probability around 0.63 whenever the discounting heuristic is enabled;
- Both CoW and PoD significantly reduce the number of states explored in a search, and CoW provides more reduction;
- The performance of a non-discounted search is not significantly affected by CoW, as predicted by Proposition 1;

[10] Knowing the rules of Breakthrough is not essential to follow our analysis. However, the reader can find a description of the rules in [Saffidine *et al.*, 2011; Gudmundsson and Björnsson, 2013].

Discounting	Pruning	States Visited(k)	Freq(%)
No	None	$4,459 \pm 27$	11.9 ± 2.2
No	CoW(1)	$1,084 \pm 8$	12.3 ± 2.6
No	CoW(1, 2)	214 ± 2	10.9 ± 2.0
No	CoW(1, 2, 3)	25 ± 1	9.8 ± 2.0
Yes	None	$2,775 \pm 26$	64.1 ± 3.4
Yes	PoD(1)	$1,924 \pm 20$	64.7 ± 3.5
Yes	PoD(1, 2)	$1,463 \pm 16$	58.6 ± 3.5
Yes	PoD(1, 2, 3)	627 ± 19	62.4 ± 3.3
Yes	Cascade	322 ± 16	64.6 ± 2.4

Table 2: Results from 900 games played using a variety of options; showing discounting options, pruning options, the number of states visited and the correct move frequency.

- Similarly the performance of a discounted search is not significantly affected by PoD, as predicted by Proposition 2; and
- Also note that PoD(1,2,3) is 25 times more expensive than CoW(1,2,3), but much more accurate.

If we take a look inside the Nested Playout and examine the quality of the choices being made by the NMC(3), we see that they are built on the quality of the choices being made by the embedded NMC(2) player for both Black and White roles. We can measure this quality by considering Black's response to White's move $does(white\ move\ b6\ a7)$. Remember that this is a trap for White as it eventually fails, but the Black NMC(2) player must spring the trap for the White NMC(3) playout to "get it right".

Black must respond with $does(black\ move\ b8\ a7)$, otherwise White has a certain win. The experimental results, using similar pruning options, were:

- Base Case, $does(black\ move\ b8\ a7) = 15\%$;
- CoW, $does(black\ move\ b8\ a7) = 15\%$;
- Discounting, $does(black\ move\ b8\ a7) = 100\%$; and
- Disc. and PoD, $does(black\ move\ b8\ a7) = 100\%$.

4.3 GGP Games

We use 9 two-player games drawn from games commonly played in GGP competitions, each played on a 5×5 board.

Breakthrough and Knightthrough are racing games where each player is trying to get one their piece across the board, these two games are popular as benchmarks in the GGP community.

Domineering and NoGo are mathematical games in which players gradually fill a board until one of them has no legal moves remaining and is declared loser, these two games are popular in the Combinatorial Game Theory community.

For each of these domain, we construct a *misere* version, which has exactly the same rules but with reverse winning condition. For instance, a player wins misere Breakthrough if they force their opponent to cross the board. To this list, we add AtariGo, a capturing game in which each player tries to surround the opponent's pieces.

AtariGo is popular as a pedagogical tool when teaching the game of Go.

4.4 Performance of the playout engine

It is well known in the games community that increasing the strength of a playout policy may not always result in a strength increase for the wrapping search [Silver and Tesauro, 2009]. Still, it often is the case in practice, and determining whether some of our heuristics improve the strength of the corresponding policy may prove informative. It is also important to know how fast playouts can be performed as it will directly influence how large the MCTS can grow in a given time budget.

Table 3 addresses these two concerns. For each of the nine games of interest, we are provided with the raw speed of the engine in thousands of playouts per second, as well as the scores of a discounted nested player against a nested player without discounting for different levels of nesting. 500 games were run per match, 250 with each color.

	k.Playouts	Nesting Level		
	per second	0	1	2
Breakthrough	1641	79.6	99.6	99.4
MisereBreakthrough	1369	42.4	80.8	90.0
Knightthrough	1632	78.6	100.0	100.0
MisereKnightthrough	1337	46.0	83.2	85.8
Domineering	1406	71.2	77.0	83.8
MisereDomineering	1419	43.4	63.2	68.4
NoGo	397	62.8	76.4	83.4
MisereNoGo	409	53.2	65.6	67.2
AtariGo	123	69.6	97.2	100.0

Table 3: Performance of the playout engine: Implementation speed and win rate of NMC(n) with discounting against NMC(n) without discounting for various nesting level.

4.5 Win Rate Performance

We want to determine whether using nested rather than plain Monte Carlo playouts could improve the performance of a UCT player in two-player games in a more systematic way. We also want to measure the effect of some of the proposed heuristics, *discounting* together with PoD as well as CoW.

We therefore played a parameterised UCT(NMC(n)) player against an optimized standard UCT opponent in a variety of games. Both players are allocated the same thinking time, ranging from 10ms per move to 320ms per move[11]. For each game, each parameter setting, and each time constraint, we run a 500 games match where UCT(NMC(n)) plays as first player 250 times and we record how frequently UCT(NMC(n)) wins. The experiments were run on a 3.0 GHz PC under Linux. The results are presented in Table 4.

The first point to make is that the win/loss ratio must be taken in context. For example 50/50 might mean that each player is playing at optimal performance, or it might mean they are both playing badly.

We can notice in Table 4 that PoD and CoW improve the performance of the search over a basic UCT player. We can

[11] the experimental setup uses hard-coded games, not GDL, so is significantly faster than a GGP.

Game		n	CoW	PoD	10ms	20ms	40ms	80ms	160ms	320ms
Breakthrough		1			3.2	6.0	12.0	11.6	7.8	6.4
		1		✔	27.6	22.6	16.8	21.6	15.4	20.4
		1	✔		22.6	25.2	30.4	34.6	35.2	39.6
		2	✔		4.6	2.0	2.4	1.4	2.4	3.8
	misere	1			85.4	83.4	70.2	60.8	57.0	56.4
	misere	1		✔	91.4	95.6	97.0	97.8	98.8	98.8
	misere	1	✔		95.2	95.2	98.0	99.0	99.8	99.8
	misere	2	✔		1.0	27.6	43.6	87.0	93.2	95.6
Knightthrough		1			42.2	57.2	9.8	49.4	50.2	50.0
		1		✔	68.6	50.2	42.4	42.4	46.4	44.6
		1	✔		27.2	25.4	28.0	43.4	49.2	49.6
		2	✔		20.0	16.4	5.8	1.8	29.2	38.2
	misere	1			43.0	31.6	20.0	15.4	11.2	12.6
	misere	1		✔	54.6	72.2	80.6	88.4	94.2	98.4
	misere	1	✔		77.8	82.2	88.8	94.4	98.2	98.6
	misere	2	✔		20.8	18.6	32.2	42.2	54.0	67.0
Domineering		1			13.4	8.6	8.6	6.0	14.2	28.0
		1		✔	40.8	34.4	37.4	48.4	50.0	50.0
		1	✔		44.4	38.6	40.6	49.4	50.0	50.0
		2	✔		11.2	14.4	20.2	25.2	32.2	45.4
	misere	1			33.4	25.2	20.0	18.8	13.2	12.2
	misere	1		✔	45.4	47.2	56.8	60.2	62.8	54.2
	misere	1	✔		69.4	66.6	71.6	70.4	68.4	58.6
	misere	2	✔		37.0	45.2	45.6	51.0	57.8	53.6
NoGo		1			5.8	3.0	2.6	3.0	0.6	0.8
		1		✔	7.2	16.0	31.8	35.2	35.4	40.6
		1	✔		37.6	39.2	38.4	40.8	47.8	48.0
		2	✔		0.4	2.8	5.4	15.0	20.6	17.0
	misere	1			14.6	6.6	5.2	3.0	2.4	1.8
	misere	1		✔	17.2	25.0	38.8	51.2	48.2	48.8
	misere	1	✔		55.4	56.6	57.0	57.6	54.6	60.8
	misere	2	✔		5.2	10.6	19.4	35.6	37.2	47.8
Atari-Go		1			0.6	2.2	4.6	5.4	6.8	7.6
		1		✔	0.2	19.2	42.0	42.0	55.4	67.2
		1	✔		42.0	59.0	60.2	71.0	71.2	77.2
		2	✔		0.2	0.0	0.6	7.4	8.6	4.8

Table 4: Win percentages for a UCT(NMC(n)) player against an optimized UCT player for various settings and thinking times: "PoD" stands for *Discounting+Pruning on Depth* and "CoW" for *Cut on Win*.

also observe that for this range of computational resources, using a nested search with CoW at both levels does not seem as effective as CoW for level 1 only.

Where the statistic move towards the 50% mark we can infer that games are being played perfectly by both the reference UCT player, and by the Nested Player. Domineering, especially, appears to be an easy task for the algorithms at hand, and Knightthrough might be quite close to solved. On the other hand, the large proportion of games lost by the reference UCT player independent of the side played demonstrate that some games are far from being solved by this algorithm, for instance misere Breakthrough, misere Knightthrough, or Atari-Go are dominated by UCT. This shows that although we have used 9 games all played on boards of the same 5×5 size, the underlying decision problems were of varying difficulty.

The performance improvement on the misere version of some the games seems to be much larger than on the original versions. A tentative explanation for this phenomenon which would be consistent with a similar intuition in single-agent domains is that Nested Player is particularly good at games where the very last moves are crucial to the final score. Since the last moves made in a level n playout are based on a search of an important fraction of the subtree, comparatively fewer mistakes are made at this stage of the game than a plain Monte Carlo playout. Therefore, the estimates of a position's value are particularly more accurate for the Nested Playout than for plain Monte Carlo Playout. This is consistent with the idea[12] that the Nested Playout is performing a Depth Limited MiniMax search at the terminus of the playout.

5 Conclusion

In this paper, we have examined the adaptation of the NMCS algorithm from single-agent problems to two-outcome two-player games.

We have proposed three types heuristic improvements to the algorithm and have shown that these suggestions indeed lead to better performance than that of the naive adaptation. In particular, discounting the reward based on the playout length increases the accuracy of the nested searches, and the various pruning strategies allow to discard very large parts of the search trees.

Together these ideas contribute to creating a new type of domain agnostic search-based artificial player which appears to be much better than a classic UCT player on some games. In particular, in the games misere Breakthrough and misere Knightthrough the new approach wins close to 99% of the games against the best known domain independent algorithm for these games.

Some important improvements to the original single-player NMCS algorithm such as *memorization of the best sequence* [Cazenave, 2009] cannot be adapted to the two-player setting because of the alternation between maximizing and minimizing steps. Still, nothing prevents attempting to generalize some of the other heuristics such as All-Moves-As-First idea [Akiyama *et al.*, 2010] and the Nested Rollout Policy Adaptation [Rosin, 2011] in future work. Future work could also examine how to further generalize NMCS to multi-outcome games.

While we built our Nested Player around a purely random policy as is most common in the GGP community, our technique could also build on the alternative domain-specific pseudo-random policies developed in the Computer Go community [Silver and Tesauro, 2009; Browne *et al.*, 2012]. The interplay between such smart elementary playouts and our nesting construction and heuristics could provide a fruitful avenue for an experimentally oriented study.

[12] in section 3.4

6 Acknowledgments

This research was supported by the Australian Research Council under grant no. DP120102023. The last author is also affiliated with the University of Western Sydney.

This work was granted access to the HPC resources of MesoPSL financed by the Region Ile de France and the project Equip@Meso (reference ANR-10-EQPX-29-01) of the programme Investissements d'Avenir supervised by the Agence Nationale pour la Recherche.

References

[Akiyama *et al.*, 2010] Haruhiko Akiyama, Kanako Komiya, and Yoshiyuki Kotani. Nested monte-carlo search with amaf heuristic. In *2012 Conference on Technologies and Applications of Artificial Intelligence*, pages 172–176. IEEE, 2010.

[Baier and Winands, 2013] Hendrik Baier and Mark HM Winands. Monte-carlo tree search and minimax hybrids. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[Browne *et al.*, 2012] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[Cazenave and Saffidine, 2010] Tristan Cazenave and Abdallah Saffidine. Score bounded Monte-Carlo tree search. volume 6515 of *Lecture Notes in Computer Science*, pages 93–104, Kanazawa, Japan, September 2010. Springer.

[Cazenave, 2009] Tristan Cazenave. Nested Monte-Carlo search. pages 456–461, Pasadena, California, USA, July 2009. AAAI Press.

[Cazenave, 2010] Tristan Cazenave. Nested Monte-Carlo expression discovery. pages 1057–1058, Lisbon, Portugal, August 2010. IOS Press.

[Chaslot *et al.*, 2009] Guillaume M.J.-B. Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud, and Mark H.M. Winands. Meta Monte-Carlo Tree Search for automatic opening book generation. pages 7–12, Pasadena, California, USA, July 2009.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, volume 8, pages 259–264, 2008.

[Furtak and Buro, 2013] Timothy Furtak and Michael Buro. Recursive Monte Carlo search for imperfect information games. pages 225–232, Niagara Falls, Canada, August 2013. IEEE Press.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Gudmundsson and Björnsson, 2013] Stefan Freyr Gudmundsson and Yngvi Björnsson. Sufficiency-based selection strategy for MCTS. pages 559–565, Beijing, China, August 2013. AAAI Press.

[Knuth and Moore, 1975] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Méhat and Cazenave, 2010] Jean Méhat and Tristan Cazenave. Combining UCT and nested Monte-Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

[Pepels *et al.*, 2014] Tom Pepels, Mandy J.W. Tak, Marc Lanctot, and Mark H.M. Winands. Quality-based rewards for Monte-Carlo Tree Search simulations. volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 705–710, Prague, Czech Republic, August 2014. IOS Press.

[Ramanujan *et al.*, 2010] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. In *ICAPS*, volume 10, pages 242–245, 2010.

[Rosin, 2011] Christopher D. Rosin. Nested rollout policy adaptation for Monte Carlo tree search. In Walsh [2011], pages 649–654.

[Saffidine *et al.*, 2011] Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave. Solving Breakthough with race patterns and Job-Level Proof Number Search. volume 7168 of *Lecture Notes in Computer Science*, pages 196–207, Tilburg, The Netherlands, November 2011. Springer.

[Schofield and Thielscher, 2015] Michael Schofield and Michael Thielscher. Lifting model sampling for general game playing to incomplete-information models. In *Proc. AAAI*, Austin Texas, January 2015.

[Silver and Tesauro, 2009] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. pages 945–952, Montreal, Quebec, Canada, June 2009. ACM.

[Smith and Nau, 1994] Stephen J.J. Smith and Dana S. Nau. An analysis of forward pruning. pages 1386–1391, Seattle, WA, USA, July 1994. AAAI Press.

[Steinhauer, 2010] Janik Steinhauer. Monte-Carlo twixt. Master's thesis, Maastricht University, Maastricht, The Netherlands, June 2010.

[Thielscher, 2011] Michael Thielscher. The general game playing description language is universal. In Walsh [2011], pages 1107–1112.

[Walsh, 2011] Toby Walsh, editor. *22nd International Joint Conference on Artificial Intelligence (IJCAI)*, Barcelona, Catalonia, Spain, July 2011. AAAI Press.

Creating Action Heuristics for General Game Playing Agents

Michal Trutman
Faculty of Information Technology
Brno University of Technology
xtrutm00@stud.fit.vutbr.cz

Stephan Schiffel
School of Computer Science
Reykjavik University
stephans@ru.is

Abstract

Monte-Carlo tree search (MCTS) is the most popular search algorithm used in General Game Playing (GGP) nowadays mainly because of its ability to perform well in the absence of domain knowledge. Several approaches have been proposed to add heuristics to MCTS in order to guide the simulations. In GGP those approaches typically learn heuristics at runtime from the results of the simulations. Because of peculiarities of GGP, it is preferable that these heuristics evaluate actions rather than game positions. We propose an approach that generates heuristics that estimate the usefulness of actions by analysing the game rules as opposed to the simulation results. We present results of experiments that show the potential of our approach.

Monte-Carlo tree search (MCTS) with UCT (Upper Confidence bounds applied to Trees) [Kocsis and Szepesvári, 2006] has seen wide-spread success in recent years and is the state-of-the-art in General Game Playing (GGP) [Genesereth *et al.*, 2005] today. One reason for the success of MCTS in GGP is that MCTS can find good moves even in the absence of domain knowledge in the form of evaluation functions or heuristics. However, that does not mean that MCTS cannot benefit from heuristics, if they are available. In fact, there are several examples, such as [Finnsson and Björnsson, 2008] or [Tak *et al.*, 2012] in GGP and [Lanctot *et al.*, 2014] in game specific settings that show how heuristics can be used in MCTS to improve the performance of a game player. In General Game Playing, a program is presented with the rules of a previously unknown game and needs to play this game well without human intervention. Thus, the main problem of using domain knowledge in the form of heuristics in a General Game Playing program is, that the program must generate or learn the heuristics automatically for the game at hand.

Heuristics used in search come traditionally in the form of a state evaluation function, that is, an evaluation of non-terminal states in a game. Especially in GGP, it seems advantageous to evaluate actions instead of states. In perfect-information turn-taking games, there is no difference between the value of an action in a given state and the value of the state that is reached by executing that action. However, games in GGP can have simultaneous moves in which case the successor state depends on the actions of all players. Even in the case of turn-taking games, evaluating an action directly instead of the successor state reached by that action saves the time needed to compute one state update. In GGP, this time is often significant [Schiffel and Björnsson, 2014], unless dominated by the time for computing the heuristics. Both reasons make it beneficial to evaluate actions instead of states, especially in the context of MCTS. Previous approaches to generate action heuristics in GGP are very limited in the sense that the learned heuristics are very simple and often ignore the context (state) in which an action is executed.

In the current work, we are exploring whether more accurate action heuristics can be generated by analysing the rules of a game instead of learning them from simulation results. The paper is organized as follows: In the next section we give a brief background of the game description language and MCTS. Then we introduce our approach to generate action heuristics and, finally, we evaluate the approach and discuss the results.

Preliminaries

Game Description Language

Games in GGP are described in the so-called *Game Description Language* (GDL) [Love *et al.*, 2008]. A game description in GDL is a logic program with a number of predefined predicates and restrictions to ensure finiteness of derivations. The rules in the program can be used to compute an initial state, legal moves, successor states, terminality of states and goal values of the players. Thus, they are sufficient to simulate the game. GDL permits to describe a large range of deterministic perfect-information simultaneous-move games with an arbitrary number of adversaries. Turn-based games can be modelled by only allowing a move with no effect for players that do not have a turn (a *noop* move). Predefined predicates have a game-specific semantic, such as for describing the initial game state ($init$), detecting ($terminal$) and scoring ($goal$) terminal states, and for generating ($legal$) moves and successor states ($next$). Each game state can be represented by the set of facts that hold in the state (e.g., $cell(1, 1, b)$).

The following figure shows a partial GDL description for a variant of the game Tic Tac Toe, where the goal was reduced to build any of the two diagonal lines on the board.

```
1 role(xplayer). role(oplayer).
```

```
2 init(cell(1, 1, b)) ... init(cell(3, 3, b)).
3 init(control(xplayer)).
4 legal(W, mark(X, Y)) :- true(cell(X, Y, b)),
5     true(control(W)).
6 legal(oplayer, noop) :-
7   true(control(xplayer)).
8 ...
9 next(cell(M, N, x)) :-
10   does(xplayer, mark(M, N)),
11   true(cell(M, N, b)).
12 next(control(oplayer)) :-
13   true(control(xplayer)).
14 ...
15 diagonal(X) :- true(cell(1, 1, X)),
16   true(cell(2, 2, X)), true(cell(3, 3, X)).
17 goal(xplayer, 100) :- diagonal(x).
18 goal(xplayer, 0) :- diagonal(o).
19 terminal :- diagonal(x).
20 ...
```

Monte-Carlo Tree Search

Monte-Carlo tree search works by running complete simulations of a game, that is, repeatedly playing a simulation of a game starting at the current state and stopping when a terminal state is reached. The simulations are used to gradually build a game tree in memory. The nodes in this tree store the average reward (goal value) achieved by executing a certain action in a certain state. When the deliberation time is up, the player plays the best move in the root node of the tree. Each simulation consists of four steps:

1. *Selection:* selecting the actions in the tree based on their average reward until a leaf node of the tree is reached,
2. *Expansion:* adding one or several nodes to the tree,
3. *Playout:* playing randomly from the leaf node of the tree until a terminal state is reached,
4. *Back-Propagation:* updating the values of the nodes in the tree with the reward achieved in the playout.

Related Work

The most common modification of MCTS algorithm is MCTS with UCT [Kocsis and Szepesvári, 2006] allowing to set a trade-off between exploration and exploitation. One of the first attempts to enrich MCTS/UCT with a heuristic was a *progressive bias* added to the UCT formula to direct search according to possibly expensive heuristic knowledge in Go [Chaslot *et al.*, 2007].

There are two ways to create heuristics. First, offline heuristics rely on game analysis and feature detection before the game starts. Once the heuristic is generated, it is used throughout the game. On the other hand, online heuristics are learned and improved during game play.

Several ways were suggested of how to automatically generate heuristic offline. While [Kuhlmann *et al.*, 2006] try to build a heuristic upon detecting common game features like a game board, game pieces or quantities; [Waledzik and Mandziuk, 2014] look for more generic and game independent concepts. [Clune, 2007] uses game properties like termination, control over the board and payoff as components in his evaluation function. In [Schiffel and Thielscher, 2007] fuzzy logic is used to evaluate the goal condition in an arbitrary state and the value is used as a measure of how close the state is to a goal state. The approach is further improved by using feature discovery and was used in Fluxplayer, when winning the GGP competition in 2006.

A different approach relies on learning a heuristic online from simulations of the game. The first notable enhancement of MCTS was *Rapid Action Value Estimation* (RAVE) [Gelly and Silver, 2007], a method to speed up the learning process of action values inside the game tree. A similar technique to learn state and move knowledge was based on which state fluents mostly occur in the winning states and which moves lie on the winning paths [Sharma *et al.*, 2008]. The state-of-the-art has also been greatly advanced by *Move-Average Sampling Technique* (MAST) [Finnsson and Björnsson, 2008]. MAST is a control scheme used in the playout phase of MCTS which learns the general value of an action independent from the context the action is used in. This and other control schemes such as *Features-to-Action Sampling Technique* (FAST) [Finnsson and Björnsson, 2011], early cutoffs and unexplored action urgency [Finnsson, 2012] were used by Cadiaplayer, a successful player that won the GGP competition three times. Recently, the MAST concept was made more accurate by using sequences of actions of given length (N-grams) instead of just single actions [Tak *et al.*, 2012]. It has also been shown, that is possible to get more information from the playouts by assessing the lengths of simulations and evaluating the quality of the terminal state reached [Pepels *et al.*, 2014].

As it was shown in [Coulom, 2006], MCTS converges slowly to the true Minimax value and therefore different combinations of Minimax and MCTS were suggested. While [Coulom, 2006] use a different operator for backing up the values through the tree instead of just averaging them; [Winands *et al.*, 2010] introduce MCTS Solver, an $\alpha\beta$-search-like approach to prove correct values of fully expanded parts of the game tree in Lines of Actions. Recently, [Lanctot *et al.*, 2014] experiment with calculating approximate Minimax backups from heuristic values to further improve node selection in Kalah, Breakthrough and Lines of Action. However, the heuristic is currently built on game specific knowledge.

Generating Action Heuristic

Our idea for generation of an action heuristic is to create an action-based version of the state evaluation function described in [Schiffel and Thielscher, 2007], which uses fuzzy logic to evaluate the degree of truth of a goal condition. Turning it into an action heuristic is achieved by taking the goal condition, regressing it one step and filtering it according to an action a of a player p. This yields a new condition which – when satisfied in the current game state – allows player p to achieve the goal condition by executing action a.

Regression

Our definition of regression is based on regression in the *situation calculus* as defined in [Reiter, 2001]. Similar to situation formulas in situation calculus, we define a *state formula*

in a game as any first-order formula over the predicate, function and constant symbols of the game description with the exception of the **does** predicate and any predicate depending on **does**.

Thus, the truth value of a state formula can be determined in any state independently on the actions that players choose in that state. For example, goal$(xplayer, 100)$ is a state formula in our Tic Tac Toe game, because **goal** may not depend on **does** according to GDL restrictions. For the purpose of this paper, we only consider variable-free state formulas and game descriptions. Generalizing the proposed algorithm to non-ground game descriptions should be straight-forward.

The regression of a variable-free state formula F by one step, denoted as $\mathcal{R}[F]$, is defined recursively as follows:

- $\mathcal{R}[\text{true}(X)] = F_1 \vee F_2 \vee \ldots \vee F_n$
  where F_i are the bodies of all rules of the following form in the game description: next(X) :– F_i
- $\mathcal{R}[\text{distinct}(a_1, a_2)] = \text{distinct}(a_1, a_2)$
- If F is any atom $p(a_1, a_2, \ldots, a_n)$ other than true or distinct, then
  $$\mathcal{R}[F] = \mathcal{R}[F_1] \vee \mathcal{R}[F_2] \vee \ldots \mathcal{R}[F_n]$$
  where F_i are the bodies of all rules with head $p(a_1, a_2, \ldots, a_n)$.
- If F is a non-atomic formula then the regression is defined as follows:
  $$\mathcal{R}[F_1 \vee F_2] = \mathcal{R}[F_1] \vee \mathcal{R}[F_2]$$
  $$\mathcal{R}[F_1 \wedge F_2] = \mathcal{R}[F_1] \wedge \mathcal{R}[F_2]$$
  $$\mathcal{R}[\neg F] = \neg\mathcal{R}[F]$$

Note, that the regression of a state formula is not a state formula in general. On the contrary, replacing true(X) with the next(X) during the regression, introduces dependencies on the **does** predicate and thus the executed moves. These dependencies will be used in the following section to define a heuristic function for each action.

Algorithm

Based on the previous definition, we propose the following algorithm to generate a heuristic function for each action a of a player p. The algorithm consists of following steps:

1. Compute $\mathcal{R}[\text{goal}(p, 100)]$, the regression of goal$(p, 100)$. [1] $\mathcal{R}[\text{goal}(p, 100)]$ represents a condition on a state and actions of players that – when fulfilled – allow to reach a goal state for player p.
2. $\mathcal{R}[\text{goal}(p, 100)]$ contains conditions on actions of players. However, we want a formula that indicates when it is a good idea for player p to execute action a. To obtain such a formula, we restrict $\mathcal{R}[\text{goal}(p, 100)]$ to those parts that are consistent with $does(p, a)$. In practice this is achieved by replacing all occurrences of does(r, b) for any r and b in $\mathcal{R}[\text{goal}(p, 100)]$ as follows:
   (a) In case $p = r$ and $a = b$, the occurrence of does(r, b) is replaced with the boolean constant **true**.
   (b) In case $p = r$, but $a \neq b$, the occurrence of does(r, b) is replaced with the boolean constant `false`.
   (c) In case $p \neq r$, the condition is on an action for another player. In that case, the replacement depends on whether or not the game is turn-taking[2]. If the game has simultaneous moves, does(r, b) is replaced with the unknown value in three-valued logic. This represents, that we are not sure, which action the opponent decides to play. In case of a turn-taking game, if b is a noop action, the does(r, b) is replaced with **true**, otherwise with `false` (because r must do a noop action if p is doing a non-noop action such as a).
3. The formula is simplified according to laws of three-valued logic. In particular, any boolean values introduced by the previous replacement step are propagated up and the formula is partially evaluated.

Tic Tac Toe

Let us demonstrate, how the algorithm works on the simplified version of the game Tic Tac Toe, where the goal was reduced to build only some of the two diagonal lines. The grounded and expanded version of the goal for the player *xplayer* is

$$\begin{aligned}\big(&\text{true}(cell(1,1,x)) \wedge \text{true}(cell(2,2,x)) \wedge \\ &\text{true}(cell(3,3,x))\big) \vee \big(\text{true}(cell(1,3,x)) \wedge \\ &\text{true}(cell(2,2,x)) \wedge \text{true}(cell(3,1,x))\big)\end{aligned} \tag{1}$$

Assume, we are computing the heuristic function for the action $mark(1,1)$ for the role *xplayer*. The regression of the goal above will replace all occurrences of true(X) with the bodies of the respective next rules. For example, for true$(cell(1,1,x))$, the grounded game description contains following *next* rules with matching arguments:

```
next(cell(1, 1, x)) :- true(cell(1, 1, b)),
  does(xplayer, mark(1, 1)).
next(cell(1, 1, x)) :- true(cell(1, 1, x)).
```

To regress *true*$(cell(1,1,x))$, we replace it with the disjunction of the bodies of the *next* rules:

$$\begin{aligned}\big(&\text{true}(cell(1,1,b)) \wedge \text{does}(xplayer, mark(1,1))\big) \vee \\ &\text{true}(cell(1,1,x))\end{aligned} \tag{2}$$

does$(xplayer, mark(1,1))$ in (2) is further replaced with boolean true, because both role and action match the ones we are interested in right now.

$$\big(\text{true}(cell(1,1,b)) \wedge \text{T}\big) \vee \text{true}(cell(1,1,x)) \tag{3}$$

[1] In the current implementation, we take into consideration only the highest valued goal for each player. Combining different goals could be done similar to the way described in [Schiffel and Thielscher, 2007], but would make the heuristics more expensive to compute.

[2] We detect whether a game is turn-taking and also which action is the noop action, using a theorem prover [Haufe *et al.*, 2012] or using random simulation in case theorem proving does not yield an answer. The cost of this is negligible compared to, e.g., grounding the game rules.

The formula (3) can be simplified, which yields (4) as the final replacement for true$(cell(1,1,x))$:

$$\text{true}(cell(1,1,b)) \vee \text{true}(cell(1,1,x)) \tag{4}$$

Going back to the goal condition (1), the next part of the formula to be regressed is true$(cell(2,2,x))$. The matching *next* rules are:

```
next(cell(2, 2, x)) :- true(cell(2, 2, b)),
  does(xplayer, mark(2, 2)).
next(cell(2, 2, x)) :- true(cell(2, 2, x)).
```

This time, the does$(xplayer, mark(2,2))$ is replaced with boolean false, because the role matches, but the action does not. This yields formula (5), which can be simplified to (6). This equals the term we have started with, meaning that true$(cell(2,2,x))$ stays in the formula untouched.

$$\big(\text{true}(cell(2,2,b)) \wedge \text{F}\big) \vee \text{true}(cell(2,2,x)) \tag{5}$$

$$\text{true}(cell(2,2,x)) \tag{6}$$

We repeat the same steps for any other *true* keywords in the goal (1). As in the previous case, each term is replaced by the term itself and nothing in the formula is changed. The final heuristic formula for *xplayer* taking action $mark(1,1)$ is:

$$\begin{aligned}\big(&\text{true}(cell(2,2,x)) \wedge \text{true}(cell(3,3,x)) \wedge \\ &(\text{true}(cell(1,1,b)) \vee \text{true}(cell(1,1,x)))\big) \vee \\ &\big(\text{true}(cell(3,1,x)) \wedge \text{true}(cell(2,2,x)) \wedge \\ &\text{true}(cell(1,3,x))\big)\end{aligned} \tag{7}$$

As can be seen, this condition describes a situation in which *xplayer* taking action $mark(1,1)$ would lead to a winning state. Thus, a boolean evaluation of such conditions for all legal moves in a state is equivalent to doing 1-ply lookahead.

Evaluation

Using the algorithm above, the heuristic formula is constructed for any role and any potentially legal move during the start clock. During game play, the formula for each legal action is evaluated against the current game state s using fuzzy logic as described in [Schiffel and Thielscher, 2007], but with different t-norm and t-co-norm, as the original ones proved to be too slow. For non-atomic formulas the evaluation function is defined as

$$\begin{aligned} eval(f \wedge g, s) &= \top(eval(f,s), eval(g,s)) \\ eval(f \vee g, s) &= \bot(eval(f,s), eval(g,s)) \\ eval(\neg f, s) &= 1 - eval(f,s) \end{aligned}$$

where $\top$ and $\bot$ are the product t-norm and t-co-norm:

$$\begin{aligned} \top(a,b) &= a \cdot b \\ \bot(a,b) &= a + b - a \cdot b \end{aligned}$$

All remaining atoms of the heuristic formula are of the form true(X), these are evaluated as

$$eval(true(f), s) = \begin{cases} p & \text{if } f \text{ is true in } s \\ 1-p & \text{otherwise} \end{cases}$$

We used $p = 0.97$ for our experiments.

Thus, our action heuristics can be defined as $H(s,r,a) = eval(F_{r,a}, s)$, where $F_{r,a}$ is the heuristic formula constructed for role r and action a.

In essence, a higher value of the evaluation means, that more prerequisites are satisfied for a particular action to lead to a goal state.

The heuristic values for each legal action in the initial state of the aforementioned version of the game Tic Tac Toe are shown in the following table.

(1, 3) 0.0026	(2, 3) 0.0018	(3, 3) 0.0026
(1, 2) 0.0018	(2, 2) 0.0035	(3, 2) 0.0018
(1, 1) 0.0026	(2, 1) 0.0018	(3, 1) 0.0026

The heuristic function was evaluated in the initial game state (empty board). We can see, that the action with the highest value, is to take the middle cell ($mark(2,2)$), followed by 4 actions taking one of the corner cells. Indeed, this corresponds with the fact that marking the middle cell as the first action leads to the most options for winning the game.

In [Schiffel and Thielscher, 2007] and [Michulke and Schiffel, 2013], we described methods for improving the fuzzy evaluation by using additional knowledge about the game for the evaluation of the atoms. The same methods can be used for the action heuristics presented here. For the experiments presented in the next section, we restricted ourselves to using only very limited additional knowledge especially selected for not increasing the time for evaluating the heuristics significantly. In particular, we only use knowledge about persistent fluents as defined in [Haufe *et al.*, 2012].

A fluent is *persistent true* if, once it holds in a state, it will persist to hold in all future states. For example, $cell(1,1,x)$ is *persistent true* in Tic Tac Toe. A fluent is *persistent false* if, once it does not hold in a state, it will never hold in any future state. For example, $cell(1,1,b)$ is *persistent false* in Tic Tac Toe.

Information like this can be detected in some, but not all of the games we tested. In case we could (automatically) infer this knowledge, we modify the evaluation function as follows:

$$eval(true(f), s) = \begin{cases} 1 & \text{if } f \text{ is true in } s \text{ and} \\ & \quad f \text{ is persistent true} \\ p & \text{if } f \text{ is true in } s \text{ and} \\ & \quad f \text{ is not persistent true} \\ 0 & \text{if } f \text{ is false in } s \text{ and} \\ & \quad f \text{ is persistent false} \\ 1-p & \text{otherwise} \end{cases}$$

Search Controls

In this section we recapitulate three ways, how an action heuristic can be utilized in the MCTS player to control differ-

Game	No heur. vs. Playout h.	No heur. vs. Tree h.	No heur. vs. Combined h.
battle	**90.2** × 80.9 (±2.46)	87.3 × **95.1** (±2.05)	**91.6** × 84.9 (±1.95)
bidding-tictactoe	19.2 × **80.8** (±3.47)	42.8 × **57.2** (±3.01)	19.7 × **80.3** (±3.57)
blocker	**61.0** × 39.0 (±5.52)	48.0 × 52.0 (±5.65)	**67.3** × 32.7 (±5.31)
breakthrough	51.3 × 48.7 (±5.66)	47.3 × 52.7 (±5.65)	48.7 × 51.3 (±5.66)
checkers-small	**94.3** × 5.7 (±2.36)	50.2 × 49.8 (±4.39)	**96.0** × 4.0 (±1.91)
chinesecheckers2	**81.0** × 69.0 (±2.75)	75.3 × 74.6 (±2.84)	**85.7** × 64.3 (±2.56)
chinook	**76.0** × 32.0 (±5.06)	54.7 × 56.3 (±5.62)	**76.0** × 37.0 (±5.15)
connect4	**81.2** × 18.8 (±4.11)	**56.8** × 43.2 (±5.44)	**86.7** × 13.3 (±3.56)
crisscross	**75.3** × 49.8 (±3.99)	62.3 × 62.8 (±4.24)	**74.3** × 50.8 (±4.03)
ghostmaze2p	28.7 × **71.3** (±3.12)	19.5 × **80.5** (±3.26)	30.8 × **69.2** (±3.38)
nineBoardTicTacToe	26.7 × **73.3** (±5.00)	32.3 × **67.7** (±5.29)	22.7 × **77.3** (±4.74)
pentago_2008	22.7 × **77.3** (±4.48)	35.5 × **64.5** (±5.10)	21.2 × **78.8** (±4.25)
sheep_and_wolf	**74.7** × 25.3 (±4.92)	51.7 × 48.3 (±5.65)	**78.7** × 21.3 (±4.64)
skirmish	70.1 × 69.0 (±1.14)	79.4 × 77.3 (±1.51)	**78.4** × 75.0 (±1.49)

Table 1: Tournament using playout, tree and combined heuristic against pure MCTS player with fixed number of simulations.

Game	MAST vs. Playout h.	RAVE vs. Tree h.	MAST+RAVE vs. Combi. h.
battle	**95.0** × 83.7 (±1.83)	87.9 × **96.1** (±1.95)	**94.1** × 80.1 (±1.77)
bidding-tictactoe	26.2 × **73.8** (±4.24)	41.1 × **58.9** (±3.05)	26.5 × **73.5** (±4.34)
blocker	**60.0** × 40.0 (±5.54)	50.8 × 49.2 (±5.67)	**59.3** × 40.7 (±5.56)
breakthrough	45.7 × 54.3 (±5.64)	50.0 × 50.0 (±5.66)	52.3 × 47.7 (±5.65)
checkers-small	**95.0** × 5.0 (±2.14)	51.0 × 49.0 (±4.33)	**95.0** × 5.0 (±2.04)
chinesecheckers2	**96.4** × 53.3 (±1.48)	74.8 × 75.2 (±2.84)	**95.1** × 54.5 (±1.69)
chinook	**84.0** × 30.7 (±4.68)	52.0 × 59.0 (±5.61)	**86.3** × 28.0 (±4.48)
connect4	**83.3** × 16.7 (±4.04)	52.8 × 47.2 (±5.45)	**83.0** × 17.0 (±3.91)
crisscross	62.8 × 62.3 (±4.24)	62.3 × 62.8 (±4.24)	62.5 × 62.5 (±4.24)
ghostmaze2p	31.3 × **68.7** (±3.37)	20.2 × **79.8** (±3.17)	33.3 × **66.7** (±3.53)
nineBoardTicTacToe	34.3 × **65.7** (±5.37)	35.3 × **64.7** (±5.41)	31.3 × **68.7** (±5.25)
pentago_2008	21.2 × **78.8** (±4.37)	42.3 × **57.7** (±5.28)	20.3 × **79.7** (±4.19)
sheep_and_wolf	**77.0** × 23.0 (±4.76)	53.7 × 46.3 (±5.64)	**76.3** × 23.7 (±4.81)
skirmish	80.5 × 79.7 (±1.52)	79.7 × 77.6 (±1.52)	**84.8** × 75.3 (±1.56)

Table 2: Tournament against MAST and RAVE player with constant time per turn.

ent part of the search. All methods are described in [Finnsson and Björnsson, 2011]. While the first method uses the heuristic to guide the random playouts, in the second one it controls the search tree growth in the selection phase. The last approach presented is a combination these two concepts. We will use these methods later together with our own heuristics.

Playout Heuristic

In the standard MCTS with UCT, actions are selected uniformly at random during the playout phase. However, if we have any information on which actions are good, it is better to bias the action selection in favor of more promising moves [Finnsson and Björnsson, 2008]. This can be accomplished using the Gibbs (Boltzmann) distribution:

$$P(s,r,a) = \frac{e^{H(s,r,a)/\tau}}{\sum_{a'} e^{H(s,r,a')/\tau}}$$

where $P(s,r,a)$ is the probability that the action a will be chosen by role r in the current playout state s and $H(s,r,a)$ is the action heuristic function. The parameter τ is *temperature* and specifies how random actions are chosen. Whereas the high values makes it rather uniform; $\tau \to 0$ means that more valued actions are chosen more likely. Based on trial and error testing, good values for τ lie somewhere between 0.5 and 2. We used $\tau = 1$ in our tests.

One drawback of this method is, that the heuristic function must be evaluated for all legal moves in every playout state within the simulation, which is sometimes too costly.

Tree Heuristic

An action heuristic commonly used in the game tree is *Rapid Action Value Estimation* (RAVE) [Gelly and Silver, 2007]. It keeps a special value $Q_{RAVE}(s,r,a)$, which is an average outcome of simulations, where action a was taken by role r in any state on the path below s.

In our case, instead of Q_{RAVE}, we use the value $H(s,r,a)$ provided by the action heuristic described earlier. Initially, only the heuristic is used to give an estimate of the action value, but as the sampled action value $Q(s,r,a)$ becomes more reliable with more simulations executed, it should be more trusted over the heuristic $H(s,r,a)$. This is achieved by using a weighted average as in RAVE:

$$Q(s,r,a)' := \beta(s) \times H(s,r,a) + (1 - \beta(s)) \times Q(s,r,a)$$

Game	Cost of gen.	Time spent on heuristic			Relative number of simuls.		
		Playout	Tree	Combi.	Playout	Tree	Combi.
battle	13.0 s	44.6 %	0.0 %	44.3 %	67.2 %	99.3 %	66.9 %
bidding-tictactoe	0.2 s	15.8 %	0.2 %	15.5 %	125.4 %	103.2 %	125.2 %
blocker	0.2 s	45.3 %	0.1 %	42.8 %	65.0 %	95.9 %	70.2 %
breakthrough	3.5 s	53.4 %	0.1 %	53.2 %	46.3 %	104.4 %	46.2 %
checkers-small	14.8 s	22.9 %	0.1 %	22.9 %	80.6 %	103.0 %	81.4 %
chinesecheckers2	0.1 s	8.3 %	0.1 %	8.4 %	91.9 %	101.0 %	91.1 %
chinook	2.5 s	9.2 %	0.0 %	9.2 %	129.6 %	102.7 %	131.4 %
connect4	1.6 s	56.1 %	1.6 %	55.0 %	91.2 %	115.8 %	96.7 %
crisscross	2.7 s	5.5 %	0.6 %	6.0 %	100.4 %	99.8 %	103.2 %
ghostmaze	0.2 s	44.4 %	0.7 %	43.3 %	70.7 %	102.9 %	74.9 %
nineBoardTicTacToe	4.0 s	47.4 %	0.6 %	46.3 %	174.7 %	103.6 %	176.4 %
pentago_2008	1.2 s	69.4 %	0.3 %	69.3 %	52.2 %	100.2 %	52.2 %
sheep_and_wolf	3.4 s	14.1 %	0.1 %	14.1 %	84.4 %	103.0 %	84.1 %
skirmish	6.8 s	26.3 %	0.1 %	26.2 %	72.2 %	100.5 %	73.0 %

Table 3: Cost of generating the action heuristic, % of the game time spent on evaluating it and relative number of simulations compared to the non-heuristic player..

with

$$\beta(s) = \sqrt{\frac{k}{3 \times N(s) + k}}$$

The *equivalence parameter* k controls, how many simulations are needed for both estimates to have equal weights. The $N(s)$ function returns number of visits of the state s.

Before use, all heuristic values $H(s, r, a)$ are normalized and scaled into the range 0 to 100 which is the range of possible game outcomes. We use $k = 20$ in our tests, as it turned out to work best with the heuristic function we use.

Combined Heuristic

Both of the previous control schemes can be combined together and a heuristic can be used to guide the action selection both during the random playouts and in the game tree. As was shown in [Finnsson and Björnsson, 2010], this combination has a synergic effect, when they used RAVE as a tree heuristic and MAST as a playout heuristic. In our case, we use the action heuristic we developed in both control schemes with the same parameters as mentioned above.

Results

We run two sets of experiments. First, we matched players using the three aforementioned concepts of the action heuristic (playout, tree and combined heuristic) against a pure MCTS/UCT player with constant number of simulations per turn. Then, we matched playout, tree and combined heuristic against MAST, RAVE and MAST+RAVE players, respectively. We did not match MAST against our tree heuristic, because they operate in different stages of MCTS and the results are not comparable. Similarly for playout heuristics vs. RAVE. Constant time per turn was used in this experiment. We set temperature $\tau = 10$ for MAST and equivalence parameter $k = 1000$ for RAVE as recommended in [Finnsson and Björnsson, 2010].

Each set consists of 300 matches per game with each control scheme. The tests were run on Linux with multicore Intel(R) Xeon(R) 2.40 GHz processor with 4 GB memory limit and 1 CPU core assigned to each agent. Rules for all the games tested can be found in the game repository at `games.ggp.org`.

Playing Strength

Table 1 shows the average scores reached by the pure MCTS and the heuristic agent along with a 95 % confidence interval. We allowed 10000 simulations per turn and the time spent on evaluating the heuristic was measured.

The game with the strongest position of the combined scheme is Bidding Tic Tac Toe with a score 80 ($\pm$3.5) against 20; it is also good in Nine Board Tic Tac Toe, Pentago and Ghost Maze. On the other hand, it is particularly bad in Checkers and Connect4, but closer look reveals, that this is only because of the playout heuristic, while the tree heuristic has not much influence in these games. The playout heuristic follows the similar trend as the combined scheme. The tree heuristic is also significantly better in Battle and there is no game with totally hopeless result as there was with the playout heuristic.

It is also worth mentioning, how the results in the combined control scheme are connected to the playout and the tree heuristic. It seems, the influence of the playout heuristic on the overall result is much higher. Especially, when it is useless or even misleading, then the combined result is dragged down by it (Checkers, Connect4). It seems that the playout heuristic is more vulnerable, while the tree heuristic control scheme can recover when the heuristic is misleading. Thus, it is essential for the playout heuristic to be good, if it is used.

Bidding Tic Tac Toe is a game, where two players are bidding coins in order to mark a cell with an objective to build a line of three symbols as in Tic Tac Toe. However, the key property of this game is the bidding part – by doing wrong bids, the game can be easily lost, even when the markers are placed in good positions on the board. The action heuristic does not help in any way with the bidding, it only helps to

arrange the markers in a line. In spite of this, all the heuristic agents still have a significant advantage in this game. One explanation is that when a player wins a bid, it can actually use its move well which makes especially the playouts more reliable.

Table 2 shows results for matches played against MAST, RAVE and MAST+RAVE. Although MAST and MAST+RAVE outperforms the playout and the combined heuristic in most of the games, they still hold their good position in Pentago or Nine Board Tic Tac Toe. The heuristic performs surprisingly well against RAVE with no significant loss. By comparing tables 1 and 2, we see that our action heuristics perform well against MAST and RAVE in exactly the same games in which they did well against a pure MCTS/UCT player. This suggests, that our approach is complementing MAST and RAVE by improving performance in games in which MAST and RAVE do not seem to have much positive effect.

In general, it can be said, that the heuristic player performs very well in Tic Tac Toe-like games, as they contain many persistently true fluents and the heuristic is built in such a way, that it leads the player to the goal directly.

Time Spent on Evaluating the Heuristic

Table 3 shows how much time during the game time was spent on evaluating the heuristic and the time needed to generate the heuristic functions for each game. These numbers do not include the time required for grounding the game description. However, most general game players use grounded game descriptions for reasoning nowadays, such that grounding needs to be done anyway.

The time required to generate the action heuristic is relatively small for the games tested, except for Checkers with almost 15 seconds and Battle with 13 seconds. However, the time spent on evaluating the heuristic is more important. While it is almost negligible for tree heuristic, it ranges from 5 to 70 % depending on the game for playout and combined heuristic. The table also shows ratio between the time needed to run 10000 game simulations by the pure and the heuristic players. Surprisingly, the heuristic player can sometimes run significantly more simulations than its non-heuristic counterpart, because the heuristic makes the simulations effectively shorter and thus taking less time. A good example of this behavior is in Nine Board Tic Tac Toe, where about 50 % of the game time is spent on evaluating the heuristic, but still with about three quarters more simulations done. Moreover, the combined heuristic player won 77 % of the matches against pure MCTS/UCT.

An idea, how to make the evaluation faster is to investigate pruning of the heuristic formula. It seems that some parts of it are triggered only in some relatively rare game states and do not contribute much to the overall result. Also evaluation of state fluents that are changing wildly from state to state (like control fluent) has probably a little influence on the playing strength.

Testing with Other Games

We have been able to generate action heuristic for 113 games out of 127 available on the Tiltyard server[3]. Of those 14 games that failed we were not able to ground the game rules in 5 cases. Others failed mostly because the goal condition was particularly complex.

A game that showed to be most problematic is Othello, because the grounded version of the goal is extremely big. Other games that failed include different versions of Chess, Amazons and Hex. On the other hand, there are some games, where the grounded game description is still rather big, but the goal condition itself is relatively simple. In this case, we are able to generate the action heuristic successfully. Examples of such games are Breakthrough or Skirmish.

Conclusion

We have presented a general method of creating action heuristic in General Game Playing based on regression and fuzzy evaluation. We used the heuristics in three different search control schemes for MCTS and demonstrated the effectiveness by comparing it with a pure MCTS/UCT, RAVE and MAST players. The combined heuristic agent outperforms these players in well-known games like Bidding Tic Tac Toe, Pentago or Nine Board Tic Tac Toe; however it shows significant loss ratio in Checkers and Connect 4.

At least partly this behavior can be explained by the fact that Tic Tac Toe-like games typically contain many persistent fluents which we use to improve the heuristics. Thus, one idea for improving the quality of the heuristics in other games is to use more feature discovery techniques, such as the ones described in [Kuhlmann *et al.*, 2006], [Schiffel and Thielscher, 2007] or [Michulke and Schiffel, 2013]. Another idea would be to regress the goal condition by more than one step. In both cases care has to be taken to not increase the evaluation time too much.

There are still certain issues to be addressed. Notably, the playout heuristic seems to be prone to be misleading. As it has the most influence on the overall performance, this behavior should be further investigated.

Future work should also investigate pruning of the heuristic formula to only include the most relevant features in order to reduce evaluation time.

References

[Chaslot *et al.*, 2007] Guillaume M. J. B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.

[Clune, 2007] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139. AAAI Press, 2007.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *CG2006*, pages 72–83, 2006.

[3] tiltyard.ggp.org

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*. AAAI Press, 2008.

[Finnsson and Björnsson, 2010] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game playing agents. In *AAAI*, pages 954–959. AAAI Press, 2010.

[Finnsson and Björnsson, 2011] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, 25(1):9–16, 2011.

[Finnsson, 2012] Hilmar Finnsson. Generalized Monte-Carlo tree search extensions for general game playing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[Gelly and Silver, 2007] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, volume 227, pages 273–280, 2007.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Haufe *et al.*, 2012] Sebastian Haufe, Stephan Schiffel, and Michael Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, 187-188:1–30, 2012.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, page 282–293, 2006.

[Kuhlmann *et al.*, 2006] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62, 2006.

[Lanctot *et al.*, 2014] Marc Lanctot, Mark H. M. Winands, Tom Pepels, and Nathan R. Sturtevant. Monte Carlo tree search with heuristic evaluations using implicit minimax backups. In *2014 IEEE Conference on Computational Intelligence and Games (CIG 2014)*, pages 341–348, 2014.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, March 2008. most recent version should be available at http://games.stanford.edu/.

[Michulke and Schiffel, 2013] Daniel Michulke and Stephan Schiffel. Admissible distance heuristics for general games. In *Agents and Artificial Intelligence*, volume 358, pages 188–203. Springer Berlin Heidelberg, 2013.

[Pepels *et al.*, 2014] Tom Pepels, Mandy J. W. Tak, Marc Lanctot, and Mark H. M. Winands. Quality-based rewards for Monte-Carlo tree search simulations. In *21st European Conference on Artificial Intelligence (ECAI 2014)*, pages 705–710. IOS Press, 2014.

[Reiter, 2001] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, pages 61–73. Massachusetts Institute of Technology, 2001.

[Schiffel and Björnsson, 2014] Stephan Schiffel and Yngvi Björnsson. Efficiency of GDL reasoners. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

[Schiffel and Thielscher, 2007] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196. AAAI Press, 2007.

[Sharma *et al.*, 2008] Shiven Sharma, Ziad Kobti, and Scott D. Goodwin. Knowledge generation for improving simulations in UCT for general game playing. In *Australasian Conference on Artificial Intelligence*, volume 5360. Springer, 2008.

[Tak *et al.*, 2012] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Björnsson. N-grams and the last-good-reply policy applied in general game playing. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 73–83, 2012.

[Waledzik and Mandziuk, 2014] K. Waledzik and J. Mandziuk. An automatically generated evaluation function in general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(3):258–270, Sept 2014.

[Winands *et al.*, 2010] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo tree search in lines of action. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 239–250, 2010.

Space-Consistent Game Equivalence Detection in General Game Playing

Haifeng Zhang
Peking University
Beijing, China
pkuzhf@pku.edu.cn

Dangyi Liu
Peking University
Beijing, China
ldy@pku.edu.cn

Wenxin Li
Peking University
Beijing, China
lwx@pku.edu.cn

Abstract

In general game playing, agents play previously unknown games by analysing game rules which are provided in runtime. Since taking advantage of experience from past games can efficiently enhance their intelligence, it is necessary for agents to detect equivalence between games. This paper defines game equivalence formally and concentrates on a specific scale, space-consistent game equivalence (SCGE). To detect SCGE, an approach is proposed mainly reducing the complex problem to some well studied problems. An evaluation of the approach is performed at the end.

1 Introduction

According to human experience, exploiting equivalence between a new problem and a studied problem provides a bridge for knowledge transfer, which efficiently enhances the understanding of the new problem. Therefore, for the aim of artificial intelligence, it is important to enable computer to recognize equivalence. Particularly, as a typical application of AI, it is necessary for game-playing agents to grasp the ability of detecting equivalence between games.

The main work of this paper is to discuss classification of game equivalence, define concepts of it formally and propose an approach to detect it. Since detecting the general equivalence between games is difficult, a narrowed scale of game equivalence, *space-consistent game equivalence*, is defined firstly. Then, an approach is proposed for agents to automatically detect space-consistent game equivalence, which intends to achieve an acceptable efficiency by defining a grounded rule graph and transferring the complex problem to the well studied problems, i.e. graph isomorphism and SAT.

This paper discusses game equivalence in the domain of General Game Playing (GGP) [Genesereth *et al.*, 2005], which sets up a framework for agents to play previously unknown games by being provided game rules in runtime. This framework obliges agents to take over the responsibility of analysing game rules from human beings. The games in GGP are turn-based, synchronized and of complete information, which are described in the Game Description Language (GDL) [Love *et al.*, 2008].

(*role* r)	r is a player
(*init* p)	proposition p holds in the initial state
(*true* p)	proposition p holds in the current state
(*legal* r a)	player r has legal action a in the current state
(*does* r a)	player r does Action a
(*next* p)	proposition p holds in the next state
terminal	the current state is terminal
(*goal* r n)	utility of player r in current terminal state is n

Table 1: GDL Keywords

The work of this paper can be applied to knowledge transfer between equivalent or similar games. For example, [Kuhlmann and Stone, 2007] introduces a method of value function transfer for speeding up reinforcement learning, based on the technique of game equivalence detection. It can also be applied to detect symmetry of games, as [Schiffel, 2010] does.

The following section provides background on GGP and introduces definitions of game. Section 3 discusses game equivalence and its narrowness. Section 4 introduces the proposed approach to detect game equivalence, which is evaluated in Section 5. Section 6 concludes the work of this paper.

2 General Game Playing

In the domain of General Game Playing, games are modelled as finite state machines. In this paper, the definitions of game derive from [Schiffel and Thielscher, 2010].

Definition (Game)**.** *Let Σ be a countable set of ground (i.e., variable-free) symbolic expressions (terms), S a set of states, and A a set of actions. A (discrete, synchronous, deterministic) game is a structure (R, s_0, T, L, u, G), where*

- *$R \subseteq \Sigma$ finite (the roles);*
- *$s_0 \in S$ (the initial state);*
- *$T \subseteq S$ finite (the terminal states);*
- *$L \subseteq R \times A \times S$ finite (the legality relation);*
- *$u : (R \to A) \times S \to S$ finite (the update function);*
- *$G \subseteq R \times \mathbb{N} \times S$ finite (the goal relation).*

Here, $A \subseteq \Sigma$ and $S \subseteq 2^{\Sigma}$. The legality relation $(r, a, s) \subseteq L$ defines action a to be a legal action for role r in state s. The

```
1 (role xplayer) (role oplayer)
2 (init (cell 1 1 b)) (init (cell 1 2 b))...(init (cell 3 3 b))
3 (init (control xplayer))
4 (<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b)) (true (control ?w)))
5 (<= (legal xplayer noop) (true (control oplayer)))
6 (<= (legal oplayer noop) (true (control xplayer)))
7 (<= (next (cell ?m ?n x)) (does xplayer (mark ?m ?n)) (true (cell ?m ?n b)))
8 (<= (next (cell ?m ?n o)) (does oplayer (mark ?m ?n)) (true (cell ?m ?n b)))
9 (<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w)) (distinct ?w b))
10 (<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k)) (true (cell ?m ?n b)) (or (distinct
       ?m ?j) (distinct ?n ?k)))
11 (<= (next (control xplayer)) (true (control oplayer)))
12 (<= (next (control oplayer)) (true (control xplayer)))
13 (<= (row ?m ?x) (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
14 (<= (column ?n ?x) (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
15 (<= (diagonal ?x) (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
16 (<= (diagonal ?x) (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
17 (<= (line ?x) (or (row ?m ?x) (column ?m ?x) (diagonal ?x))
18 (<= open (true (cell ?m ?n b)))
19 (<= (goal xplayer 100) (line x))
20 (<= (goal xplayer 50) (not (line x)) (not (line o)) (not open))
21 (<= (goal xplayer 0) (line o))
22 (<= (goal oplayer 100) (line o))
23 (<= (goal oplayer 50) (not (line x)) (not (line o)) (not open))
24 (<= (goal oplayer 0) (line x))
25 (<= terminal (or (line x) (line o) (not open)))
```

Listing 1: Rules of Tic-tac-toe

update function u takes an action for each role and (synchronously) applies the joint actions to a current state, resulting in the updated state. The goal relation $(r, n, s) \subseteq G$ defines n to be the utility for role r in state s.

In General Game Playing, rules of games are described in the GDL, which is a Prolog-like language using prefix syntax. Some keywords of the GDL are defined in Table 1. As a demonstration of the GDL, the rules of Tic-tac-toe are provided in Listing 1.

Here, the symbol <= is the implication operator. Tokens starting with a question mark are variables. The first line declares two roles of the game. Lines 2-3 define the initial state. Lines 4-6 define legal actions for roles. In order to describe an asynchronous turn-based game, an extra action noop is provided to players during their opponents' turns. Lines 7-12 define the update function. For example, Line 7 implies that (cell 1 1 x) holds in the next state if xplayer does the action (mark 1 1) and (cell 1 1 b) holds in the current state. Lines 13-18 define several auxiliary propositions describing properties of the current state. It is convenient to use these propositions in the following rules. Lines 19-24 define the goal relation of the game, while Line 25 defines the terminal states.

Except the keywords and logical words, which are printed italic, all tokens are game-specific and can be replaced by other tokens without changing the meaning of the game. Auxiliary propositions and variables are used for convenience and compactness, which can be eliminated without changing the meaning of the game.

Provided a GDL description, a game is defined as follows.

Definition (Game for GDL)**.** *Let D be a valid GDL game description, whose signature determines the set of ground terms Σ. The game for D is the game (R, s_0, T, L, u, G), where*

- $R = \{r \in \Sigma | D \models$ *(role r)* $\}$
- $s_0 = \{p \in \Sigma | D \models$ *(init p)* $\}$
- $T = \{s \in S | D \cup s^{true} \models$ *terminal* $\}$

2	9	4
7	5	3
6	1	8

Figure 1: Mapping between Tic-tac-toe and Number Scrabble. Picking a number corresponds to marking a cell, and collecting three numbers summing up to 15 corresponds to drawing a line.

- $L = \{(r, a, s) \in R \times A \times S | D \cup s^{true} \models$ *(legal r a)* $\}$
- $u(j : R \to A, s) = \{p \in \Sigma | D \cup j^{does} \cup s^{true} \models$ *(next p)* $\}$
- $G = \{(r, n, s) \in R \times \mathbb{N} \times S | D \cup s^{true} \models$ *(goal r n)* $\}$

Here, S is defined as 2^P where $P = \{p |$ (true p) $\in \Sigma\}$, A as $\{a |$ (legal r a) $\in \Sigma\}$, s^{true} as $\{$ (true p) $| p \in s\}$ and $(j : R \to A)^{does}$ as $\{$ (does r j(r)) $| r \in R\}$.

3 Game Equivalence

Two games looking different in rules may be identical in nature. [Pell, 1993] points out that Tic-tac-toe is identical to Number Scrabble[1]. In fact, filling the numbers of Number Scrabble into the cells of Tic-tac-toe as Figure 1 reveals the mapping between them.

Essentially, two games are equivalent exactly if the state machines described them are identical. Corresponding to the definition of game, game equivalence is defined as follows.

Definition (Game Equivalence)**.** *Game $\Gamma = (R, s_0, T, L, u, G)$ and Game $\Gamma' = (R', s_0', T', L', u', G')$ (Σ and Σ' are their grounds sets, S and S' state sets, A and A' action sets, respectively) are equivalent iff there is a bijection set $\sigma = (\sigma^R : R \leftrightarrow R', \sigma^S : S \leftrightarrow S', \sigma^A : A \leftrightarrow A')$ s.t.*

- $\sigma^S(s_0) = s_0'$
- $(\forall t)\ t \in T \Leftrightarrow \sigma^S(t) \in T'$
- $(\forall r, a, s)\ (r, a, s) \in L \Leftrightarrow (\sigma^R(r), \sigma^A(a), \sigma^S(s)) \in L'$
- $(\forall j : R \to A, \forall s_{cur}, s_{next} \in S)$ $u(j, s_{cur}) = s_{next} \Leftrightarrow u'(j', \sigma^S(s_{cur})) = \sigma^S(s_{next})$, *where* $j' : R' \to A'$ *satisfies* $j'(\sigma^R(r)) = \sigma^A(j(r))$
- $(\forall r, n, s)\ (r, n, s) \in G \Leftrightarrow (\sigma^R(r), n, \sigma^S(s)) \in G'$

The bijection set σ is called a game equivalence between Γ and Γ'.

Previous works successfully detect some kinds of game equivalence. [Kuhlmann and Stone, 2007] proposes a rule graph to detect game equivalence caused by rules reordering and tokens scrambling. Based on it, [Schiffel, 2010] enhances the rule graph to handle arguments reordering. However, more kinds of equivalence exist, such as:

- auxiliary propositions elimination, e.g. replacing (<= (p_0) (p_1)) (<= (p_1) (p_2)) by (<= (p_0) (p_2));

[1] Number Scrabble is a game for two players taking turns to pick numbers from a pool of 1-9, whose goals are collecting three numbers summing up to 15 before the opponent achieving it.

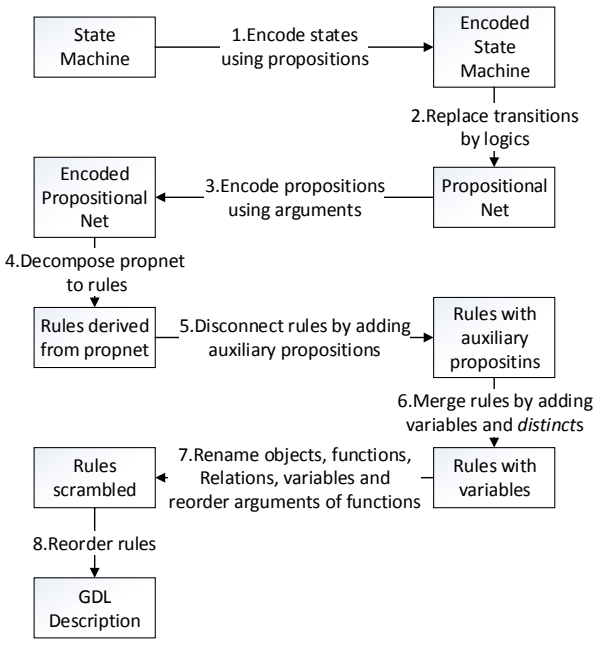


Figure 2: Transformation steps from state machine to GDL rules. Each step can yield different targets from a single source, except for Step 4 which decomposes a propnet into certain rules.

- logical conversion, e.g. replacing (<= (consequence) (not (or (condition$_1$) (condition$_2$)))) by (<= (consequence) (not (condition$_1$)) (not (condition$_2$)));
- arguments re-encoding, e.g. replacing (*true* (cell 1..3 1..3 x)) by (*true* (cell 1..9 x)).

In general, game equivalence is caused by the uncertainty of transformation from a state machine to a GDL description. A state machine can be transformed into different but equivalent propositional nets, which can be further transformed into different but equivalent GDL descriptions. Figure 2 demonstrates a reasonable sequence of transformation steps.

According to Figure 2, each kind of game equivalence is caused by one of the steps. As to the mentioned ones, rules reordering is caused by Step 8, tokens scrambling and arguments reordering are caused by Step 7, auxiliary propositions elimination is caused by Step 5, arguments re-encoding is caused by Step 3, and logical conversion is caused by Step 2.

This paper considers the steps after the encoding state machine in Figure 2. The encoded state machines derived from a particular state machine share the same propositions. These propositions form a state space, which is also shared by the encoded state machines. To describe it, space-consistent game equivalence is defined.

Definition (Space-Consistent Game Equivalence (SCGE))**.** *A game equivalence $\sigma = (\sigma^R, \sigma^S, \sigma^A)$ is a space-consistent game equivalence for two games Γ and Γ' iff there is a bijection $\sigma^P : P \leftrightarrow P'$ (where $P = \{p | p \in s, s \in S\}$, P' likewise, i.e. P and P' contains propositions forming the states) satisfying $(\forall s \in S)\ p \in s \Leftrightarrow \sigma^P(p) \in \sigma^S(s)$. Here, S is the set of states of Γ. P and P' are called state spaces. The SCGE σ can be written as $(\sigma^R, \sigma^P, \sigma^A)$, since σ^S can be determined by σ^P.*

SCGE for GDL is defined as follows, which rewrites the definition of SCGE in the context of GDL without changing the meaning.

Definition (Space-Consistent Game Equivalence for GDL)**.** *Let D and D' be valid GDL game descriptions, whose signatures determine the sets of ground terms Σ and Σ' respectively. A space-consistent game equivalence $\sigma = (\sigma^R : R \leftrightarrow R', \sigma^P : P \leftrightarrow P', \sigma^A : A \leftrightarrow A')$ for D and D' satisfies:*

- $D \models p^{init} \Leftrightarrow D' \models (\sigma^P(p))^{init}$
- $D \cup s^{true} \models$ *terminal* $\Leftrightarrow D' \cup (\sigma^S(s))^{true} \models$ *terminal*
- $D \cup s^{true} \models$ *(legal* r a*)* $\Leftrightarrow$ $D' \cup (\sigma^S(s))^{true} \models$ *(legal* $\sigma^R(r)$ $\sigma^A(a)$*)*
- $D \cup s^{true} \cup \{$*(does* r $j(r)$*)* $|r \in R\} \models$ *(next p)* $\Leftrightarrow D' \cup (\sigma^S(s))^{true} \cup \{$*(does* r $j'(r)$*)* $|r \in R'\} \models$ *(next* $\sigma^P(p)$*)*
- $D \cup s^{true} \models$ *(goal* r n*)* $\Leftrightarrow$ $D' \cup (\sigma^S(s))^{true} \models$ *(goal* $\sigma^R(r)$ n*)*

Here, p^{init} is defined as (init p), s^{true} as $\{$ (true p) $|p \in s\}$ and $\sigma^S(s)$ as $\{\sigma^P(p) | p \in s\}$. $j : R \to A$ and $j' : R' \to A'$ satisfy that $(\forall r \in R)\ j'(\sigma^R(r)) = \sigma^A(j(r))$.

SCGE covers the kinds of game equivalence caused by Step 2 and after in Figure 2. It narrows the concept of game equivalence by building a bijection between P and P' which determines the bijection between S and S', instead of building bijection between S and S' directly. For an example of space-inconsistent game equivalence which is caused by Step 1, replacing (*true* (cell 1 1 b)) in Tic-tac-toe by (not (or (*true* (cell 1 1 o)) (*true* (cell 1 1 x))) doesn't change the game, but reduces the state space of the game.

For solving the whole problem of game equivalence detection, comparing state machines directly is a method with a very high complexity. However, comparing propositional nets covers the kinds of game equivalence caused by Step 3 and after, whose complexity is logarithmic to the corresponding state machines in most cases. Since logical conversion is a quite common situation of game equivalence, Step 2 should be also taken into consideration. This is the significance of SCGE detection.

4 Space-Consistent Game Equivalence Detection

Based on the definition of SCGE for GDL, a brute force approach to detect it is enumerating all σs mapping roles, actions and propositions of states, then checking whether all pairs of mapped terms are equivalent to each other. Specifically speaking, it consists of three phases. The first phase is generating the logical implications between keyword-propositions, which is related to the propositional net. The second phase is enumerating all possible σs mapping R to R', P to P' and A to A' so that all keyword-propositions are mapped in accordance. The third phase is verifying whether

mapped keyword-propositions are equivalent to each other by comparing the logical implications generated in the first phase.

The brute force approach takes exponential time due to bijection enumeration and logical implication comparison. Therefore, Space-Consistent Game Equivalence Detection Approach (SCGEDA, or GEDA for short) is proposed. It transfers the problem to two well studied problems, i.e. graph isomorphism and boolean satisfiability, to achieve the state-of-the-art efficiency.

The GEDA consists of three phases as the brute force approach does:

- rule grounding, which is to generate all logical implications between grounded keyword-propositions;
- graph building and mapping, which is to build a dependency graph of keyword-propositions and inspect graph isomorphisms to map keyword-propositions;
- logical equivalence verifying, which is to verify whether the mapped logical implications are equivalent.

In addition, an analysis of complexity and some efficient improvements are to be introduced.

4.1 Rule Grounding

The aim of this phase is to transfer GDL rules to equivalent rules that only contain logical implications of keyword-propositions. An example of grounded rule is displayed as follows:

(<= (*goal* xplayer 100)
  (*or* (*and* (*true* (cell 1 1 x)) (*true* (cell 1 2 x)) (*true* (cell 1 3 x)))
    (*and* (*true* (cell 2 1 x)) (*true* (cell 2 2 x)) (*true* (cell 2 3 x)))
    (*and* (*true* (cell 3 1 x)) (*true* (cell 3 2 x)) (*true* (cell 3 3 x)))
    (*and* (*true* (cell 1 1 x)) (*true* (cell 2 1 x)) (*true* (cell 3 1 x)))
    (*and* (*true* (cell 1 2 x)) (*true* (cell 2 2 x)) (*true* (cell 3 2 x)))
    (*and* (*true* (cell 1 3 x)) (*true* (cell 2 3 x)) (*true* (cell 3 3 x)))
    (*and* (*true* (cell 1 1 x)) (*true* (cell 2 2 x)) (*true* (cell 3 3 x)))
    (*and* (*true* (cell 1 3 x)) (*true* (cell 2 2 x)) (*true* (cell 3 1 x))))).

To achieve it, several procedures are taken.

1. Calculate ranges of arguments, such as
   (*true* (cell {1,2,3} {1,2,3} {x,o})).
2. Replace variables by constants according to ranges of arguments, e.g. replace
   (<= (diagonal ?x)
     (*true* (cell 1 1 ?x)) (*true* (cell 2 2 ?x)) (*true* (cell 3 3 ?x)))
   by
   (<= (diagonal x)
     (*true* (cell 1 1 x)) (*true* (cell 2 2 x)) (*true* (cell 3 3 x)))
   (<= (diagonal o)
     (*true* (cell 1 1 o)) (*true* (cell 2 2 o)) (*true* (cell 3 3 o))).
   (The consistency of variables is ensured. If a *distinct*-proposition is contained in a rule, its logical value is computed and applied to the rule during this procedure.)
3. Eliminate auxiliary propositions stage by stage, e.g. replace line by row, column and diagonal before replacing row, column and diagonal by *true*.
4. Remove non-state-relative propositions and *role*-propositions from premises of rules, because their values are always true.
5. Remove the rules which use auxiliary propositions as consequences, because they are no longer of use.
6. Merge the rules which use the same propositions as consequences so that one proposition acts as the consequence in only one rule. For example, the rule with (goal xplayer 100) printed above is merged from eight partial ones with (goal xplayer 100).

After rule grounding, all rules are in the form of
(<= (consequence) Func(condition1, condition2, condition3...)),

where consequence and conditions are keyword-propositions. Keywords in consequence include *role, init, next, legal, goal* and *terminal*, while *true* and *does* are the keywords in conditions. Particularly, *role*- and *init*-propositions depend on no propositions as conditions, *next*-propositions depend on *true*- and *does*-propositions and the remaining consequences only depend on *true*-propositions. Func is a logical function connecting conditions by *and, or* and *not*, which is called the *reasoning function* of the consequence.

After this phase, rules of equivalent games are normalized except the reasoning functions.

4.2 Graph Building and Mapping

In this phase, the grounded rules excluding the reasoning functions are modelled as a so-called ground graph, which is mainly a dependency graph of keyword-propositions. Thus, the number of enumerated bijections between propositions is determined by the number of isomorphisms between the graphs, which is much smaller than completely enumeration.

Definition (Ground Graph)**.** *A ground graph $G = (V, E, l)$ for grounded rules GR is an directed labelled graph with the following properties:*

- *($\forall p$, p is a keyword-proposition appearing in GR with a keyword k as its predicate)* $p \in V$ *and* $l(p) = k$*;*
- $(\forall n \in \mathbb{N}, n \in [0, 100])$ $n \in V$ *and* $l(n) = n$*;*
- $(\forall v_s, v_t \in V, r \in GR, v_s$ *is a condition of* r *and* v_t *is the consequence of* $r)$ $(v_s, v_t) \in E$*;*
- $(\forall p^{init}, p^{true} \in V)$ $(p^{init}, p^{true}) \in E$*;*
- $(\forall p^{next}, p^{true} \in V)$ $(p^{next}, p^{true}) \in E$*;*
- $(\forall a^{does,r}, r^{role} \in V)$ $(a^{does,r}, r^{role}) \in E$*;*
- $(\forall a^{legal,r}, r^{role} \in V)$ $(a^{legal,r}, r^{role}) \in E$*;*
- $(\forall r^{goal,n}, r^{role} \in V)$ $(r^{goal,n}, r^{role}) \in E$*;*
- $(\forall a^{does,r}, a^{legal,r} \in V)$ $(a^{does,r}, a^{legal,r}) \in E$*;*
- $(\forall r^{goal,n}, n \in V)$ $(r^{goal,n}, n) \in E$*;*

Here, p^{init}*,* p^{true}*,* p^{next}*,* $a^{does,r}$*,* r^{role}*,* $a^{legal,r}$ *and* $r^{goal,n}$ *express (init p), (true p), (next p), (does r a), (role r), (legal r a), (goal r n) respectively.* $(\forall n \in \mathbb{N}, n \in [0, 100])$ *represents all possible utilities in a valid GDL description.*

Thus, a ground graph has two types of nodes, which are proposition-nodes and integer-nodes. It also has two types of edges, which are logical-dependency-edges and consistency-maintaining-edges. It only reserves logical dependencies of propositions and discards reasoning functions. Figure 3 displays a brief structure of a ground graph.

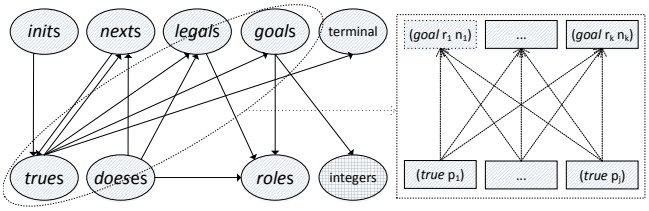


Figure 3: Brief structure of ground graph. The solid ellipses stand for sets of nodes, while the solid rectangles stand for particular nodes.

After two ground graphs are built, they are tested for isomorphism. An isomorphism between directed labelled graphs $iso : V \leftrightarrow V'$ satisfies 1) $(\forall v \in V)\ l(v) = l'(iso(v))$; 2) $(\forall v_s, v_t \in V)\ (v_s, v_t) \in E \Leftrightarrow (iso(v_s), iso(v_t)) \in E'$.

Therefore, according to the definition of ground graph, an isomorphism between two ground graphs satisfies that 1) proposition-nodes map to proposition-nodes containing the same predicates and integer-nodes map to integer-nodes with the same value; 2) for two mapped propositions, their logically dependent propositions are also mapped; Since *init*-propositions are mapped, the initial states are equivalent; the consistencies between *next*- and *true*-propositions, *does*-, *legal*-, *goal*-propositions and *role*-propositions, *does*- and *legal*-propositions respectively are maintained; the mapped *goal*-propositions have the same utility.

After this phase, if an isomorphism is built, the two games may be equivalent. The remaining uncertainty is the reasoning functions of each proposition ,which is to be considered in the next phase.

Since a game may have symmetries [Schiffel, 2010], there may be several game equivalences between two games. In general, detecting one of them is sufficient for applications such as knowledge transfer. However, in this phase all isomorphisms of ground graphs need to be found, because any of them may cause an equivalence between games. Thus, for each isomorphism, the following phase is applied.

4.3 Logical Equivalence Verifying

In this phase, the unnormalized part of grounded rules, the reasoning functions, is handled.

By rule grounding, reasoning functions of all keyword-propositions are clear. By the last phase, mappings between keyword-propositions of two games are provided, so the reasoning functions are mapped in accordance. Moreover, propositions as conditions are also mapped. In other words, variables of reasoning functions are mapped. So, the actual problem is verifying the logical equivalence of two mapped logical functions, provided the consistent variable list. For example, there are two grounded rules (<= (p1) Func(p2, p3)) and (<= (q1) Func2(q2, q3)) of two games respectively, px maps qx respectively, then the problem is checking if Func(x, y) equals Func2(x, y).

For solving this problem, the naive approach that compares the truth tables of two logical functions takes exponential time. However, the problem can be transferred to the well studied boolean satisfiability problem (SAT) to achieve an state-of-the-art efficiency. For example, testing whether logical functions f_1 and f_2 are equivalent can be transferred to testing whether $(((not\ f_1)\ and\ f_2)\ or\ (f_1\ and\ (not\ f_2)))$ is unsatisfiable. By using a SAT solver, the equivalence of two reasoning functions can be judged. So the remaining work is to verify the equivalence of all mapped reasoning functions in sequence with the SAT solver. Only if the verification is passed, the two games are equivalent and the SCGE σ can be obtained from the isomorphism of ground graphs.

4.4 Complexity

Let n be the number of reasoning functions, l the number of terms in the longest reasoning function.

For the first phase, the complexity is $O(nl)$, since the cost of grounding process is linear to the length of results.

The complexity of the second phase is at most NP-complete about n, since graph isomorphism is a special case of the NP-complete subgraph isomorphism problem [Karp, 1972].

The bottleneck is the third phase, which costs $O(m * n * NP - complete(l))$, where m denotes the number of maps generated by the second phase and $NP - complete(l)$ is the complexity of SAT problem [Karp, 1972].

The overall complexity is high. However, the approach is more efficient in practice than in theory.

4.5 Improvements

There are several improvements which can be applied to the GEDA, listed by order of importance as follows.

Heuristically grouping nodes of ground graph. The number of isomorphisms of ground graphs can be huge. For example, the number of automorphisms of Tic-tac-toe's ground graph is 9!, since all 9 cells of the board are equivalent when discarding the information expressed by reasoning functions. However, the 9 cells can be grouped into 4 corner-cells, 4 border-cells and 1 center-cell by counting the numbers they are possible to form a line, which are 3, 2 and 4 respectively. This dramatically reduces the number of automorphisms to 4!4!1!. In general, analysing the symmetry of the reasoning functions helps to group the elements of state space, so as the corresponding nodes of ground graph. Since the structure of reasoning function can be arbitrary, it is a heuristic grouping. However, it works for most occasions, because the symmetric structure is usually used by default.

Caching bad reasoning functions. Mapped reasoning functions have different possibilities to be equivalent for some reasons such as the different complexities. Caching the bad reasoning functions helps to prune early during verification.

Simplifying ground graph. Integer-nodes of ground graphs can be removed by adding a phase after graph mapping to verify the equivalence of utilities. Corresponding *true*- and *next*-proposition-nodes, *legal*- and *does*-proposition-nodes can be merged respectively. The *init*-proposition-nodes can be replaced by a single *init*-node.

Generating propositional net. Since grounded rules may need exponential space, it is more efficient to generate a propositional net and dynamically compute reasoning functions.

Game	Phase 1	Node No.	Edge No.	Phase 2	Func No.	Bijection No.	Retry No.	Phase 3
ConnectFour	$0.107s$	217	2093	$0.021s$	103	16	7	$5.369s$
ConnectFourSuicide	$0.100s$	217	2093	$0.020s$	103	16	14	$19.967s$
ConnectFourLarge	$0.302s$	465	4717	$0.144s$	223	64	5	$8.986s$
ConnectFourLarger	$1.841s$	1649	17533	$9.322s$	807	1024	7	$46.214s$

Table 2: ConnectFour series games tested with their modified versions. Node No. and Edge No. express the scale of ground graph. Func No. is the number of logical functions to be verified. Bijection No. is the number of bijections generated by Phase 2 with heuristic grouping. Retry No. is the number of bijections verified by Phase 3 to find the first equivalence.

```
1  (sum15 1 5 9) (sum15 1 6 8) (sum15 2 4 9) (sum15 2 5 8)
2  (sum15 2 6 7) (sum15 3 4 8) (sum15 3 5 7) (sum15 4 5 6)
3  (<= (win x) (sum15 ?a ?b ?c) (true (cell ?a x)) (true (cell ?b x)) (true (cell ?c x)))
4  (<= (win o) (sum15 ?a ?b ?c) (true (cell ?a o)) (true (cell ?b o)) (true (cell ?c o)))
5  (<= (goal xplayer 100) (win x))
6  (<= (goal xplayer 50) (not (win x)) (not (win o)) (not open))
7  (<= (goal xplayer 0) (win o))
8  (<= (goal oplayer 100) (win o))
9  (<= (goal oplayer 50) (not (win x)) (not (win o)) (not open))
10 (<= (goal oplayer 0) (win x))
11 (<= terminal (or (win x) (win o) (not open)))
```

Listing 2: Partial rules of Number Scrabble

Game	Brute Force	GEDA	GEDA+	Goal
Tic-tac-toe	27!	9!	4!4!1!	8
Blocker	48!	16!	4!4!4!4!	4
Breakthrough	128!	2	2	2
Peg Jumping	66!	8	8	8
Connect Four	96!	8!	2!2!2!2!	2

Table 3: Self-mapping numbers of some games. Brute Force enumerates all permutations of the elements of state space. GEDA+ stands for the GEDA with heuristic grouping. Goal stands for the number of symmetries of a game in nature.

5 Evaluation

As introduced in Section 3, Tic-tac-toe is equivalent to Number Scrabble. The different part of Number Scrabble's rules is provided in Listing 2. The auxiliary propositions defined in Lines 3-4 represent the winning conditions. The *goal*- and *terminal*-propositions are dependent on the winning conditions. The state space consists of (cell [1,9] {x,o,b}) and (control {xplayer,oplayer}), which is consistent with Tic-tac-toe. Therefore, the GEDA can work on it.

By the phase of rule grounding, 68 grounded rules are generated for each game.

In the phase of graph building and mapping, two ground graphs are built. Each ground graph has 59 nodes with the improvement of graph simplification. To generate isomorphisms of them, NAUTYv2.5 [McKay and Piperno, 2014] is applied. As mentioned above, 9! isomorphisms are found between them, which can be reduced to 4!4!1!2! by the improvement of nodes grouping. The 2! is caused by the permutation of the 2 groups with 4 nodes. In fact, enumerating these isomorphisms by an agent corresponds to repeatedly trying filling the numbers in the cells by human.

For the phase of logical equivalence verifying, MiniSAT [Een and Sörensson, 2005] is applied as a SAT solver. Since MiniSAT only accepts inputs in Conjunctive Normal Form (CNF), Tseitin transformation [Tseitin, 1968] is used to transfer the logical functions to CNF.

As a result, the equivalence of Tic-tac-toe and Number Scrabble is detected by the GEDA in 9.73 seconds on average over 10 experiments, running on a laptop with an Intel i5 CPU.

Since game equivalence happens rarely in nature, some manual examples are tested. Four ConnectFour series games are modified with some logical conversions. Each game is tested if it is equivalent with its modified version by the GEDA with improvements. Table 2 shows the results.

In practice, the number of enumerated bijections primarily determines the running time of a game equivalence detection approach. Table 3 displays a comparison of the enumerated self-mapping numbers of some games using different approaches, which simulate the bijection numbers between equivalent games. It reveals that the performance of the GEDA is close to the optimal for some games, while for some other games it is still unsatisfactory.

Taking into consideration that it usually takes negligible time to reject inequivalent games, the GEDA has potential to be applied in real applications.

6 Conclusion

This work makes progress toward detecting game equivalence automatically by an agent. First, it discusses the classification of game equivalence and defines the SCGE, which covers more complex game equivalences than the previous works. Second, it proposes the GEDA, which solves the problem of detecting the SCGE by using a grounded rule graph and transferring the problem to well studied problems to achieve state-of-the-art efficiency. It works well for some small games, while there is still room for further improvement.

This work benefits knowledge transfer between equivalent games, and can be easily extended to similar games by relaxing some conditions. Based on this work, solutions which standardize state spaces of equivalent games can be proposed for space-inconsistent game equivalence detection in the future.

References

[Een and Sörensson, 2005] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.

[Genesereth *et al.*, 2005] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[Karp, 1972] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[Kuhlmann and Stone, 2007] Gregory Kuhlmann and Peter Stone. Graph-based domain mapping for transfer learning in general games. In *Machine Learning: ECML 2007*, pages 188–200. Springer, 2007.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification, 2008.

[McKay and Piperno, 2014] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[Pell, 1993] Barney Pell. *Strategy generation and evaluation for meta-game playing*. PhD thesis, University of Cambridge, 1993.

[Schiffel and Thielscher, 2010] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In Joaquim Filipe, Ana Fred, and Bernadette Sharp, editors, *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 44–55. Springer Berlin Heidelberg, 2010.

[Schiffel, 2010] Stephan Schiffel. Symmetry detection in general game playing. In *AAAI*, 2010.

[Tseitin, 1968] Grigorii Samuilovich Tseitin. On the complexity of proof in prepositional calculus. *Zapiski Nauchnykh Seminarov POMI*, 8:234–259, 1968.

The *GRL* System: Learning Board Game Rules With Piece-Move Interactions

Peter Gregory
Digital Futures Institute
School of Computing
Teesside University
Middlesbrough, UK
p.gregory@tees.ac.uk

Yngvi Björnsson and **Stephan Schiffel**
School of Computer Science
Reykjavik University
Reykjavik, Iceland
{yngvi,stephans}@ru.is

Abstract

Many real-world systems can be represented as formal state transition systems. The modelling process, in other words the process of constructing these systems, is a time-consuming and error-prone activity. In order to counter these difficulties, efforts have been made in various communities to learn the models from input data. One learning approach is to learn models from example transition sequences. Learning state transition systems from example transition sequences is helpful in many situations. For example, where no formal description of a transition system already exists, or when wishing to translate between different formalisms.

In this work, we study the problem of learning formal models of the rules of board games, using as input only example sequences of the moves made in playing those games. Our work is distinguished from previous work in this area in that we learn the interactions between the pieces in the games. We supplement a previous game rule acquisition system by allowing pieces to be added and removed from the board during play, and using a planning domain model acquisition system to encode the relationships between the pieces that interact during a move.

1 Introduction

Over the last decade, or ever since the advent of the *General Game-Playing (GGP)* competition [Genesereth *et al.*, 2005], research interest in general approaches to intelligent game playing has become increasingly mainstay. GGP systems autonomously learn how to skilfully play a wide variety of (simultaneous or alternating) turn-based games, given only a description of the game rules. Similarly, *General Video-Game (GVG)* systems learn strategies for playing various video games in real-time and non-turn-based settings.

In the above mentioned systems the domain model (i.e., rules) for the game at hand is sent to the game-playing agent at the beginning of each match, allowing legitimate play off the bat. For example, games in GGP are described in a language named *Game Description Language (GDL)* [Love *et al.*, 2006], which has axioms for describing the initial game state, the generation of legal moves and how they alter the game state, and how to detect and score terminal positions. Respectively, video games are described in the *Video Game Description Language (VGDL)* [Schaul, 2013]. The agent then gradually learns improved strategies for playing the game at a competitive level, typically by playing against itself or other agents. However, ideally one would like to build fully autonomous game-playing systems, that is, systems capable of learning not only the necessary game-playing strategies but also the underlying domain model. Such systems would learn skilful play simply by observing others play.

Automated model acquisition is an active research area spanning many domains, including constraint programming and computer security (e.g. [O'Sullivan, 2010; Bessiere *et al.*, 2014; Aarts *et al.*, 2013]). There has been some recent work in GGP in that direction using a simplified subset of board games, henceforth referred to as *Simplifed Game Rule Learner (SGRL)* [Björnsson, 2012]. In the related field of autonomous planning, the *LOCM* family of domain model acquisition systems [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011; Cresswell *et al.*, 2013] learn planning domain models from collections of plans. In comparison to other systems of the same type, these systems require only a minimal amount of information in order to form hypotheses: they only require plan traces, where other systems require state information.

In this work we extend current work on learning formal models of the rules of (simplified) board games, using as input only example sequences of the moves made in playing those games. More specifically, we extend the previous *SGRL* game rule acquisition system by allowing pieces to be added and removed from the board during play, and by using the *LOCM* planning domain model acquisition system for encoding and learning the relationships between the pieces that interact during a move, allowing modelling of moves that have side effects (such as castling in chess). Our work is thus distinguished from previous work in this area in that we learn the interactions between the pieces in the games.

The paper is structured as follows: the next section provides necessary background material on *LOCM* and *SGRL*, followed by a description of the combined approach. This is followed by empirical evaluation and overview of related work, before concluding and discussing future work.

2 Background

In this section we provide background information about the LOCM and SGRL model acquisition systems that we base the present work on.

2.1 LOCM

The *LOCM* family of domain model acquisition systems [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011] are inductive reasoning systems that learn planning domain models from only action traces. This is large restriction, as other similar systems require extra information (such as predicate definitions, initial and goal states, etc.). *LOCM* is able to recover domain information from such a limited amount of input due to assumptions about the structure of the output domain.

A full discussion of the *LOCM* algorithm is omitted and the interested reader is referred to the background literature [Cresswell *et al.*, 2009; Cresswell and Gregory, 2011; Gregory and Cresswell, 2015] for more information. However, we discuss those aspects of the system as relevant to this work. We use the well-known Blocksworld domain as an example to demonstrate the form of input to, and output gained from, *LOCM*. Although a simple domain, it is useful as it demonstrates a range of features from both *LOCM* and *LOCM2* that are relevant to this work.

The input to the *LOCM* system is a collection of plan traces. Suppose, in the Blocksworld domain, we had the problem of reversing a two block tower, where block A is initially placed on block B. The following plan trace is a valid plan for this problem in the Blocksworld domain:

```
(unstack A B)
(put-down A)
(pick-up B)
(stack B A)
```

Each action comprises a set of indexed object transitions. For example, the unstack action comprises a transition for block A (which we denote as *unstack.1*) and another for block B (which we denote as *unstack.2*). A key assumption in the *LOCM* algorithm is that the behaviour of each type of object can be encoded in one or more DFAs, where each transition appears at most once. This assumption means that for two object plan traces with the same prefix, the next transition for that object must exit from the same state. Consider plan trace 1 and 2 below:

```
1: (unstack A B)
1: (put-down A)

2: (unstack X Y)
2: (stack X Z)
```

In the first plan trace, block A is unstacked, before being put down on to the table. In the second plan trace, X is unstacked from block Y before being stacked on to another block, Z. The assumption that each transition only exists once within the DFA description of an object type means that the state that is achieved following an *unstack.1* transition is the same state that precedes both a *put-down.1* and a *stack.1* transition.

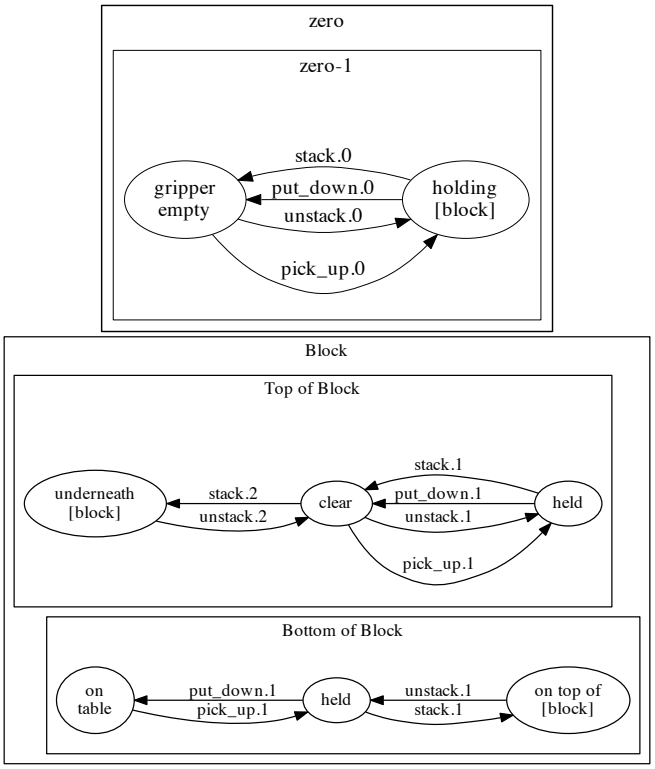


Figure 1: State machines learnt by *LOCM* in the Blocksworld planning domain. State labels are manually annotated to aid comprehension.

The output formalism of *LOCM* represents each object type as one or more parameterised DFAs. Figure 1 shows the output DFAs of *LOCM* for the Blocksworld domain. In this figure, we have manually annotated the state names in order to highlight the meanings of each state. The edges in the *LOCM* state machines represent object transitions, where each transition is labelled with an action name and a parameter position.

LOCM works in two stages: firstly to encode the structure of the DFAs, secondly to detect the state parameters.

Blocks are the only type of object in Blocksworld, and are represented by the two state machines at the bottom of Figure 1. Informally, these machines can be seen to represent what is happening above and below the block, respectively. In each planning state, a block is represented by two DFA states (for example, a block placed on the table with nothing above it would be represented by the 'clear' and 'on table' DFA states). Each of the block DFAs transition simultaneously when an action is performed, so when the previously discussed block is picked up from the table (transition pick_up.1) both the top and bottom machines move to the 'held' state.

The machine at the top of Figure 1 is a special machine known as the zero machine (this refers to an imaginary zeroth parameter in every action) and this machine can be seen as encoding the structural regularities of the input action sequences. In the case of Blocksworld, the zero machine encodes the behaviour of the gripper that picks up and puts

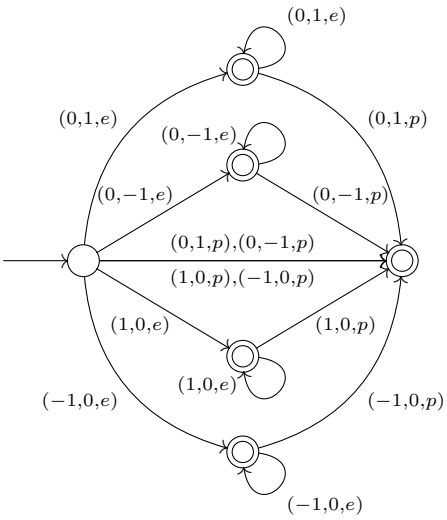


Figure 2: A DFA, D_{rook}, describing the movements of a rook in chess

down each block. Relationships between objects are represented by state parameters. As an example, in the 'Bottom of Block' machine, the state labelled 'on top of' has a block state parameter which represents the block directly underneath the current block.

2.2 The *SGRL* System

The *SGRL* approach [Björnsson, 2012] models the movements of each piece type in the game individually using a deterministic finite automata (DFA). One can think of the DFA representing a language where each word describes a completed move (or *piece movement pattern*) on the board and where each letter in the word —represented by a triplet $(\Delta x, \Delta y, on)$— describes an atomic movement. The triplet is read in the context of a game board position and a current square, with the first two coordinates telling relative board displacement from the current square (file and rank, respectively) and the third coordinate telling the content of the resulting relative square (the letter *e* indicates an empty square, *w* an o*w*n piece, and *p* an o*p*ponent's piece). For example, $(0\ 1\ e)$ indicates a piece moving one square up the board to an empty square, and the word $(0\ 1\ e)\ (0\ 1\ e)\ (0\ 1\ p)$ a movement where a piece steps three squares up (over two empty squares) and captures an opponent's piece. Figure 2 shows an example DFA describing the movements of a rook.

There are pros and cons with using the DFA formalism for representing legitimate piece movements. One nice aspect of the approach is that well-known methods can be used for the domain acquisition task, which is to infer from observed piece movement patterns a consistent DFA for each piece type (we are only concerned with piece movements here; for details about how terminal conditions and other model aspects are acquired we refer to the original paper). Another important aspect, especially for a game-playing program, is that state-space manipulation is fast. For example, when generating legal moves for a piece the DFA is traversed in a depth-first manner. On each transition the label of an edge is used to find which square to reference and its expected content. If there are no matching edges the search backtracks. A transition into a final DFA state s generates a move in the form of a piece movement pattern consisting of the edge labels that were traversed from the start state to reach s. A special provision is taken to detect and avoid cyclic square reference in piece-movement patterns.

Unfortunately, simplifying compromises were necessary in *SGRL* to allow the convenient domain-learning mechanism and fast state-space manipulation. One such is that moves are not allowed to have side effects, that is, a piece movement is not allowed to affect other piece locations or types (with the only exception that a moving piece captures the piece it lands on). These restrictions for example disallow castling, en-passant, and promotion moves in chess. We now look at how these restrictions can be relaxed.

3 The *GRL* System

In this section, we introduce a boardgame rule learning system that combines the strengths of both the *SGRL* and *LOCM* systems. One strength of the *SGRL* rule learning system is that it uses a relative coordinate system in order to generalise the input gameplay traces into concise DFAs. A strength of the *LOCM* system is that it can generalise relationships between objects that undergo simultaneous transitions.

One feature of the *SGRL* input language is that it represents the distinction between pieces belonging to the player and the opponent, but not the piece identities. This provides a trade-off between what it is possible to model in the DFAs and the efficiency of learning (far more games would have to be observed to learn if the same rules had to be learnt for every piece that can be moved over or taken, and the size of the automata learnt would be massive). In some cases, however, the identities of the interacting pieces are significant in the game.

We present the *GRL* system that enhances the *SGRL* system in such a way as to allow it to discover a restricted form of side-effects of actions. These side-effects are the rules that govern piece-taking, piece-production and composite moves. *SGRL* allows for limited piece-taking, but not the types such as in checkers where the piece taken is not on the destination square of the piece moving. Piece-production (for example, promotion in chess or adding men at the start of nine mens morris) is not at all possible in *SGRL*. Composite moves, such as castling in chess or arrow firing in Amazons also cannot be represented in the *SGRL* DFA formalism.

3.1 The *GRL* Algorithm

The *GRL* algorithm can be specified in three steps:

1. Perform an extended *SGRL* analysis on the game traces (we call this extended analysis *SGRL*+). *SGRL* is extended by adding additional vocabulary to the input language that encode the addition and removal of pieces from the board. Minor modifications have to be made to the consistency checks of *SGRL* in order to allow this additional vocabulary.
2. Transform the game traces and the *SGRL*+ output into *LOCM* input plan traces, one for each game piece.

Within this step, a single piece move includes everything that happens from the time a player picks up a piece until it is the next player's turn. For example, a compound move will have all of its steps represented in the plan trace.

3. Use the *LOCM* system to generate a planning domain model for each piece type. These domain models encode how a player's move can progress for each piece type. Crucially, unlike the *SGRL* automata, the learn domains refer to multiple piece types, and the relationships between objects through the transitions.

The output of this procedure will be a set of planning domains (one for each piece) which can generate the legal moves of a board game. By specifying a planning problem with the current board as the initial state, then enumerating the state space of the problem provides all possible moves for that piece. We now describe these three stages in more detail.

3.2 Extending *SGRL* Analysis for Piece Addition and Deletion

The first stage of *GRL* is to perform an extended version of the *SGRL* analysis. The input alphabet has been extended in order to include vocabulary to encode piece addition and removal. To this end, we add the following two letters as possible commands to the DFA alphabet:

1. The letter 'a' which means that the piece has just been added at the specified cell (the relative coordinates of this move will always be $(0, 0)$). The piece that is picked up now becomes the piece that is 'controlled' by the subsequent micro-moves.
2. The letter 'd' which means that the square has a piece underneath it which is removed from the game. This is to model taking pieces in games like peg solitaire or checkers.

The order in which these new words in the vocabulary are used is significant. An 'a' implies that the piece underneath the new piece remains in place after the move, unless it is on the final move of the sequence of micro-moves (this is consistent with the more limited piece-taking in *SGRL*, where pieces are taken only at the end of a move, if the controlled piece is another piece's square). For example, the following sequences both describe white pawn promotion in chess:

```
(0 1 e) (0 0 a) (0 0 d)
(0 1 e) (0 0 a)
```

Adding this additional vocabulary complicates the *SGRL* analysis in two ways: firstly in the definition of the game state, and secondly in the consistency checking of candidate DFAs. There is no issue when dealing with the removal of pieces and state definitions. There is, however, a small issue when dealing with piece addition. A move specified in *SGRL* input language is in the following form:

```
(12 (0 1 e) (0 0 a) (0 1 e))
```

This move specifies that the piece on square 12 moves vertically up a square, then a new piece piece appears (and becomes the controlled piece), which subsequently moves another square upwards. The issue is that for each distinct move, in *SGRL*, the controlled piece is defined by the square specified at the start of the move (in this case square 12). If a piece appears during a move, then *SGRL*+ needs to have some way of knowing which piece has appeared, in order to add this sequence to the prefix tree of the piece type. To mitigate this problem, we require all added pieces to be placed in the state before they are added, and as such they can now be readily identified. This approach allows us to learn the DFAs for all piece types, including piece addition.

The consistency algorithm of *SGRL* verifies whether a hypothesised DFA is consistent with the input data. One part of this algorithm that is not discussed in prior work is the *generateMoves* function, that generates the valid sentences (and hence the valid piece moves) of the state machine. There are two termination criteria for this function: firstly the dimensions of the board (a piece cannot move outside of the confines of the board), secondly on revisiting squares (this is important in case cycles are generated in the hypothesis generation phase). This second case is relaxed in *GRL* slightly since squares may be visited more than once due to pieces appearing and / or disappearing. However, we still wish to prevent looping behaviour, and so instead of terminating when the same square is visited, we terminate when the same square is visited in the same *state* in the DFA.

With these two changes to *SGRL* analysis we have provided a means to representing pieces that appear and that remove other pieces from the board. However, the state machines do not tell us how the individual piece movement, additions and removals combine to create a complete move in the game. For this task, we employ use of the *LOCM* system, in order to learn the piece-move interactions.

3.3 Converting to *LOCM* Input

We use the *LOCM* system to discover relationships between the pieces and the squares, and to do this we need to convert the micro-moves generated by the *SGRL*+ DFAs to sequences of actions. To convert these, we introduce the following action templates (where X is the name of one of the pieces):

```
appearX ( square )
removeX ( square )
moveX ( squareFrom, squareTo )
moveOverX ( squareFrom, squareVia, squareTo )
moveAndRemoveX ( squareFrom, squareTo )
moveAndRemoveX ( squareFrom, squareVia, squareTo )
```

These action templates mirror the input language of the extended *SGRL*+ system detailed above. Each plan that is provided as input to *LOCM* is a sequence of micro-moves that define an entire player's turn. When *LOCM* learns a model for each piece type, the interactions between the pieces and the squares they visit are encoded in the resultant domain model.

The input actions, therefore, encode the entire (possibly compound and with piece-taking side-effects) move. As an example from the Amazons[1] game, the following plan trace

[1] Amazons is a two-player game played on a 10×10 board, in which each player has four pieces. On each turn, a player moves a piece as the queen moves in chess, before firing an arrow from the destination square, again in the move pattern of the queen in chess. The final square of the arrow is then removed from the game. The loser is the first player unable to make a move.

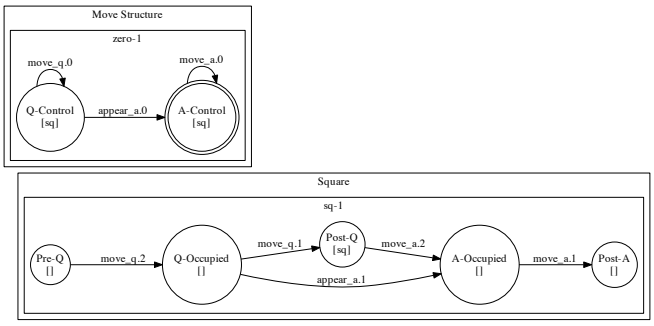


Figure 3: The *LOCM* State Machines learnt for the Queen piece in Amazons. The zero machine at the top of the figure shows the higher-level structure of the Queen's move. The Queen moves for a certain number of moves, then an arrow appears, and the arrow moves for a certain number of moves.

may be generated from the example micro-move sequence, meaning a queen is moved from A1 to B1, and then an arrow appears, and is fired to square C1:

```
Micro-move Sequence:
(0 (1 0 e) (0 0 a) (1 0 e))

Equivalent Plan Trace:
moveQueen ( A1, B1 )
appearArrow ( B1 )
moveArrow ( B1, C1 )
```

In all plan traces for the Queen piece in the Amazons game, the destination square of the Queen will always be the square in which the Arrow appears, and from which the Arrow piece begins its movement. Once the *LOCM* analysis is complete, this relationship is detected, and the induced planning domain will only allow arrows to be fired from the square that the queen moves to.

Figure 3 shows the actual *LOCM* state machines learnt for the Queen piece from a collection of game traces. The top state machine represents the zero machine, which describes the general plan structure. The machine underneath represents one of the squares on the board. The parameters of the zero machine represent the current square underneath the controlled piece. As can be seen from the generated PDDL action for $appear\_a$ (shown in Figure 4), this parameter ensures that the arrow appears at the same location that the queen ended its move on. It also ensures that the moves are sequenced in the correct way: the Queen piece moves, the Arrow piece appears and finally the Arrow piece moves to its destination. The machine at the bottom of Figure 3 encodes the transitions that a square on the board goes through during a Queen move. Each square undergoes a similar transition path, with two slightly different variants, and to understand these transitions it is important to see the two options of what happens on a square once a queen arrives there. The first option is that the Queen transits the square, moving on to the next square. The second option is that the Queen ends its movement; in this case, the Arrow appears, before transiting the square. These two cases are taken into account in the two different paths through the state machine.

As another example, consider the game of Checkers. The state machines generated by *LOCM* for the Pawn piece type

```
(:action appear_a
 :parameters (?Sq1 - sq)
 :precondition
   (and (Q_Control ?Sq1)
        (Q_Occupied ?Sq1))
   :effect
   (and (A_Control ?Sq1)
        (not (Q_Control ?Sq1))
        (A_Occupied ?Sq1)
        (not (Q_Occupied ?Sq1))))
```

Figure 4: The PDDL Action for `appear_a`, representing the arrow appearing in the Amazons game.

are shown in Figure 5. There are two zero machines in this example, which in this case means that *LOCM2* analysis was required to model the transition sequence, and there are separable behaviours in the move structure. These separable behaviours are the movement of the Pawn itself and of the King, if and when the Pawn is promoted. The top machine models the alternation between taking a piece and moving on to the next square for both Pawns and Kings, the bottom machine ensures that firstly a King cannot move until it has appeared, and secondly that a Pawn cannot continue to move once it has been promoted.

3.4 Move Generation

We now have enough information to generate possible moves based on a current state. The *GRL* automata can be used in order to generate moves, restricted by the *LOCM* state machines, where the *LOCM* machines define the order in which the pieces can move within a single turn.

The algorithm for generating moves is presented here as Algorithm 1. The algorithm takes as input an *SGRL*+ state, a *LOCM* state and a square on the board. A depth-first search is then performed over the search space, in order to generate all valid traces. A state in this context is a triple of a board square, an *SGRL*+ piece state and a *LOCM* state. Lines 2 to 3 define the termination criteria (a state has already been visited), lines 4 to 5 define when a valid move has been defined (when each of the *SGRL*+ and *LOCM* DFAs are in a terminal state), and lines 6 to 7 define the recursion (in these lines, we refer to a state as being consistent. Consistency in this sense means that the state is generated by a valid synchronous transition in both the *SGRL*+ and *LOCM* DFAs.

Algorithm 1 Function to generate legal moves for a player from a combination of *SGRL*+ and *LOCM* DFAs.

1: **function** GenerateMoves(sq,$\mathrm{S_L}$,$\mathrm{S_P}$)
2: **if** Visited $\langle sq, \mathrm{S_L}, \mathrm{S_P} \rangle$ **then**
3: **return**
4: **if** terminal($\mathrm{S_L}$) and terminal($\mathrm{S_P}$) **then**
5: add the current micro-move path to moves.
6: **for all** consistent next states $\langle sq', S'_L, S'_P \rangle$ **do**
7: GenerateMoves(sq', $\mathrm{S'_L}$,$\mathrm{S'_P}$)

This completes the definition of the *GRL* system: an extended *SGRL* learns DFAs for each piece type, the *LOCM* system learns how these piece types combine to create entire moves, and then the move generation algorithm produces the

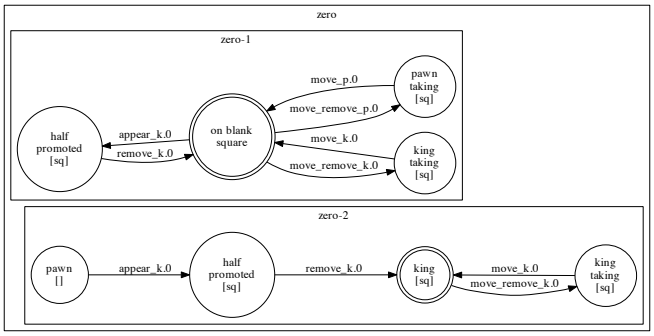


Figure 5: The *LOCM* State Machines learnt for the Pawn piece in the game of Checkers. The state machine incorporates the possibility of the Pawn being promoted into a King, and then subsequently can take other pieces.

valid moves in any given state. Next, we provide an evaluation of *GRL*, and detail the cost of each element of the system.

4 Empirical Evaluation

In this section, we provide an evaluation of the *GRL* system. In order to perform this evaluation, we learn the rules of Amazons, Peg Solitaire and Checkers. We also provide evaluation for the three games Breakthrough, Breakthrough Checkers, and Breakthrough Chess as used in the *SGRL* evaluation [Björnsson, 2012] in order to demonstrate that performance is not adversely affected due to the changes in *GRL*.We report on the time taken in the extended *SGRL+* phase, and the *LOCM* phase, to show the balance of time taken in each phase. We generate two forms of game traces for each game: one has a single move per turn (for these we generate 1000 game traces) and the other enumerates all possible moves (for these we generate 50 game traces). This method of evaluation is consistent with that chosen in the prior evaluation of *GRL*. All experiments are run on Mac OSX 10.10, running on an Intel i7 4650U CPU, with 8GB system RAM.

Table 1 shows the results of *GRL* on our benchmark problems. We show the time taken for each element of *GRL* to learn its model. We report both the time taken to learn the model for the first and second players (reported under 'Player1' and 'Player2'). In the case of Peg Solitaire, there is only one player, which explains the missing data. The missing data in the *LOCM* element for Amazons A piece is because an arrow piece in Amazons never starts a move itself and hence has no plan trace data. We first observe that the Breakthrough, Breakthrough Checkers and Breakthrough Chess results are not significantly different to the results for *SGRL*. Therefore the modifications made to the *SGRL* algorithm are not adversely affecting performance. We do not report *LOCM* time here, since these games do not require the *LOCM* analysis, as *SGRL+* is sufficient to represent them. It is notable that the time taken to learn the *LOCM* models is significantly larger than the time to learn the *SGRL+* individual piece models. The main cause of this difference is that the input plan traces are for every *turn* of the game, rather than the entire game trace. Because of this, *LOCM* has 25,000 input traces for the Queen piece in Amazons and 11,000 for the Pawn piece in Checkers, for example. The time required to parse and analyse this number of plans is necessarily time consuming.

Game	Piece	Element	Player1	Player2
Amazons	Q	*SGRL+*	8.1	35.2
		LOCM	20.9	30.0
	A	*SGRL+*	9.3	10.8
		LOCM	–	–
Checkers	P	*SGRL+*	<0.01	<0.01
		LOCM	260.4	275.3
	K	*SGRL+*	<0.01	<0.01
		LOCM	52.9	47.0
Peg Solitaire	P	*SGRL+*	<0.01	–
		LOCM	8.8	–
BT	P	*SGRL+*	<0.01	<0.01
CheckerBT	P	*SGRL+*	0.16	0.17
ChessBT	P	*SGRL+*	<0.01	<0.01
	K	*SGRL+*	<0.01	<0.01
	N	*SGRL+*	<0.01	<0.01
	B	*SGRL+*	3.3	3.4
	R	*SGRL+*	3.9	3.8
	Q	*SGRL+*	12.3	12.5

Table 1: Learning time in seconds to learn models for each of the pieces in the problem set with all moves known. (Using all moves only affects the time for the *SGRL+* element, as *LOCM* does not accept input when all moves are known).

Table 2 shows the results of learning when only a single move per turn is known. Learning times are increased, as it takes longer to find consistent DFAs in the *SGRL+* phase. Note that on several occasions, *SGRL+* returns incorrect solutions. In these cases, there is simply insufficient input data. *GRL* is still effective in the majority of cases, however, and does not degrade the performance of *SGRL* on the previous game data.

5 Related Work

Within the planning literature there are several domain model acquisition systems, each with varying levels of detail in their input observations. The *Opmaker2* system [McCluskey *et al.*, 2009; Richardson, 2008] learns models in the target language of OCL [McCluskey and Porteous, 1997] and requires a partial domain model, along with example plans as input. The ARMS system [Wu *et al.*, 2007], can learn STRIPS domain models with partial or no observation of intermediate states in the plans, but does at least require predicates to be declared. The LAMP system [Zhuo *et al.*, 2010] can target PDDL representations with quantifiers and logical implications.

As for domain model acquisition in boardgames, an ILP approach exists for inducing chess variant rules from a set of positive and negative examples using background knowledge and theory revision [Muggleton *et al.*, 2009]. Furthermore, [Kaiser, 2012] presents a system that learns games such as Connect4 and Breakthrough from video demonstrations us-

Game	Piece	Element	Player1	Player2
Amazons	Q	*SGRL+*	51.2*	54.9
		LOCM	20.9	30.0
	A	*SGRL+*	86.2*	34.6*
		LOCM	–	–
Checkers	P	*SGRL+*	<0.01	<0.01
		LOCM	260.4	275.3
	K	*SGRL+*	<0.01	<0.01
		LOCM	52.9	47.0
Peg Solitaire	P	*SGRL+*	<0.01	<0.01
		LOCM	8.8	–
BT	P	*SGRL+*	<0.01	<0.01
CheckerBT	P	*SGRL+*	1.2*	3.1*
ChessBT	P	*SGRL+*	<0.01	<0.01
	K	*SGRL+*	<0.01	<0.01
	N	*SGRL+*	<0.01	<0.01
	B	*SGRL+*	43.2	41.0
	R	*SGRL+*	52.4	50.0
	Q	*SGRL+*	145.3	140.4

Table 2: Learning time in seconds to learn models for each of the pieces in the problem set with only a single move known per state. Results marked with a (*) returned an incorrect DFA.

ing minimal background knowledge. Rosie [Kirk and Laird, 2013] is an agent implemented in Soar that learns game rules (and other concepts) for simple board games via restricted interactions with a human. As for general video-game playing, a neuro-evolution algorithm showed good promise playing a large set of Atari 2600 games using little background knowledge [Hausknecht *et al.*, 2014].

6 Conclusions and Future Work

In this work, we presented a system capable of inducing the rules of more complex games than the current state-of-the-art system. This can help in both constructing the rules of games, or replacing the current move generation routine of an existing general game playing system (for fitting games). *GRL* improves *SGRL* by allowing piece addition, removal and structured compound moves. This was achieved by combining two techniques for domain-model acquisition, one rooted in game playing and the other in autonomous planning.

However, there remain interesting classes of board game rule that cannot be learnt by *GRL*. One interesting rule class is that of a state-bound movement restriction. Games that exhibit this behaviour allow only a subset of moves to occur in certain contexts: examples of this are check in chess (where only the subset of moves that exit check are allowed) and the compulsion to take pieces in certain varieties of checkers (thus restricting the possible moves to the subset that take pieces when forced). An approach to learning these restrictions could be developed given knowledge of all possible moves at each game state in the game.

Learning the termination criteria of a game is also an important step, if the complete set of rules of a board game are to be learnt. This requires learning properties of individual states, rather than state transition systems. However, many games have termination criteria of the type that only one player (or no players) can make a move. For this class of game, and others based on which moves are possible, it should be possible to extend the *GRL* system to learn how to detect terminal states.

References

[Aarts *et al.*, 2013] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468. Ieee, March 2013.

[Bessiere *et al.*, 2014] Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El Houssine Bouyakhf. Boosting Constraint Acquisition via Generalization Queries. In *ECAI*, 2014.

[Björnsson, 2012] Yngvi Björnsson. Learning Rules of Simplified Boardgames by Observing. In *ECAI*, pages 175–180, 2012.

[Cresswell and Gregory, 2011] Stephen Cresswell and Peter Gregory. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, pages 42 – 49, 2011.

[Cresswell *et al.*, 2009] Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquisition of object-centred domain models from planning examples. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.

[Cresswell *et al.*, 2013] SN Cresswell, TL McCluskey, and MM West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(2):195 – 213, 2013.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Gregory and Cresswell, 2015] Peter Gregory and Stephen Cresswell. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling*, pages 97–105, 2015.

[Hausknecht *et al.*, 2014] Matthew J. Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(4):355–366, 2014.

[Kaiser, 2012] Lukasz Kaiser. Learning games from videos guided by descriptive complexity. In Jörg Hoffmann 0001 and Bart Selman, editors, *AAAI*, pages 963–970. AAAI Press, 2012.

[Kirk and Laird, 2013] James R. Kirk and John Laird. Interactive task learning for simple games. In *Advances in Cognitive Systems*, pages 11–28. AAAI Press, 2013.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, and Michael Genesereth. General Game Playing: Game description language specification. Technical Report April 4 2006, Stanford University, 2006.

[McCluskey and Porteous, 1997] Thomas Lee McCluskey and Julie Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95(1):1–65, 1997.

[McCluskey *et al.*, 2009] T. L. McCluskey, S. N. Cresswell, N. E. Richardson, and M. M. West. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, pages 93–100, January 2009.

[Muggleton *et al.*, 2009] Stephen Muggleton, Aline Paes, Vítor Santos Costa, and Gerson Zaverucha. Chess revision: Acquiring the rules of chess variants through FOL theory revision from examples. In Luc De Raedt, editor, *ILP*, volume 5989 of *LNCS*, pages 123–130. Springer, 2009.

[O'Sullivan, 2010] B O'Sullivan. Automated Modelling and Solving in Constraint Programming. *AAAI*, pages 1493–1497, 2010.

[Richardson, 2008] N. E. Richardson. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. PhD thesis, School of Computing and Engineering, University of Huddersfield, UK, 2008.

[Schaul, 2013] Tom Schaul. A video game description language for model-based or interactive learning. In *CIG*, pages 1–8. IEEE, 2013.

[Wu *et al.*, 2007] Kangheng Wu, Qiang Yang, and Yunfei Jiang. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *Knowl. Eng. Rev.*, 22(2):135–152, 2007.

[Zhuo *et al.*, 2010] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, 174:1540–1569, December 2010.

MCTS Playouts Parallelization with a MPPA Architecture

Aline Hufschmitt and **Jean Méhat** and **Jean-Noël Vittaut**
LIASD - University of Paris 8, France
{alinehuf,jm,jnv}@ai.univ-paris8.fr

Abstract

We present a study of the use of a *Multi-Purpose Processor Array* (MPPA) architecture for the parallelization of MCTS algorithms applied to the field of *General Game Playing*. We evaluate the constraints imposed by this architecture and show that the only feasible MCTS parallelization on MPPA is a *leaf parallelization*. We show that the MPPA provides good scalability when increasing the size of the communications which is useful when using synchronous communications to send large sets of game initial positions to be processed. We consider two approaches for the calculation of playouts: the distributed computing of a playout on each cluster and the calculation of several playouts per cluster; the second approach gives better results. Finally, we describe experiments concerning the thread management and present a surprising result: it is more efficient to create new threads than to synchronize permanent threads.

1 Introduction

General Game Playing (GGP) is a branch of Artificial Intelligence with the aim of achieving versatile programs capable of playing any game without human intervention. These programs must be able to analyze the rules of an unknown game, to understand the goals, to discover the sequences of moves leading to victory and to play with expertise.

To find out what moves to play, different techniques of tree search have been developed to explore positions of the game and different branches of possibilities that depend on the selected moves. The root of the tree represents the initial state of the game. The goal is to reach one of the leaves, corresponding to the end of the game, the score of which is the highest possible for the player.

Currently, the best GGP players use variants of *Monte Carlo Tree Search* (MCTS) combining the construction of a game tree with game simulations (playouts) [Browne *et al.*, 2012]. UCT, the most used MCTS algorithm consists in four phases: selection of a path in the game tree according to some policy; expansion of the tree with the creation of a new node; game simulation playing random moves until a terminal position is reached; back-propagation of the playout results to update node's evaluation. The number of playouts done by MCTS is a key factor for the quality of the evaluation of a game tree node [Kocsis and Szepesvári, 2006].

Former GGP players have used Prolog unification to interpret the language describing the rules. These calculations were slow [Schiffel and Björnsson, 2014]. Recently, the use of *Propositional Networks* (propnets) has allowed an important improvement in playout computation speed. A Propositional Network [Schkufza *et al.*, 2008] is a graph representing game rules as a logic circuit with AND, OR and NOT gates and transitions that represent the passage from one game state to another. At each game step, the current position is set at the input of the circuit. The signal is propagated and, at the output, a flag indicates whether this position is terminal. If it is, the player's scores are accessed through flags; otherwise, the list of legal moves is available at the circuit output; the moves chosen by the players are set at the input of the circuit and the signal is propagated to get the next state. A transition consist of setting the next state as the current position.

To increase the number of playouts made in a given time, various approaches of parallelization of MCTS have been implemented on machines with multi-core and/or multi-threads CPU used alone or networked to provide more computing power. Some of these techniques are applied to *General Game Playing* [Méhat and Cazenave, 2011a; Finnsson, 2012] but none, to our knowledge, realizes this parallelism on a player using a *propnet*. The important acceleration of playout computation provided by the use of propnets brings new conditions for parallelization of UCT because communication and synchronization times become significant.

In this article, we study the parallelization of MCTS using *propnets* to interpret game rules on a *Multi-Purpose Processor Array* (MPPA), an architecture dedicated to many-core processing, marketed since 2013 by the Kalray Company (Essonne, France). This recent architecture has barely been tested on practical applications, thus we explore its possibilities and evaluate its limitations.

Section 2 describes the different parallelization techniques proposed in the literature. Section 3 presents the MPPA architecture. Section 4 presents the feasible parallelization on MPPA. Section 5 evaluates the scalability using communications of different size. Section 6 establishes the best approach to compute the playouts. Section 7 compares different ways of synchronizing threads inside clusters.

2 Parallelization approaches of MCTS

In this section, we present the different approaches for parallelizing the MCTS.

Tree parallelization [Gelly *et al.*, 2006] involves several processes in the construction of a single game tree in a shared memory. Threads perform the selection, expansion, simulation and back-propagation phases independently. A global mutex used to protect the tree from concurrent updates creates a bottleneck, so improvements consist in using local mutexes or a lock free algorithm [Enzenberger and Müller, 2010]. *Virtual losses* avoid the selection of the same node by all the threads [Chaslot *et al.*, 2008].

On a distributed memory system, *root parallelization* [Cazenave and Jouandeau, 2007] develops different trees from the same game position, usually on different machines. The different evaluations are collected and combined to choose the best move [Méhat and Cazenave, 2011b; Soejima *et al.*, 2010]. An improvement consists in synchronizing the evaluations of the top nodes at regular intervals to direct the search using the shared information [Gelly *et al.*, 2008]. A drawback is that the same nodes have to be generated on different machines.

With *leaf parallelization* [Cazenave and Jouandeau, 2007], a single tree is constructed by a master process. It executes the selection and the expansion phases and sends the positions to be explored to slave processes calculating the playouts. The back-propagation phase is performed by the master process using the score returned by each slave. Asynchronous communications provide good scalability [Cazenave and Jouandeau, 2008]. This technique is less effective than *root or tree parallelizations* because it suffers from limitations caused by the master process, which is alone to perform the selection phase, and communications needed for each playout [Soejima *et al.*, 2010]. To minimize communication costs, [Finnsson, 2012] has proposed to realize multiple playouts per position but some unworthy positions are then over-exploited. [Chaslot *et al.*, 2008] have proposed to stop a group of playouts that does not look promising to search again from another position but this approach does not provide good scalability.

In the *UCT-Treesplit* algorithm [Graf *et al.*, 2011; Schaefers *et al.*, 2011] the nodes of a single tree are spread over distributed memories. The machine on which a node is stored is selected using a hash key of this node. During the selection phase, the traversal of edges between nodes stored into different machines is implemented by a request. If a machine receives a request for an existing node, it takes over the selection; if not, it creates this new node and launches a playout. After the playout, an update message has to be sent to all the machines storing parent nodes for the back-propagation. A drawback is the importance of communications during selection and back-propagation.

3 MPPA architecture

The MPPA-256 chip is a multi-core processor composed of 256 processing cores (PC) organized in a grid of 16 clusters connected through a high-speed *Network-on-Chip* (NoC). It is the first member of the MPPA MANYCORE family, the others reaching up to 1024 processors in a single silicon chip.

Each cluster contains 2MB of memory shared by the 16 PCs and a system core. The system core, running NodeOS, supervises the scheduling and execution of tasks on PCs and data transfers while the 16 PCs are dedicated to application code. Each PC can execute its own code, stored in the shared memory.

The limitation of a cluster memory to 2MB is a major constraint in the development of applications for the MPPA as it has to contain the system core OS, the 16 PC code and data. In our case, we see in section 4 that it reduces the choices of possible parallelization approaches.

Four I/O interfaces allow communications between the host machine and the clusters for two of them and between clusters and the Ethernet network for the other two, but with the current version of the middleware we only have access to one of the I/O interfaces: the software tools designed for the MPPA and grouped under the name of MPPA ACCESSCORE are currently under development. This I/O interface has a quad core SMP processor with a 4GB DDR3 memory and a PCI Gen3 interface for communications with the host. Different sofware connectors are available to implement synchronous or asynchronous communications using a simple buffer or queues.

[Jouandeau, 2013] states that "the computing capabilities of Intel i7 3820 processor with 8 cores and a MPPA processor with 256 cores are close". The estimate is based on the performance observed for a single core on solving the spin glass problem and multiplied by the number of cores, which implies parallel algorithms with zero communication time. Even if the MPPA-256 card seems to be limited in its performance, it is only the first member of the MPPA MANYCORE family. A *Coolidge* processor with 1024 cores is expected in 2015. In addition, several MPPA-256 cards can be used together in the same host machine to increase the computing capacity. Therefore, the MPPA architecture presents possibilities that deserve investigations.

4 Setup of the parallelization on the MPPA

Our experiments were carried out on a server equipped with an Intel Core I7 at 3.6GHz running Linux OS and a PCIe Application Board AB01 equipped with a chip MPPA-256 [Kalray, 2013].

Our program is based on Jean-Noël Vittaut's *LeJoueur*, written in C++. This player uses Prolog to realize a fast instantiation of the game rules [Vittaut and Méhat, 2014] and creates a propnet. The propnet logic circuit is factorised to reduce its size and each layer of logic gates of the circuit is then translated into a set of rules that can be evaluated very quickly with binary operators. This is the set of rules that is sent to the MPPA to explore the game.

The limited memory inside the clusters implies a coding as concise as possible so that the *propnet* can fit into it. Unfortunately, for the most complex games, like Hex, the size of the *propnet* exceeds the available space.

The size of the remaining memory inside the clusters does not allow the construction of a game tree of significant size.

Therefore *tree parallelization* inside each individual cluster, *root parallelization* and *UCT-Treesplit* cannot be performed on a MPPA. The only remaining choice is *leaf parallelization*.

For all the experiments presented in this article we used synchronous communications. Thus the benchmark results can later be compared with the ones using asynchronous communications. The host machine send sets of game positions to the MPPA I/O interface which distributes them to the different clusters. The first PC (PC0) of each cluster is responsible for communicating with the I/O and for starting threads on other PCs via the system core. A thread can be executed by each one of the 15 remaining PCs. The PC0 sends received positions to the threads and waits for all the results before sending them back at once to the I/O.

The I/O interface waits for the results from all the clusters before sending them back at once to the host. Each transmission of a position set, calculation of the playouts and recovery of the results (the scores) is referred to as a *run*.

5 Scaling for variable size of communications

At first, we investigate the capacity of the communication networks to transmit game positions, which can present a significantly different size depending on the game at hand, between the host computer and the PCs without an hindering variation of the communication time.

In GGP the size of position descriptions varies significantly from one game description to another in a ratio from one to ten: we need about 200 to 2000 bytes to represent a position for commonly used games. To make one playout from a different position on each PC, the size of a communication has to be multiplied by the number of PCs.

To evaluate the scalability of the MPPA we varied the size of positions sent between the host and the I/O node and between the I/O node and the clusters from 200 to 2000 bytes. To prevent the variable time required to calculate the playouts from disturbing measurements, no playout was actually calculated and an empty result was returned immediately. Each execution of the program performs 1000 runs. We measured performance as the time necessary to execute these 1000 runs for 1 to 16 clusters with 1 to 16 threads per cluster.

Figure 1 presents the experimental results. With one thread on one cluster, the running time is almost constant while the size of communications is multiplied by ten. With 16 threads on one cluster, there is additional time corresponding to the thread initialization and the data distribution among threads. The curve corresponding to one thread on each of the 16 clusters shows the additional time required for the I/O node to set the connector to the destination cluster. Scaling is then constrained by two aspects: first, when the I/O node receives a set of positions it needs time to distribute them among the clusters and, second, the thread start time inside a cluster is not negligible.

Scaling is only slightly hindered when considering these two aspects separately but it is not the case when we combine them. The curve for 16 clusters and 16 threads per cluster shows a significant degradation of the scaling. However, we see that scaling remains acceptable since the execution time is multiplied by about 5 when the message size is multiplied by 10.

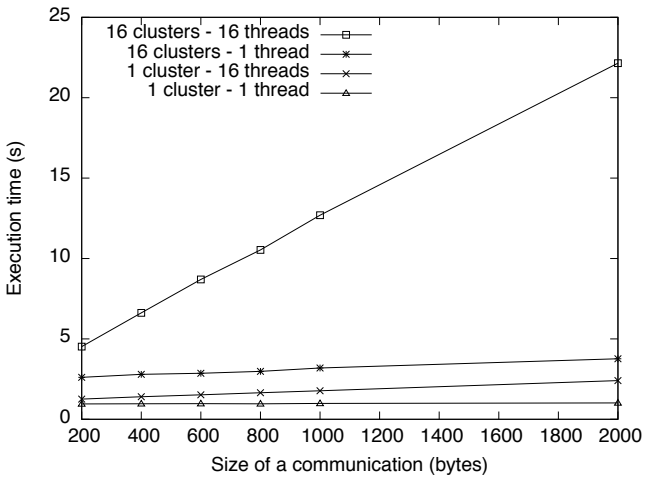


Figure 1: Evolution of the execution time depending on the size of communications (1000 runs).

6 Playout calculation approaches

In the restrictive conditions of a *leaf parallelization*, the choice remains of the playout calculation approach. All the PCs of one cluster can be used to speed up each playout by distributing the evaluation of the propnet layers or each PC can perform its own private playout evaluating all the layers by itself.

We conducted these experiments on three games: *Tictactoe* which has short playouts (between 5 and 9 moves) and a small *propnet* which is quick to evaluate, *EightPuzzle* which has a small *propnet* but playouts up to 60 moves and *Breakthrough* which has a *propnet* taking longer to evaluate. Each execution of the program performs 10000 runs. We measured the performance based on the total number of playouts carried out per second.

6.1 Parallelization of propnet evaluations

We first consider the distribution of the propnet layers evaluation between the PCs; the rules of each layer can be evaluated in any order but each layer depends on its predecessor and a synchronization is needed.

Figure 2 presents the performance obtained on the game of *Tictactoe* for 1 to 16 clusters with 1 to 16 threads per cluster. The different curves confirm that increasing the number of active clusters does improve the number of playouts per second and has no impact on the variation of performance. The latter depends on the number of threads used in the clusters. Each curve shows a large degradation as the evaluation of propnet layers is distributed between the threads. It demonstrates the significant synchronization cost generated by the division of playout computation.

We conclude that the distributed computing of a playout cannot provide any benefit considering the overhead introduced by the synchronization barrier between layers.

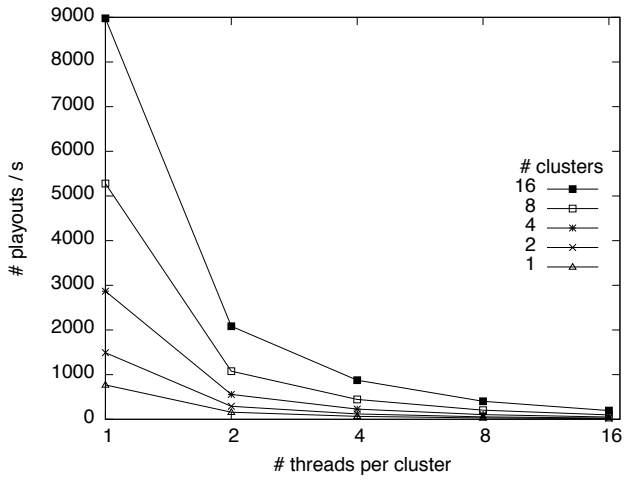


Figure 2: Evolution of the mean number of playouts per second for 10000 runs of the game of *Tictactoe* when a playout calculation is distributed over threads.

6.2 Parallelization of playouts

To avoid the cost generated by the inter-layer synchronization, it is possible to have every PC perform its own private playout, layer after layer. Synchronization is only needed when the initial positions are set and when the results are gathered.

Figure 3 presents the results for the three games with 1 to 16 clusters and 1 to 16 threads per cluster. It shows that the performance scales well.

Results for the game of Tictactoe show a progression from $\approx$450 playouts/s for one cluster with one thread to $\approx$77700 playouts/s for 16 clusters with 16 threads, i.e. a multiplication by 170 of the number of playouts per second for a multiplication by 256 of the computing power.

For the game *EightPuzzle*, the number of playouts is multiplied, at best, by 133 (for 14 threads per cluster) using 224 PCs. The performance does not scale well after 14 threads. We explain this result by the constant length of playouts: playing randomly, the solution has little chance of being found and playouts are stopped after 60 moves by the stepper. Therefore, all the scores are sent to the I/O node at the same time by all clusters causing a congestion of the communication network. A similar slowdown hindering scalability can be observed in Tictactoe by forcing the players to completely fill the grid for each game: playout length is then always set to 9 moves.

Scaling is better for *Breakthrough*. Computation time is longer, therefore communication and synchronization times are lower in comparison. The number of playouts is multiplied by 155 for a multiplication by 256 of the computing power.

7 Thread management

In the previous experiment threads were restarted for each set of playouts. Another strategy is to keep the threads alive waiting for playout requests. To test different synchronization

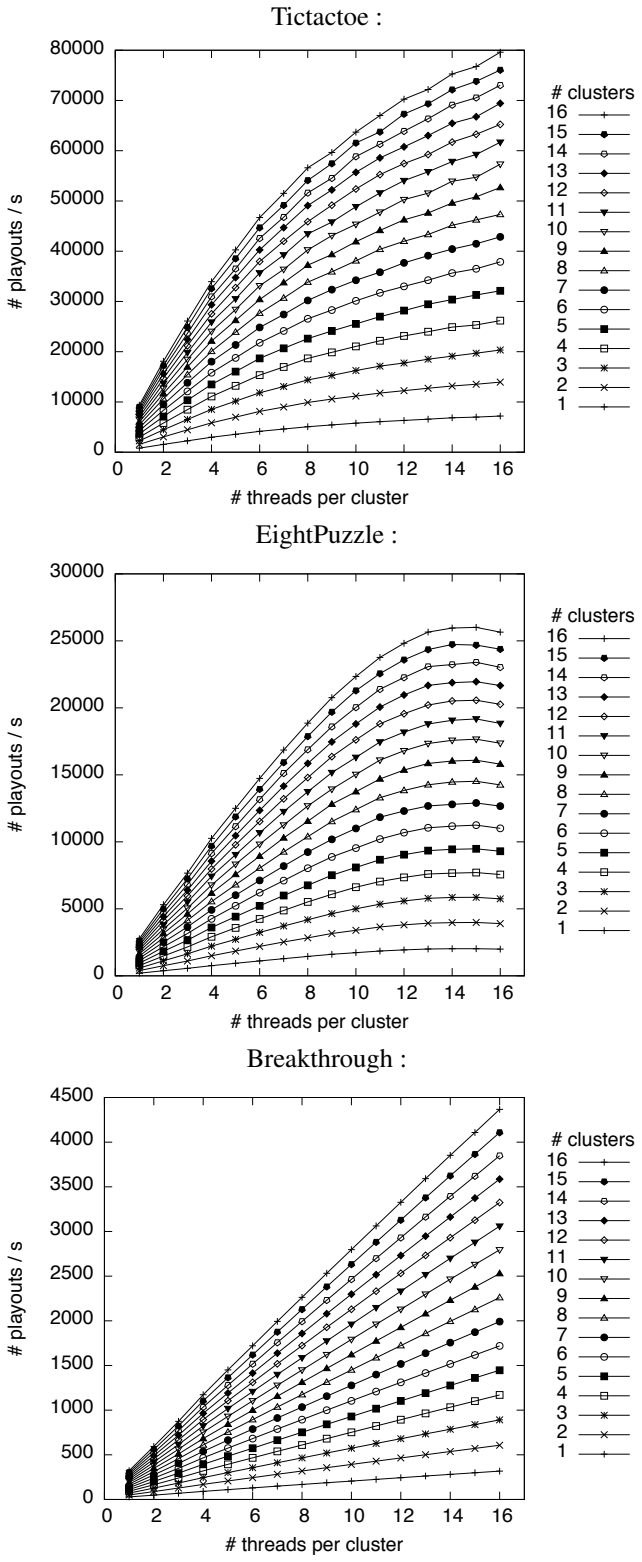


Figure 3: Evolution of the number of playouts per second (average over 10000 runs) with the calculation of one playout per thread.

approaches, we changed the thread management in the setting of the experiment of section 6.2: on each PC of each cluster, a thread is started waiting for a position.

All the positions sent were initial positions of the game and the id of each PC was used as a random seed for the random playouts it executed. The tests were performed on different games, varying the number of clusters and number of threads per cluster; 10000 runs was executed every time.

In the first part of the experiment we used the Posix functions `pthread_cond_wait` and `pthread_cond_signal` to communicate computation requests to the threads. In the second part, we replaced the Posix functions by a busy wait. In the third part we used portal connectors which are part of the MPPA ACCESCORE tools and use the NoC to create communication paths between the PC0 thread and the other PC threads of the cluster. This third approach avoids the use of mutexes.

Results are displayed in figure 4 for 16 clusters and 1 to 16 threads per cluster and compared to the results obtained with restarted threads. For the games *EightPuzzle* and *Breakthrough*, we compare only the restart of the threads with the use of the Posix functions `pthread_cond_wait` and `pthread_cond_signal` because they were sufficient for conclusive results.

With the game of *Tictactoe*, the use of the portal is less effective regardless of the number of threads used, and the different approaches keeping threads alive do not scale well above 14 threads per cluster. We note with surprise that restarting threads for each request of calculation provides better scalability and gives the best results. The results obtained on the games *EightPuzzle* and *Breakthrough* do not show such a marked difference since the curves are nearly identical. It confirms that keeping threads alive provides no benefit.

This can be explained by the use of a memory shared by all the PCs of a cluster and the fact that all the synchronization primitives end on a spinlock or equivalent. When a thread has finished its work, it waits for a signal by polling in the shared memory. Therefore, if the other threads have not finished the calculation of their playout, they are slowed down since they also need to access the shared memory. The main process running on PC0 and responsible for the communications with the I/O node is also slowed down. The more threads finish their task, the slower execution of other threads is. On the contrary, when a thread uses `pthread_exit` the PC running this thread is placed in an idle state and does not disturb the work of other PCs. Halting and relaunching the threads is then more efficient.

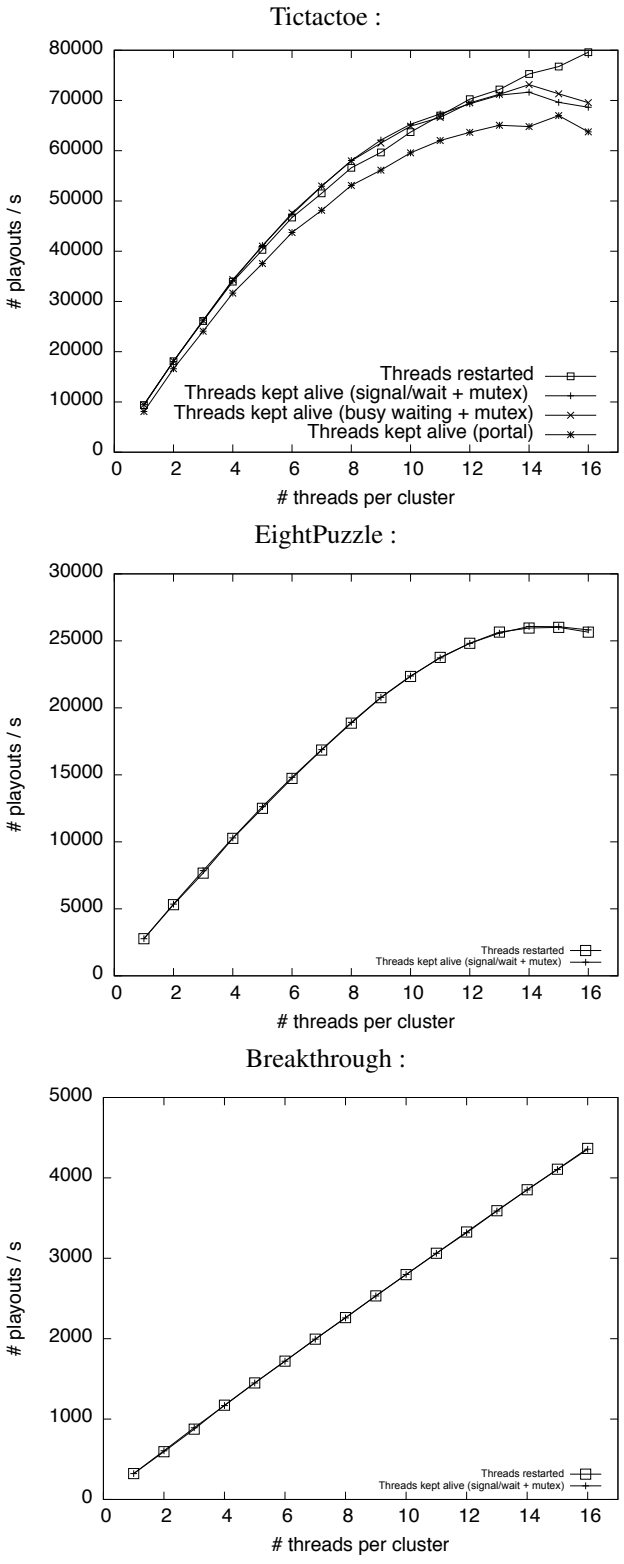


Figure 4: Evolution of the performance i.e. number of playouts per second (10000 runs, 16 clusters) with different thread management modes.

8 Conclusion

In this paper we have studied the capabilities offered by a *Multi Purpose Processor Array* (MPPA) architecture for the parallelization of MCTS algorithms in the field of *General Game Playing*. The limitation of the memory inside the cluster to a size of 2MB is a major constraint. Among the various parallelization techniques described in the literature the only applicable one, with this limited memory space, is *leaf parallelization*.

We have demonstrated that the MPPA provides good scal-

ability when increasing the size of communications, giving good results when using synchronous communications and sending large sets of game initial positions to be processed in a single run.

We have considered two approaches for the calculation of playouts. The distributed computing of a playout on the different PCs of a cluster causes an important synchronization overhead. The calculation of a complete playout per PC gives better results.

We were able to establish that on MPPA it is more efficient to restart threads for each calculation request. All synchronization primitives on the MPPA end on a spinlock. Therefore, idle threads kept alive slow down the working ones because of memory access competition.

It would be possible to reduce the weight of communications by making several playouts from each sent position, but carrying out several playouts from the same position may be less beneficial for the UCT exploration than starting from different positions [Chaslot et al., 2008].

One may think that the use of asynchronous communications would improve these results but unfortunately the first experiments we have conducted have yielded disastrous results. This comes from middleware problems that our experiments have brought to light. These problems should be fixed by Kalray in the next release. The use of asynchronous communications will therefore be the subject of future experiments.

We will also compare our results with what could be achieved with a GPU using a framework like Cuda on the same problem. The SIMT architecture requires the use of synchronous operations but this can be an advantage to distribute the calculation of each layer of the *propnet* without the need for a specific synchronization barrier. Moreover, the large quantity of memory shared by computing units allows the implementation of different parallelization techniques.

Future works also include the test of new approaches for MCTS parallelization. For example, the creation of mini-trees in cluster memory can save communication costs.

The SHOT alternative to UCT [Cazenave, 2015] offers interesting perspectives as it scales well in addition to using less memory and it can be efficiently parallelized.

The work presented here is an updated version of [Hufschmitt *et al.*, 2015].

References

[Browne *et al.*, 2012] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[Cazenave and Jouandeau, 2007] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Proceedings of the Computer Games Workshop*, pages 93–101, 06 2007.

[Cazenave and Jouandeau, 2008] Tristan Cazenave and Nicolas Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2008.

[Cazenave, 2015] Tristan Cazenave. Sequential Halving Applied to Trees. In *IEEE Trans. Comput. Intellig. and AI in Games*, volume 7, pages 102–105, 2015.

[Chaslot *et al.*, 2008] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, pages 60–71, 2008.

[Enzenberger and Müller, 2010] Markus Enzenberger and Martin Müller. A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pages 14–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[Finnsson, 2012] Hilmar Finnsson. *Simulation-Based General Game Playing*. Doctor of philosophy, School of Computer Science, Reykjavík University, 2012.

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rmi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo go. Technical Report 6062, Inria, 2006.

[Gelly *et al.*, 2008] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Y. Kalemkarian. The Parallelization of Monte-Carlo Planning - Parallelization of MC-Planning. In Joaquim Filipe, Juan Andrade-Cetto, and Jean-Louis Ferrier, editors, *ICINCO-ICSO*, pages 244–249. INSTICC Press, 2008.

[Graf *et al.*, 2011] Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers. Parallel Monte-Carlo Tree Search for HPC Systems. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 365–376, Berlin, Heidelberg, 2011. Springer-Verlag.

[Hufschmitt *et al.*, 2015] Aline Hufschmitt, Jean Méhat, and Jean-Noël Vittaut. Using MPPA architecture for UCT parallelization. In *Proceedings of the 8th International Conference on Game and Entertainment Technologies*, 2015.

[Jouandeau, 2013] Nicolas Jouandeau. Intel versus MPPA. Technical report, LIASD Universit Paris8, 11 2013.

[Kalray, 2013] Kalray. *MPPA ACCESSCORE 1.0.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA, 07 2013. 142 pages.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[Méhat and Cazenave, 2011a] Jean Méhat and Tristan Cazenave. A Parallel General Game Player. *KI*, 25(1):43–47, 2011.

[Méhat and Cazenave, 2011b] Jean Méhat and Tristan Cazenave. Tree parallelization of Ary on a cluster. In *GIGA 2011, IJCAI 2011*, Barcelona, 07 2011.

[Schaefers *et al.*, 2011] Lars Schaefers, Marco Platzner, and Ulf Lorenz. UCT-Treesplit - Parallel MCTS on Distributed Memory. In *MCTS Workshop*, Freiburg, Germany, 06 2011.

[Schiffel and Björnsson, 2014] Stephan Schiffel and Yngvi Björnsson. Efficiency of GDL Reasoners. In *IEEE Trans. Comput. Intellig. and AI in Games*, volume 6, pages 343–354, 2014.

[Schkufza *et al.*, 2008] Eric Schkufza, Nathaniel Love, and Michael R. Genesereth. Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. In *AI 2008: Advances in Artificial Intelligence, 21st Australasian Joint Conference on Artificial Intelligence, Auckland, New Zealand, December 1-5, 2008. Proceedings*, pages 56–66, 2008.

[Soejima *et al.*, 2010] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating Root Parallelization in Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287, 2010.

[Vittaut and Méhat, 2014] Jean-Noël Vittaut and Jean Méhat. Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 1121–1122, 2014.

Investigating MCTS Modifications in General Video Game Playing

Frederik Frydenberg[1], **Kasper R. Andersen**[1], **Sebastian Risi**[1], **Julian Togelius**[2]
[1]IT University of Copenhagen, Copenhagen, Denmark
[2]New York University, New York, USA
chbf@itu.dk, kasr@itu.dk, sebr@itu.dk, julian@togelius.com

Abstract

While Monte Carlo tree search (MCTS) methods have shown promise in a variety of different board games, more complex video games still present significant challenges. Recently, several modifications to the core MCTS algorithm have been proposed with the hope to increase its effectiveness on arcade-style video games. This paper investigates of how well these modifications perform in general video game playing using the *general video game AI* (GVG-AI) framework and introduces a new MCTS modification called *UCT reverse penalty* that penalizes the MCTS controller for exploring recently visited children. The results of our experiments show that a combination of two MCTS modifications can improve the performance of the vanilla MCTS controller, but the effectiveness of the modifications highly depends on the particular game being played.

1 Introduction

Game-based AI competitions have become popular in benchmarking AI algorithms [22]. However, typical AI competitions focus only on one type of game and not on the ability to play a variety of different games well (i.e. the controllers only work on one particular game / game type / problem). In this context, an important question is if it is possible to create controllers that can play a variety of different types of games with little or no retraining for each game.

The *general video game AI* (GVG-AI) competition explores this challenge. To enter the competition, a controller has to be implemented in the GVG-AI framework (available at: http://www.gvgai.net/). The framework contains two sets with ten different games in each set. The games are replicas of popular arcade games that have different winning conditions, scoring mechanisms, sprites and player actions. While playing a game the framework gives the controller a time limit of 40 milliseconds to return the next action. If this limit is exceeded, the controller will be disqualified. The competition and framework is designed by a group of researchers at University of Essex, New York University and Google DeepMind [18; 17].

The Monte Carlo tree search (MCTS) algorithm, performs well in many types of games [4; 9; 10; 8]. MCTS was first applied successfully to the Asian board game *Go*, for which it rapidly redefined the state of the art for the game. Whereas previous controllers had been comparable to human beginners, MCTS-based controllers were soon comparable to intermediate human players [10]. MCTS is particularly strong in games with relatively high branching factors and games in which it is hard to develop a reliable state value estimation function. Therefore, MCTS-based agents are generally the winners in the annual *general game playing competition*, which is focused on board games and similar discrete, turn-based perfect-information games [7; 1].

Beyond board games, MCTS has also been applied to arcade games and similar video games. In particular, the algorithm has performed relatively well in Ms. Pac-Man [13] and Super Mario Bros [8], though not better than the state of the art. In the general video game playing competition, the best agents are generally based on MCTS or some variation thereof [16; 17]. However, this is not to say that these agents perform very well – in fact, they perform poorly on most games. One could note that arcade-like video games present a rather different set of challenges to most board games, one of the key differences being that random play often does not lead to any termination condition.

The canonical form of MCTS was invented in 2006, and since then many modifications have been devised that perform more or less well on certain types of problems. Certain modifications, such as *rapid action value estimation* (RAVE) perform very well on games such as Go [6] but show limited generalization to other game types. The work on Super Mario Bros mentioned above introduced several modifications to the MCTS algorithm that markedly improved performance on that particular game [8]. However, an important open question is if those modifications would help in other arcade-like video games as well?

The goal of this paper is to investigate how well certain previously proposed modifications to the MCTS algorithm perform in general video game playing. The "vanilla MCTS" of the GVG-AI competition framework is our basis to test different modifications to the algorithm. In addition to comparing existing MCTS modifications, this paper presents a new modification called *reversal penalty*, which penalizes the MCTS controller for exploring recently visited positions. Given that the games provided with the GVG-AI framework differ along a number of design dimensions, we expect this evaluation to give a better picture of the capabilities of our new MCTS variants than any one game could do.

The paper is structured as follows: Section 2 describes related work in GVG, MCTS and use of MCTS. Section 3 describes the GVG-AI framework and competition and how we

used it. Section 4 explains the MCTS algorithm, followed by the tested MCTS modifications in Section 5. Section 6 details the experimental work and finally Section 7 discuss the results and describes future work.

2 Related work

2.1 General Video Game Playing

The AAAI general game playing competition by Stanford Logic Group of Stanford University [7] is one of the oldest and most well-known general game playing frameworks. The controllers submitted for this competition receive descriptions of games at runtime, and use the information to play these games effectively. The controllers do not know the type or rules of the game beforehand. In all recent iterations of the competition, different variants of the MCTS algorithm can be found among the winners of the competition.

The general video game playing competition is a recent addition to the set of game competitions [18; 17]. Like the Stanford GGP competition, submitted controllers are scored on multiple unseen games. However, unlike the Stanford GGP competition the games are arcade games inspired by 1980's video games, and the controllers are not given descriptions of the games. They are however given forward models of the games. The competition was first run in 2014, and the sample MCTS algorithm reached third place. In first and second place were MCTS-like controllers, i.e. controllers based on the general idea of stochastic tree search but implemented differently. The sample MCTS algorithm is a vanilla implementation and is described in Browne et al. [2]. The iterations of the algorithm rarely reach a terminal state due to the time constraints in the framework. The algorithm evaluates the states by giving a high reward for a won game and a negative reward for a lost game. If the game was neither won or lost, the reward is the game's score. The play-out depth of the algorithm is ten moves.

2.2 MCTS Improvements

A number of methods for improving the performance of MCTS on particular games have been suggested since the invention of the algorithm [19; 2]. A survey of key MCTS modifications can be found in Browne et al. [2]. Since the MCTS algorithm has been used in a wide collection of games, this paper investigates how the different MCTS modifications perform in general video game playing.

Some of these strategies to improve the performance of MCTS were deployed by Jacobsen et al. [8] to play Super Mario Bros. The authors created a vanilla MCTS controller for a Mario AI competition, which they augmented with additional features. To reduce cowardliness of the controller they increased the weight for the largest reward. Additionally, macro actions [15; 8] were employed to make the search go further without increasing the number of iterations. Partial expansion is another technique to achieve a similar effect as macro actions. These modifications resulted in a very good performing controller for the Mario game. It performed better than Robin Baumgarten's A* version in noisy situations and performed almost as well in normal playthroughs.

Pepels et al. [13] implemented five different strategies to improve existing MCTS controllers: A variable depth tree, playout strategies for the ghost-team and Pac-Man, long-term goals in scoring, endgame tactics and a last-good-reply policy for memorizing rewarding moves. The authors achieved an average performance gain of 40962 points, compared to the CIG'11 Pac-Man controller.

Chaslot et al. [3] proposed two strategies to enhance MCTS: Progressive bias to direct the search according to possibly time-expensive heuristic knowledge and progressive unpruning, which reduces the branching factor by removing children nodes with low heuristic value. By implementing these techniques in their Go program, it performed significantly better.

An interesting and well-performing submission to the general game playing competition is *Ary*, developed by Méhat et al. [11]. This controller implements parallelization of MCTS, in particular a "root parallel" algorithm. The idea is to perform individual Monte Carlo tree searches in parallel on different CPUs. When the framework asks for a move, a master component chooses the best action among the best actions suggested by the different trees.

Perez et al. [16] used the GVG-AI framework and proposed augmentations to deal with some of the shortcomings of the sample MCTS controller. MCTS was provided with a knowledge base to bias the simulations to maximize knowledge gain. The authors use fast evolutionary MCTS, in which every roll-out evaluates a single individual of the evolutionary algorithm and provides the reward calculated at the end of the roll-out as a fitness value. They also defined a score function that uses a concept knowledge base with two factors: curiosity and experience. The new controller was better in almost every game compared to the sample MCTS controller. However, the algorithm still struggled in some cases, for example in games in which the direction of a collision matters.

This paper uses the GVG-AI framework and builds on the sample MCTS controller. The next section will describe the GVG-AI framework in more detail.

3 GVG-AI Competition & Framework

The GVG-AI competition tries to encourage the creation of AI for general video game playing. The controller submitted to the competition webpage is tested in a series of unknown games, thereby limiting the possibility of applying any specific domain knowledge. The competition is held as part of several international conferences since 2014. Part of the GVG-AI competition is the *video game description language* (VGDL) [5; 21] that describes games in a very concise manner; all of the games used in the competition are encoded in this language. Examples are available from the GVG-AI website.

Users participate in the competition by submitting Java code defining an agent. At each discrete step of the game simulation, the controller is supposed to provide an action for the avatar. The controller has a limited time of 40ms to respond with an action. In order for the controller to simulate possible moves, the framework provides a forward model of the game. The controller can use this to simulate the game for as many ticks as the time limit allows.

For a more detailed explanation of the framework, see the GVG-AI website or the competition report [17].

4 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a statistical tree search algorithm that often provides very good results in time restricted

The CIG 2014 Training Game Set	
Aliens(G_1)	In this game you control a ship at the bottom of the screen shooting aliens that come from space. You better kill them all before they reach you! Based on *Space Invaders*.
Boulderdash(G_2)	Your objective here is to move your player through a cave, collecting diamonds, before finding the exit. Beware of enemies that hide underground!
Butterflies(G_3)	You are a happy butterfly hunter. This is how you live your life, and you like it. So be careful, you don't want them to become extinct!
Chase(G_4)	You like to chase goats. And kill them. However, they usually don't like you to do it, so try not to get caught doing that!
Frogs(G_5)	Why did the frog cross the road? Because there is a river at the other side. What would you cross the river as well? Because your home is there, and it's cosy.
Missile Command(G_6)	Some missiles are being shot to cities in your country, you better destroy them before they reach them!
Portals(G_7)	You control an avatar that needs to find the exit of a maze, but moving around is not so simple. Find the correct doors that take you to the exit!
Sokoban(G_8)	In this puzzle you must push the boxes in the maze to make them fall through some holes. Be sure you push them properly!
Survive Zombies(G_9)	How long can you survive before you become their main course for dinner? Hint: zombies don't like honey (didn't you know that?).
Zelda(G_10)	Get your way out of the dungeon infested with enemies. Remember to find the key that opens the door that leads you to freedom!

Table 1: The game descriptions of the training set from the official competition site.

The CIG 2014 Evaluation Game Set	
Camel Race(G_1)	The avatar must get to the finish line before any other camel does.
Digdug(G_2)	The avatar must collect all gems and gold coins in the cave, digging its way through it. There are also enemies in the level that kill the player on collision with him. Also, the player can shoot boulders by pressing USE two consecutive time steps, which kill enemies.
Firestorms(G_3)	The avatar must find its way to the exit while avoiding the flames in the level, spawned by some portals from hell. The avatar can collect water in its way. One unit of water saves the avatar from one hit of a flame, but the game will be lost if flames touch the avatar and he has no water.
Infection(G_4)	The avatar can get infected by colliding with some bugs scattered around the level, or other animals that are infected (orange). The goal is to infect all healthy animals (green). Blue sprites are medics that cure infected animals and the avatar, but don't worry, they can be killed with your mighty sword.
Firecaster(G_5)	The avatar must find its way to the exit by burning wooden boxes down. In order to be able to shoot, the avatar needs to collect ammunition (mana) scattered around the level. Flames spread, being able to destroy more than one box, but they can also hit the avatar. The avatar has health, that decreases when a flame touches him. If health goes down to 0, the player loses.
Overload(G_6)	The avatar must reach the exit with a determined number of coins, but if the amount of collected coins is higher than a (different) number, the avatar is trapped when traversing marsh and the game finishes. In that case, the avatar may kill marsh sprites with the sword, if he collects it first.
Pacman(G_7)	The avatar must clear the maze by eating all pellets and power pills. There are ghosts that kill the player if he hasn't eaten a power pill when colliding (otherwise, the avatar kills the ghost). There are also fruit pieces that must be collected.
Seaquest(G_8)	The player controls a submarine that must avoid being killed by animals and rescue divers taking them to the surface. Also, the submarine must return to the surface regularly to collect more oxygen, or the avatar would lose. Submarine capacity is for 4 divers, and it can shoot torpedoes to the animals.
Whackamole(G_9)	The avatar must collect moles that pop out of holes. There is also a cat in the level doing the same. If the cat collides with the player, this one loses the game.
Eggomania(G_10)	There is a chicken at the top of the level throwing eggs down. The avatar must move from left to right to avoid eggs breaking on the floor. Only when the avatar has collected enough eggs, he can shoot at the chicken to win the game. If a single egg is broken, the player loses the game.

Table 2: The game descriptions of the evaluation set from the official competition site.

situations. It constructs a search tree by doing random playouts, using a forward model, and propagates the results back up the tree. Each iteration of the algorithm adds another node to the tree and can be divided into four distinct parts. Figure 1 depicts these four steps. The first step is the **selection** step, which selects the best leaf candidate for further expansion of the tree. Starting from the root, the tree is traversed downwards, until a leaf is reached. At each level of the tree the best child node is chosen, based on the *upper confidence bound* (UCB) formula (described below). When a leaf is reached, and this leaf is not a terminal state of the game, the tree is **expanded** with a single child node from the action space of the game.

From the point of the newly expanded node, the game is **simulated** using the forward model. The simulation consist of doing random moves starting from this game state, until a terminal state is reached. For complex or, as in our case, time critical games, simulation until a terminal state is often unfeasible. Instead the simulation can be limited to only forward the game a certain amount of steps. After the simulation is finished, the final game state reached is evaluated and assigned a score. The score of the simulation is **backpropagated** up through the parents of the tree, until the root node is reached. Each node holds a total score, which is the sum of all backpropagated scores, and a counter that keeps track of the number of times the score was updated; this counter is equal to the number of times the node was visited.

4.1 Upper Confidence Bound - UCB

The UCB formula selects the best child node at each level when traversing the tree. It is based on the bandit problem, in which it selects the optimal arm to pull, in order to maximize rewards. When used together with MCTS it is often referred to as upper confidence bounds applied to trees, or UCT [2]:

$$\text{UCT} = \overline{X_j} + 2C_p\sqrt{\frac{2\ln n}{n_j}},$$

where $\overline{X}_J$ is the average score of child j. n is the number of times the parent node was visited and n_j is the number of times this particular child was visited. C_P is a constant adjusting the value of the second term. At each level of the the selection step, the child with the highest UCT value is chosen.

4.2 Exploration vs. Exploitation

The two terms of the UCT formula can be described as the balance between exploiting nodes with previously good scores, and exploring nodes that rarely have been visited [2].

The first term of the equation, $\overline{X}_J$, represents the exploitation part. It increases as the backpropagated scores from its child nodes increases.

The second term, $\sqrt{\frac{2ln(n)}{n_j}}$, increases each time the parent node has been visited, but a different child was chosen. The constant C_p simply adjust the contribution of the second term.

4.3 Return Value

When the search is halted, the best child node of the root is returned as a move to the game. The best child can either be the node most visited, or the one with the highest average value. This will often, but not always, be the same node [2].

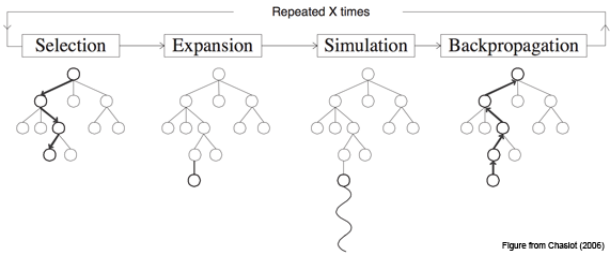


Figure 1: The main steps in the Monte-Carlo Tree Search.

4.4 Algorithm Characteristics

Anytime

One of the strengths of MCTS in a game with very limited time per turn, is that the search can be halted at anytime, and the currently best move can be returned.

Non-heuristic

MCTS only needs a set of legal moves and terminal conditions to work. This trait is very important in the GVG-AI setting, where the games are unknown to the controller. If a playout from any given state has a high probability of reaching a terminal state (win or lose), no state evaluation function needs to be used; as this is not the case in general for games in the GVG-AI set, a state evaluator is necessary.

Asymmetric

Compared to algorithms like minimax, MCTS builds an asymmetric search tree. Instead of mapping out the entire search space, it focuses efforts on previously promising areas. This is crucial in time critical application.

5 Modifications to MCTS

Here we list the particular modifications to the basic MCTS algorithm that we have investigated in this paper.

5.1 MixMax backups

Mixmax increases the risk-seeking behavior of the algorithm. It modifies the exploitation part of UCT, by interpolating between the average score and the maximum score:

$$Q \cdot \text{maxScore} + (1 - Q) \cdot \overline{X_j},$$

where Q is a value in the range $[0, 1]$. A good path of actions will not greatly affect the average score of a node, if all other children lead to bad scores. By using mixmax the good path contributes more to the average than the bad ones, thereby reducing the defensiveness of the algorithm. This modification was proposed in response to a problem observed when applying MCTS to Super Mario Bros, where Mario would act cowardly and e.g. never initiate a jump over a gap as most possible paths would involve falling into the gap. Mixmax backups made Mario considerably bolder in the previously cited work [8].

For the experiments with mixmax in this paper, the Q value was set to 0.1. The Q value was determined through prior experimentation.

5.2 Macro Actions

As stated previously, each action has to be decided upon in a very short time-span. This requirement can often lead to search tree with a limited depth, and as such only the nearest states are taken into consideration when deciding on the chosen action. Macro actions enable a deeper search, at the cost of precision. Powley et al. have previously shown that this is an acceptable tradeoff in some continuous domains [20].

Macro actions consists of modifying the expansion process, such that each action is repeated a fixed number of times before a child node is created. That is, each branch corresponds to a series of identical actions. This process builds a tree that reaches further into the search space, but using coarser paths.

5.3 Partial Expansion

A high branching factor relative to the time limit of the controller results in very limited depth and visits to previously promising paths. Even though a child might have resulted in a high reward, it will not be considered again, before all other children have been expanded at least once. The *partial expansion* modification allows the algorithm to consider "grandchildren" (and any further descendants) of a node before all children of that node have been explored. This allows for a deeper search at the cost of exploration. In the Mario study, Partial Expansion was useful combined with other modifications [8].

5.4 Reversal Penalty

A new MCTS modification introduced in this paper is *UCT reverse penalty*. A problem with the standard MCTS controller is that it would often just go back and forth between a few adjacent tiles. This oscillation is most likely due to the fact that only a small amount of playouts are performed each time MCTS is run, and therefore a single, or a few high scoring random playouts completely dominate the outcome. Additionally, when there are no nearby actions that result in a score increase, an action is chosen at random, which often also leads to behaviors that move back and forth. Instead, the goal of UCT reverse penalty is to create a controller that explores more of the given map, without increasing the search depth. To achieve this the algorithm adds a slight penalty to the UCT value of children that lead to a recently visited level tile (i.e. "physical position" in the 2D game world). However, the penalty has to be very small so it does not interfere with the normal decisions of the algorithm, but only affect situations in which the controller is going back and forth between fields. This modification is similar but not identical to exploration-promoting MCTS modifications proposed in some recent work [12; 14].

In the current experiments, a list of the five most recently visited positions is kept for every node, and the penalty is 0.05; when a node represents a state where the avatar position is one of the five most recent, its UCT value is multiplied by 0.95.

6 Experiments

The experiments in this paper are performed on the twenty games presented in Table 3 and Table 4. In order to make our results comparable with the controllers submitted to the general video game playing competition, we use the same software and scoring method. Each game is played five times for each combination of MCTS modifications, one playthrough per game level. This configuration follows the standard setup for judging competition entries. Each level has variations on the locations of sprites and in some games variations on non-player character (NPC) behavior. There are nine different combinations plus the vanilla MCTS controller, which gives 900 games played in total. The experiments were performed on the official competition site, by submitting a controller following the competition guidelines. All combinations derive from the four modifications explained in the previous section: mixmax scores, macro actions, partial expansion and UCT reverse penalty.

Two measures were applied when analyzing the experiments: the number of victories and the score. The GVG-AI competition weighs the number of victories higher than the achieved score when ranking a controller; it is more important to win the game rather than losing the game with a high score. In both Table 3 and 4 the scores are normalized to values between zero and one.

7 Results

The top three modifications are UCT reverse penalty, mixmax and partial expansion. According to the total amount of wins, the MCTS controller with UCT reverse penalty is the best performing controller. Thirty wins in the training set and seventeen wins in the validation set. The number of wins is slightly better than the number of wins of the vanilla MCTS controller (27). However, the vanilla controller receives a higher number of points (756), compared to the number of points of the UCT reverse penalty modification (601). Videos of the UCT reverse penalty controller can be found here: `http://bit.ly/1INlpF9`.

Compared to the vanilla MCTS controller, the mixmax modification alone does not increase the total amount of wins or points. It does however improve the performance in some games. In Missile Command the vanilla controller scored higher than mixmax, but they have an equal chance of winning. In the game Boulderdash, mixmax wins more often but scores less points. By applying mixmax, the controller wins games faster; points are scored whenever a diamond spawns (time based) and when the controller collects diamonds. Therefore the faster the win, the less points.

Combining UCT reverse penalty and mixmax shows promising results (Table 4). This controller was the highest scoring and most winning controller looking at the total values. It was the only controller winning any playthroughs in the game Eggomania. The gameplay is characterized by a controller that moves from side to side, whereas the other controllers only move as far as they can "see".

Interestingly, whenever a combination of modifications contains macro actions, the controller performs badly, both in terms of total score and total wins. As stated previously macro actions enables a deeper search, at the cost of precision. This ability enables macro controllers to succeed in games like Camelrace; as soon as it finds the goal the agent will move to it. The other MCTS controllers fail in Camelrace because they do not search the tree far enough. However, the MCTS modifications with macro action lose almost every other game type due to lack of precision. For example, in Pac-Man-like games, it searches too deep, likely moving past the maze junctions and never succeeding in moving around properly. The macro action experiments were done with a repeat value of three (i.e. using the same action three times in a row). The repeat value

has a profound effect on how the controller performs, and is very domain specific. The repeat value for Camelrace should be very high, but in Pac-Man it should be very low for it to not miss any junctions.

The game Camelrace is uncovering one major problem of the MCTS controllers; the play-out depth is very limited, which is a problem in all games with a bigger search space. If the controller in Pac-man clears one of the corners of the maze, it can get stuck in that corner, therefore never reaching nodes that give points. The only controller that wins any games in Pac-Man is UCT reverse penalty and its combination with mixmax. UCT reverse penalty without mixmax scores most points, but with mixmax it wins all playthroughs and is ranked second in achieved score. The depth cannot be increased due to the time-limitations in the competition framework.

In the game Frogs, the avatar shows problematic behavior. The avatar has to cross the road quickly without getting hit by the trucks in the lanes. Most roll-outs are unable to achieve this. The most common behavior observed is a controller that moves in parallel to the lanes and is never crossing it. No controllers are able to win all the playthroughs, but controllers using mixmax scores or UCT reverse penalty sometimes win.

When comparing our results with the ranking on the official competition site, our controller is performing better on the validation set. The sampleMCTS scores 37 points and wins 16 of 50, where our controllers scores 75 points and wins 20 of 50. This places the controller on seventh place, four places higher than sampleMCTS. In the training set our controller scores less points, but wins three games more than the sampleMCTS. This places our controller on tenth place, three places lower than sampleMCTS.

8 Discussion and Future Work

According to our experiments the [UCT reverse penalty, mixmax] combination was the one that performed best overall, and the only one that convincingly beat Vanilla MCTS on the validation set. It should be noted that while we used the official scoring mechanism of the competition, higher number of playthroughs might have been preferable given the variability between games. Several games contains NPCs, and those have very different behaviors. Additionally, their behaviors are not only different per game, but also per playthrough. In games like Pac-Man (G_7 in the validation set), the enemy ghosts behave very stochastically. Because of this stochastic behavior, the results of five playthroughs will vary even using the same controller.

The presented results show that each MCTS modification only affects subsets of games, and often different subsets. One could argue that the sampleMCTS controller in the framework is rather well-balanced.

One could also argue that the fact that no single MCTS modification provides an advantage in all games shows that the set of benchmark games in GVG-AI provides a rich set of complementary challenges, and thus actually is a test of "general intelligence" to a greater degree than existing video game-based AI competitions. It remains to be seen whether any modification to MCTS would allow it to perform better across these games; if it does, it would be a genuine improvement across a rather large set of problems.

9 Conclusion

This paper investigated the performance of several MCTS modifications on the games used in the General Video Game Playing Competition. One of these modifications is reported for the first time in this paper: UCT reverse penalty, which penalizes the MCTS controller for exploring recently visited children. While some modifications increased performance on some subset of games, it seems that no one MCTS variation performs best in all games; every game has particular features that are best dealt with by different MCTS variations. This confirms the generality of AI challenge offered by the GVG-AI framework.

References

[1] Yngvi Bjornsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):4–15, 2009.

[2] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[3] G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[4] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[5] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. 2013.

[6] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[7] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62, 2005.

[8] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte Mario: Platforming with MCTS. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 293–300, New York, NY, USA, 2014. ACM.

[9] Niels Justesen, Tillman Balint, Julian Togelius, and Sebastian Risi. Script-and cluster-based UCT for StarCraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[10] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, J-B Hoock, Arpad Rimmel, F Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

[11] Jean Méhat and Tristan Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.

Training set (Normalized scores)																						
Controller	**G_1**		**G_2**		**G_3**		**G_4**		**G_5**		**G_6**		**G_7**		**G_8**		**G_9**		**G_10**		**Total points**	**Total wins**
	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v		
sampleMCTS	**1.00**	5	**1.00**	1	0.89	5	0.88	4	0.83	1	0.69	4	0.67	2	0.88	1	0.44	2	0.24	2	**756**	27
[mixmax]	0.90	4	0.53	3	0.38	5	0.92	4	0.33	0	**1.00**	4	0.33	1	**1.00**	1	0.81	3	0.41	1	727	26
[macroActions]	**0.11**	0	0.26	1	**0.08**	4	**0.00**	1	**0.00**	0	**0.03**	2	**0.00**	0	0.38	0	**0.17**	1	**0.19**	1	448	10
[partialExp]	0.91	5	0.47	1	**1.00**	5	0.62	3	**0.00**	0	0.80	5	0.33	1	**1.00**	2	0.29	2	0.54	2	707	26
[mixmax,macroActions]	**0.00**	4	**0.15**	0	0.54	4	**0.08**	1	**0.00**	0	0.29	2	**0.00**	0	0.50	0	**0.00**	0	**0.11**	0	453	11
[mixmax,partialExp]	0.47	2	0.47	0	0.41	4	0.23	2	0.83	1	0.69	4	0.33	1	0.50	0	0.51	3	0.57	3	615	20
[macroActions,partialExp]	0.84	5	**0.15**	1	0.27	5	0.62	3	**0.00**	0	0.43	2	**1.00**	3	0.25	0	0.23	1	0.68	3	607	23
[macroActions,mixMax,partialExp]	0.27	3	**0.00**	0	0.43	5	0.23	1	**0.00**	0	**0.00**	1	**0.00**	0	0.38	0	0.21	1	**0.00**	0	481	11
[UCT reverse penalty]	**0.15**	1	0.69	3	**0.14**	5	**1.00**	5	**1.00**	2	0.69	5	0.67	2	**1.00**	1	0.56	2	0.62	4	601	**30**
[UCT reverse penalty, macroactions]	0.65	3	**0.13**	0	0.35	5	0.50	4	**0.00**	0	0.31	3	0.33	1	**0.00**	0	**0.09**	1	0.41	1	552	18
[UCT reverse penalty, mixMax]	0.82	5	0.76	3	**0.00**	5	**1.00**	5	**0.00**	0	0.69	3	**0.00**	0	0.75	1	**1.00**	2	**1.00**	5	730	29

Table 3: Results on the CIG2014 training set. Scores are normalized between 0 and 1 - G_1: Aliens , G_2: Boulderdash , G_3: Butterflies , G_4: Chase , G_5: Frogs , G_6: Missile command , G_7: Portals , G_8: Sokoban , G_9: Survive zombies , G_10: Zelda

Validation set (Normalized scores)																						
Controller	**G_1**		**G_2**		**G_3**		**G_4**		**G_5**		**G_6**		**G_7**		**G_8**		**G_9**		**G_10**		**Total points**	**Total wins**
	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v	p	v		
sampleMCTS	**0.00**	0	0.24	0	0.63	2	0.48	2	**1.00**	1	0.62	3	0.60	0	0.54	5	0.82	3	**0.09**	0	7545	16
[mixmax]	**1.00**	1	0.75	0	0.44	0	0.85	4	0.46	0	0.82	2	0.68	0	0.46	5	0.92	4	**0.04**	0	6849	16
[macroActions]	**1.00**	1	**0.03**	0	0.81	0	**0.00**	2	**0.06**	0	0.57	1	**0.08**	0	**0.18**	0	0.30	1	**0.00**	0	2656	5
[partialExp]	**0.00**	0	**0.00**	0	**0.13**	1	**1.00**	3	0.80	0	0.68	3	0.70	0	0.73	5	0.72	3	**0.15**	0	9849	15
[mixmax,macroActions]	**1.00**	1	0.07	0	0.81	0	**0.02**	1	0.29	0	**0.00**	2	**0.00**	0	**0.00**	0	**0.00**	1	**0.02**	0	515	5
[mixmax,partialExp]	**0.00**	0	0.72	0	0.31	0	0.63	3	0.60	0	**0.13**	3	0.20	0	**0.19**	3	0.70	4	**0.04**	0	3172	13
[macroActions,partialExp]	**1.00**	1	0.27	0	0.75	0	0.66	4	**0.00**	0	0.57	2	0.51	0	**0.09**	1	0.56	2	**0.00**	0	2364	10
[macroActions,mixMax,partialExp]	**1.00**	1	0.52	0	0.75	0	0.53	3	0.23	0	0.54	2	**0.15**	0	**0.09**	1	0.32	1	**0.02**	0	1968	8
[UCT reverse penalty]	**1.00**	1	0.62	0	**0.00**	0	0.46	5	0.26	1	0.69	3	**1.00**	1	0.45	3	**1.00**	3	**0.05**	0	6873	17
[UCT reverse penalty, macroactions]	**1.00**	1	0.31	0	**1.00**	1	0.26	4	**0.11**	0	0.68	0	0.73	0	**0.00**	0	0.75	2	**0.01**	0	1460	8
[UCT reverse penalty, mixMax]	**1.00**	1	**1.00**	0	0.69	2	0.52	4	0.23	0	**1.00**	1	0.98	5	**1.00**	4	0.96	1	**1.00**	2	**13358**	**20**

Table 4: Results on the CIG2014 validation set. Scores are normalized between 0 and 1 - G_1: Camel race, G_2: Digdug, G_3: Firestorms, G_4: Infection, G_5: Firecaster, G_6: Overload, G_7: Pacman, G_8: Seaquest, G_9: Whackamole, G_10: Eggomania

[12] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. General video game evaluation using relative algorithm performance profiles. In *Applications of Evolutionary Computation*, pages 369–380. Springer, 2015.

[13] Tom Pepels and Mark H. M. Winands. Enhancements for Monte Carlo tree search in Ms Pac-Man. In *CIG*, pages 265–272. IEEE, 2012.

[14] Diego Perez, Jens Dieskau, Martin Hünermund, Sanaz Mostaghim, and Simon M Lucas. Open loop search for general video game playing. 2015.

[15] Diego Perez, Edward J Powley, Daniel Whitehouse, Philipp Rohlfshagen, Spyridon Samothrakis, Peter Cowling, Simon M Lucas, et al. Solving the physical traveling salesman problem: Tree search and macro actions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(1):31–45, 2014.

[16] Diego Perez, Spyridon. Samothrakis, and Simon Lucas. Knowledge-based fast evolutionary MCTS for general video game playing. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8, Aug 2014.

[17] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon Lucas, Adrien Couëtoux, Jeyull Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

[18] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Lucas Simon. The general video game AI competition - 2014. `http://www.gvgai.net/`, 2014. [Online; accessed 25-December-2014].

[19] Edward J Powley, Peter I Cowling, and Daniel Whitehouse. Information capture and reuse strategies in Monte Carlo tree search, with applications to games of hidden information. *Artificial Intelligence*, 217:92–116, 2014.

[20] Edward J. Powley, Daniel Whitehouse, and Peter I. Cowling. Monte Carlo tree search with macro-actions and heuristic route planning for the multiobjective physical travelling salesman problem. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[21] Tom Schaul. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[22] Julian Togelius. How to run a successful game-based AI competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

A Neural Network Approach to Model Learning for Stochastic Discrete Environments

Alex Braylan *braylan@cs.utexas.edu*
Risto Miikkulainen *risto@cs.utexas.edu*
The University of Texas at Austin

Abstract

A model learning agent is one which learns its environment's state transition probabilities and uses them to inform its operation. This skill seems critical for incorporating human-level reasoning into decision-making. We introduce a neural network system that learns to sample approximately from an observed transition probability function and use generated predictions to inform action decisions. We demonstrate its effectiveness and resilience in stochastic toy domains where the reward function may be absent or changing before and after learning. We also show how these neural networks can learn more compact and practical model representations.

1 Introduction

Neural networks are known to be universal function approximators [Hornik *et al.*, 1989] and have demonstrated some ability to model various low-level and perceptual cognitive functions. This kind of cognition involves producing an output from a given input. Sensory input – a representation of the thinking agent's current state – enters the brain and leads to the production of an output such as a motor action, a classification decision, a prediction, or a feeling.

Reinforcement learning is a field of artificial intelligence in which the anticipation of rewards guides an agent's learning of such input-output functions. As an intelligent agent explores its environment it learns how to choose outputs given its inputs in such a way as to maximize the reward (or minimize the cost) delivered by the environment [Barto, 1998].

Two classical families of reinforcement learning are: *policy-based* and *value-based*. Policy methods involve searching through the universe of policies, where a chosen action is a function of the current state, to find the ones that collect the most reward. Value methods learn to represent future expected reward using functions of a state and action.

On the other side of the cognitive spectrum from intuitive thinking is reflective reasoning. While neural networks are a popular tool when modeling intuitive cognition, reflective cognition is usually expressed using symbolic logic due to its discrete, grammatical, and compositional properties. However, there are many approaches to implementing such logic using neural architectures [Garcez *et al.*, 2008]. Neural networks bring several benefits such as parallel processing, fault tolerance, holographic representation, generalization ability, and a broad range of learning algorithms. This paper will explore how to use neural networks to learn and represent a model of an environment so as to inform high-level reasoning operations.

2 Model Learning Without Rewards

A model of an environment is often described as a *Markov decision process*, which is a system in which transitions between states are probabilistic and depend on the state and on the actions taken by an agent. Formally, it is a set of states S, actions A, and transition probabilities $P(s'|s,a)$ for all $s, s' \in S, a \in A$ such that $\sum_{s' \in S} P(s'|s,a) = 1$.

If an intelligent agent can guess these transition probabilities with some accuracy, then we may say it has learned an approximate model of its environment. With this model, its behavior can adapt to a changing specification of reward without having to explore or learn anything new about its environment. In other words it can perform a variety of tasks which it was not specifically trained to perform. This is in contrast to the methods of reinforcement learning that learn based on what is relevant to a preexisting reward function.

A sufficient condition to say an agent has learned an approximate model of its environment is when it can sample from a distribution approximately similar to the real transition probability distribution. With this ability it can make predictions and inferences, calculate statistical moments, and generate an arbitrarily large finite state machine representation of the reachable state space. An agent that predicts state transitions rather than just rewards is interesting from a neuroscience perspective as well, given the discovery of distinct neural signatures in humans performing state transition prediction tasks versus reward prediction tasks [Gläscher *et al.*, 2010]. The two modes are not mutually exclusive and may work sequentially or in unison.

2.1 Factored Representations

Model learning agents could conceivably maintain full representations of the estimated state transition probabilities in tables. However, this approach is invalidated by susceptibility to combinatorial explosions in the state or action space. Consider a state as composed of binary variables $x_1, x_2, \ldots, x_j$ or

an action composed of binary variables $y_1, y_2, \ldots, y_k$. Then the number of state-action combinations is $2^j 2^k$. The size of the stored table grows exponentially, thus this approach can be computationally infeasible for large values of j or k.

One common solution to this problem is to work directly with the variables composing the states and actions rather than the states and actions themselves. For example, an approximate value function can be trained to estimate future reward based on the variables of a current state and candidate action [Powell, 2007]. Alternatively, a neural network can be evolved to output favorable actions given state variables as inputs, e.g. [Stanley and Miikkulainen, 2002]. Figures 1 and 2 illustrate typical architectures of a neural networks designed for value learning and policy learning, respectively.

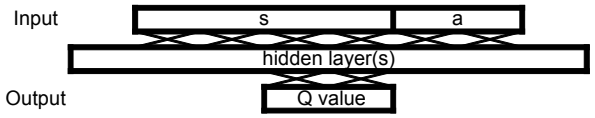


Figure 1: Layout of a value approximation network

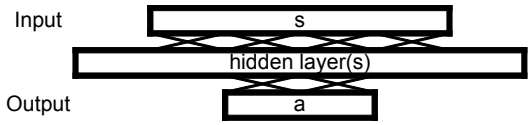


Figure 2: Layout of a policy network

In this paper, we also take a factored approach to compress the transition probabilities for the purpose of sampling. We use feed-forward neural networks that represent the state and action variables with the nodes in the input and output layers. Specifically, the input takes in observed or estimated values of the variables, and the output represents approximated probabilities. Thus we reduce the number of parameters to work with to $(2 * j + k) * h + h * h * n + j * h + (j + h) * b$ where h is the number of hidden nodes in n hidden layers and b is 1 or 0 depending on the existence of a bias (threshold) input. If h is chosen to be at worst polynomial in the number of input and output nodes, the result is a number of parameters that is only polynomial in the number of state variables rather than the exponential complexity of tabulating all states and actions. Later in this paper we discuss using *modularization* to reduce complexity even further.

Dynamic Bayesian Networks (DBN) may be used to represent factored-state transitions [Murphy, 2002]. These are directed graphical models with adjacent layers representing subsequent factored states. Bayesian networks such as DBNs can use neural networks to encode the conditional probabilities represented by their nodes [Bengio and Bengio, 1999]. Using a neural network encoding to learn and sample probabilities has advantages and disadvantages, some of which we investigate in this paper.

3 Neural Approximation of Model Transition Probabilities

Generative models based on neural network architectures have been previously used to represent probability distributions underlying the variables of a time series. Restricted Boltzmann Machines (RBMs) are one tool to learn and sample from conditional relationships in state sequences [Sutskever and Hinton, 2007]. RBMs are stochastic recurrent networks where weights between visible and hidden units are bidirectional and symmetrical [Hinton and Sejnowski, 1986].

Another family of generative neural network methods are the Neural Autoregressive Distribution Estimator (NADE) [Larochelle and Murray, 2011] and its relatives [Gregor *et al.*, 2013]. These seem to be quite state-of-the-art and efficient methods for modeling data. However, there is relatively scarce examination of their effectiveness for modeling transition probabilities in MDPs.

This paper considers an approach specialized for this MDP model learning problem, based on feed-forward networks with no recurrent or lateral connections. Our goal is to produce a neural network that takes an input representing a condition and produces an output that can be used to sample from the conditional probability distribution given that input. First we demonstrate how to accomplish this for multiple Bernoulli random variables distributed independently of each other.

3.1 Predictor Network

Consider the conditional probability $P(Y|X)$ where X consists of $x_1, x_2, \ldots, x_j$, Y consists of $y_1, y_2, \ldots, y_k$, x and y are random draws from $\{0, 1\}$, and $P(y_h, y_i) = P(y_h)P(y_i)$ for all h, i (independence assumption). The distribution $P(Y|X)$ can be encoded by a neural network by training in the following straightforward manner. Starting with a neural network with input layer of size j and output size k, for all known instantiations of X, clamp each $x \in X$ to the input layer, set the target output for each $y \in Y$ to $P(y|X)$ and perform a standard training algorithm such as backpropagation [Rumelhart *et al.*, 1988]. Learning may alternatively be implemented in an on-line manner directly from observed data by setting each target output to y rather than $P(y|X)$ and training over all observations. This on-line method automatically learns to represent observed data frequencies as probabilities and can be useful in environments with changing dynamics, but it is slower and sensitive to the order of training.

We refer to this neural network as the *predictor network* and use it to generate samples from $P(Y|X)$ when the variables in Y are independent. This is done by setting X as the input to the network, feeding activations forward, and sampling from the Bernoulli probabilities represented by the activations of the output nodes. The output activations should be bound by [0,1] in order to represent valid probabilities, but this should be roughly achieved through the training process if not ensured by the activation function.

In the context of learning a model of a discrete stochastic environment, the predictor network can be used to learn the transition probabilities $P(s'|s, a)$ by setting the concatenations of state s and action a variables as inputs and the subse-

quent state s' variables as target outputs. The model learning agent can collect this data by exploring its environment.

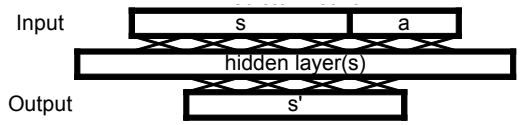


Figure 3: Layout of a predictor network

3.2 Conditioner Network

We now consider the case where the random variables are not conditionally independent of one another, *i.e.* $P(y_i, y_j) \neq P(y_i)P(y_j)$ for some i, j. The predictor network alone is no longer sufficient to generate realistic samples. To illustrate, consider the following two example environments with two state variables:

- In environment 1, the only two next states s' from s are (0,0) and (1,1).
- In environment 2, the only two next states s' from s are (1,0) and (0,1).

Suppose in both environments the probability of transitioning to each s' from s is 50%. For both environments, a predictor network will learn to output approximately (0.5, 0.5) given input s, and all four states will be sampled at equal rates for both environments. In order to only sample states that are reachable in the learned environment, the output from the predictor network is fed into a *conditioner* network to arrive at more appropriate outputs.

The conditioner network applies learned relationships between model variables and can be trained simultaneously with the predictor network. It works analogously to a Gibbs sampler by taking advantage of conditional probabilities. Gibbs sampling involves sampling a joint distribution of random variables by iteratively changing the value of one variable at a time according to its probability conditioned on the values of the other variables. In our implementation, the initial guess provided by the predictor network reduces the need for a *burn-in* period during which samples are rejected to give the procedure time to converge. This in turn allows a short fresh initialization before each sample is accepted which can alleviate a difficulty faced by Gibbs samplers when data features are highly correlated as in the example above [Justel and Peña, 1996].

The procedure to produce one sample works as follows:

1. Feed-forward conditioning evidence X into predictor network.
2. Take outputs of predictor network as first guess $\bar{Y}^{(t=0)}$.
3. Feed-forward concatenation of X and $\bar{Y}^{(t)}$ into conditioner network.
4. Take output i of conditioner network as probability of value for next guess $\bar{y_i}^{(t+1)}$.
5. Repeat last two steps iterating through variables i in random order
6. Accept sample $\bar{Y}$.

In order for this to work, each output node i of the conditioner network must come to represent the probability $P(\bar{y}_i|X, \bar{Y}_{\neq i})$. Training therefore involves providing X and Y to the input and Y as the target output for each observed data point. If the input activation representing variable y_i can propagate to the output node representing the same y_i, then the learning of relationships between variables will fail because each variable's relationship with itself dominates learning. Therefore it is necessary to prune all weights that can enable this effect. One way to do this when there is a single hidden layer between the input and output layers is to group hidden nodes into a number of groups equal to the number of Y variables, prune all weights from each input node i to all hidden nodes in group i, and prune all weights from all hidden nodes not in group i to each output node i.

A model learner can use the predictor network and the conditioner network together to generate approximately correct samples from $P(s'|s, a)$ given appropriate training in a Markov decision process.

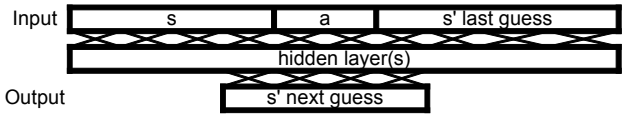


Figure 4: Layout of a conditioner network

3.3 Experiments: Sampling Accuracy

To test the neural network sampling methods described above we ran several tests in which the model learner was presented with observations of Bernoulli-distributed binary variables $[X, Y]$ and subsequently asked to infer transition probabilities $P(Y|X)$. In each trial the conditional probability originally observed from the data was compared against an inference made by the predictor network alone and against an inference made by the combined predictor and conditioner sampling algorithm.

Data Generation

In order to generate experiment data that could be considered representative of various flavors of state transition functions, the parameters that were allowed to vary were:

1. State-space size: The number of possible states is exponential in the number of dimensions of the variables.
2. Intra-variable correlation: The correlation between dimensions in each output impacts the difficulty of modeling joint probability.
3. Transition sparsity: The number of possible outputs per input is one marker of system stochasticity.

The following procedure was followed to produce experiment data according to those parameters:

1. Determine size of data N: Choose the number of data points generated in each experiment to be $N = bcd^2$ where c is a parameter indicating the number of possible

outputs for each input considered, d is the dimensionality, and b is an optional multiplier for reducing large data sets.

2. Set inputs X: Enumerate all 2^d possible inputs, shuffle them, and select the subset $X^{(i)}, i \in [1 \dots d^2]$ for the data set to limit computational expense.
3. Decide intra-variable correlation: This parameter could be either $high$ or low. In the case it is set to $high$, for each selected input, create a correlation-inducing matrix $M^{(i)}$. This is not a correlation matrix, but it is a d by d symmetrical matrix with entries randomly uniformly drawn between -1 and 1. We use it to induce correlation between the values in the different dimensions of an output.
4. Set outputs Y: For each selected input $X^{(i)}$,
   (a) Draw values of an output $Y^{(ij)}, j \in [1 \dots c]$ from a Bernoulli distribution with $p = 0.5$ if intra-variable correlation is low.
   (b) If intra-variable correlation is $high$, then set values for each dimension of $Y^{(ij)}$ in shuffled order to $y_k^{(ij)}$ where $k \in [1 \dots d]$. $y_0^{(ij)}$ is drawn from a Bernoulli distribution with $p = 0.5$. Every subsequent $y_k^{(ij)}$ is drawn from a Bernoulli distribution with $p = 0.5$ with probability $1 - |r|$, $r = M_{k,k-1}^{(i)}$. Otherwise it is set to $x_k^{(i)}$ (the same-dimension value in $X^{(i)}$) if $r > 0$ or $-x_k^{(i)}$ if $r < 0$.
5. Add each data point $[X^{(i)}, Y^{(ij)}]$ created in the above manner to the data set.

Training and Testing

Two neural network sampling methods were tested: the predictor network sampler alone, and the combined predictor and conditioner sampler. Both networks consisted of an input, hidden, and output layer. The number of nodes in the predictor network's hidden layer was $2d+1$, and for the conditioner network was $2d^2 + 1$. The neural networks were trained on the data set for 250 iterations or until convergence of error, reshuffling the data on each iteration.

For each input $X^{(i)}$ presented, the samplers were asked to generate c outputs $\bar{Y}^{(ij)}$. We measure each $P(Y^{(ij)}|X^{(i)})$ and $P(\bar{Y}^{(ij)}|X^{(i)})$ for all i and j. We use the Pearson correlation coefficient between these two as a measure of how closely the recovered estimates relate to the original transition (conditional) probabilities.

Results

The details of the trials are as follow:

1. 2-d vectors, 4 possible outcomes of Y for each X
   (a) low intra-variable correlation
   (b) high intra-variable correlation
2. 8-d vectors, high intra-variable correlation
   (a) 3 possible outcomes of Y for each X
   (b) 5 possible outcomes of Y for each X
   (c) 10 possible outcomes of Y for each X
3. 32-d vectors, high intra-variable correlation, 5 possible outcomes of Y for each X

Following is the correlation of the observed transition probabilities against each of the two samplers' inferences:

Trial:	1a	1b	2a	2b	2c	3
Predictor	0.80	0.51	0.42	0.27	0.41	-0.12
Combined	0.93	0.89	0.89	0.79	0.67	0.56

Figures 5 through 7 show observed probabilities (x-axis) versus inferences (y-axis). The light grey diamonds are by the predictor network alone and the black "x"s are by the combined sampler. The combined sampler is the choice method and thus the black "x" scatter plots should be considered more relevant.

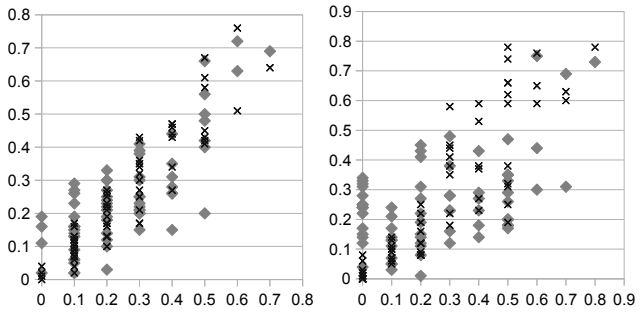


Figure 5: Scatterplots of Trials 1a and 1b

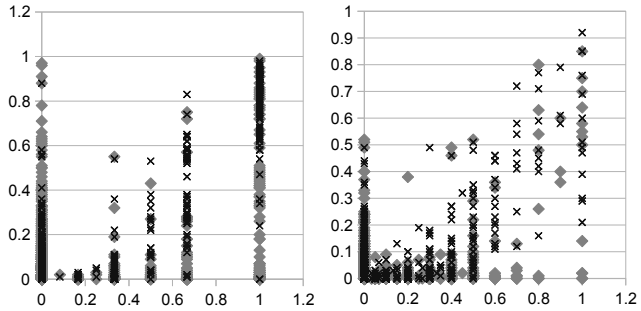


Figure 6: Scatterplots of Trials 2a and 2c

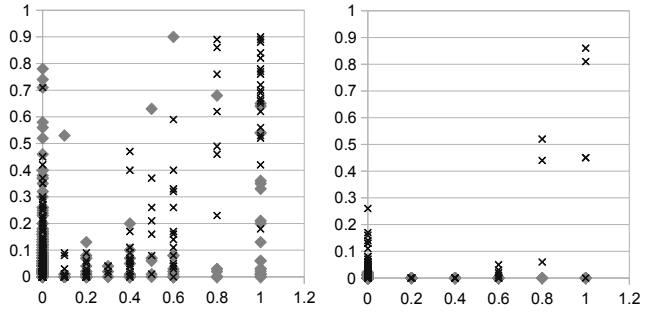


Figure 7: Scatterplots of Trials 2b and 3

3.4 Discussion: Sampling Accuracy

A few conclusions can be made from the above results. First, the combined sampler is always superior to using the predictor network alone. If the random variables in the data are uncorrelated, as seen in Figure 5, this difference is milder. As correlation between variables is normal in most environments, this result indicates the usefulness of the conditioner network.

Second, performance deteriorates as the total dimensionality increases as seen in Figure 7. This is overall not surprising as the larger networks become more difficult to train, and as the amount of training data presented to them in our experiments is smaller relative to the total size of the possible state space.

Third, sparser data is easier to model as seen in Figure 6. We are not overly worried about the decrease in accuracy as the number of possible transitions increases, because high-dimensional environments typically have a small number of possible transitions from each state (low stochasticity) relative to the number of dimensions.

Finally, the graphs illustrate a tendency for our method to overestimate transitions with observed probabilities equal to zero and underestimate transitions with observed probabilities equal to one. This is because the neural network weight updates are only caused by occurrences of – not absence of – observation. The result is for the model learner to believe more transitions to be possible than might actually be possible. More training or more observations can help push the estimated probabilities closer to their frequency values, but when less data is used to inform the agent's actions this seems to be the normal and desirable phenomenon of how neural networks approximate Bayesian a posteriori probabilities [Richard and Lippmann, 1991]. In other words the agent starts with a prior model of its environment in which all transitions have roughly equal probability, then updates this model through observation, rather than using the exact observed transition frequencies as its model probabilities.

It must be noted that the neural networks used were very basic single-layer networks trained using a very basic back-propagation algorithm. There should be much room for improvement in accuracy just from implementation tuning. Overall, the combined neural network method demonstrates an ability to approximately form reasonable joint probability estimates from its experiences.

4 Action Selection

The above neural network methods illustrate how an agent can learn to sample transition probabilities and make predictions about its environment, but we have not yet addressed the mechanisms for using such ability to act intelligently in the modeled environment. Although the focus of this paper is on model learning rather than action selection, this issue must be addressed in order to evaluate the usefulness of our model learning method. Separation of reward and policy modules from the environment prediction modules seems like a natural and beneficial architecture for an intelligent agent.

4.1 Monte Carlo Tree Search (MCTS)

The goal in reinforcement learning problems is to maximize the expected sum of immediate reward and future reward. In a *Monte Carlo Tree Search* (MCTS), several simulations are rolled out up to a bounded number of steps using the transition sampling method. At the end of each simulation the reward is counted, and the action which produced the greatest reward is selected. This approach involves a trade-off between speed and quality of action selection. There is a great wealth of variations and enhancements to this method [Browne *et al.*, 2012], but all of our experiments use a basic implementation of MCTS. Taking a bounded number of samples rather than fully expanding the future state space allows the planner to choose near-optimal actions while scaling to environments with large state and action space, with complexity only being exponential in the horizon length [Kearns *et al.*, 2002]. A benefit to having the action selection mechanism separated from the model learner is that it becomes free to evaluate and evolve independently.

4.2 Error Accumulation

A common problem with using a single-step transition model to plan multiple steps ahead is that the prediction error compounds increasingly over longer time horizons. One alternative, if the agent wants to plan n steps in advance, is to learn a model for transition probabilities between states separated by n steps given a string of n actions. Each action in the string occupies an additional group of inputs in the neural networks. If the planner requires forecasts over different time horizons, it can use several separate models corresponding to each desired value of n, an approach known as *independent value prediction* [Cheng *et al.*, 2006]. While this avoids the problem of error accumulation, the true models for longer-horizon forecasts can be more difficult to learn.

Another response to the problem of error accumulation is to adapt in real time by revising the forecast at each time step [Moriguchi and Lipson, 2011]. This way, although an action taken at one time step depends on an error-prone long forecast, the agent's ability to periodically stop and recalculate its short-term actions partially mitigates the risk of a bad trajectory. The agents in our experiments all make use of this real-time re-planning.

4.3 Experiments: Decision-making in Stochastic Environments

To test the integration of our model learning algorithm with a MCTS planner, we introduce several toy domains which are small stochastic games with varying reward functions. The first mini-game we present is *Geyser Runner*. In this game the player controls a man who can move right or left and collects rewards proportional to how far to the right he is standing. However, geysers which hurt the man (negative reward) can shoot out from the ground on the right half of the screen.

In one mode of the game, the geysers shoot deterministically in a predictable order. The agent who learns this game always finds a way to move all the way to the right, dodging left and back only when the geyser is about to shoot him. In the other mode of the game, however, geysers shoot randomly. The agent who learns the stochastic geyser game does

Figure 8: Geyser Runner

not time his movement according to the order of the geysers but rather will either go straight right or sit in the safe zone indefinitely depending on the difference in reward collected by moving and the cost of being hit by a geyser.

This test demonstrates a basic ability of the integrated model learner and MCTS planner to behave reasonably in environments with different degrees of stochasticity. Although many simple single-player games may exhibit little or no stochasticity, we believe robustness to uncertainty is important for scaling to more complex games, especially ones with human or human-like adversaries whose behavior can be impossible to model deterministically.

4.4 Experiments: Reward-free Learning

Our second mini-game demonstrates how a model-learner can use knowledge gained about its environment in the absence of rewards to subsequently inform its behavior after goals are introduced. In *Maneuverer*, the agent wanders a grid world with randomly generated walls. The agent may be given a target location where he can collect a reward. Once reached, the target as well as the walls are randomly relocated.

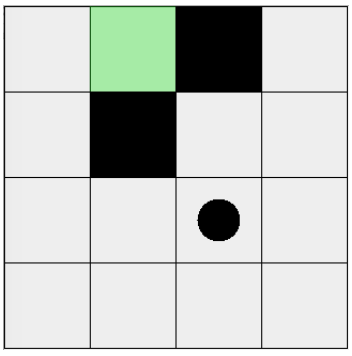


Figure 9: Maneuverer

The model learner is trained by randomly exploring without seeing any rewards. Given proper training and a sufficiently long planning horizon, it can subsequently navigate each new grid to a designated goal continuously. This game demonstrates our hybrid agent's ability to learn about its environment prior to observing a reward and subsequently use that knowledge to guide its actions once a reward function is introduced. It also shows how this separation of the reward function from learning allows the agent to carry on acting without relearning every time the reward function changes.

Separating the reward from the model in this way frees the agent to potentially learn goals independently – and thereby using fewer parameters – from the learning of other modules. Learning *intrinsic motivation* enables an agent to engage in play behavior and exploration, and can be driven, for instance, by an evolutionary algorithm [Singh *et al.*, 2010].

5 Modularization

Using a factored representation of states and actions allows us to confine the number of necessary parameters to a polynomial bound. However, growth in the number of variables can still become a substantial expense, as we have seen how inference accuracy decreases with size. A larger number of parameters can also slow down sampling and planning.

One way to decrease the parameters is through modularization. That is, if we can find that some sets of variables are conditionally independent of one another, we can separate them into different *modules*. This takes the number of parameters from a polynomial of the total number of variables down to a sum of polynomials of variables in each module. Modularization has biological plausibility and has been used successfully with other factored-state reinforcement learning methods [Rothkopf and Ballard, 2010] and particularly with neural networks [Reisinger *et al.*, 2004]. One example of a concrete method for dividing the state space to help learning scale is by providing an object-oriented representation [Diuk *et al.*, 2008].

A benefit of using neural networks to represent knowledge is that they have an automatic method for encouraging modularization: *weight pruning*. Pruning is the permanent detachment of a link between two nodes, generally based on the sensitivity of the network to changes in that connection's weight [Reed, 1993; LeCun *et al.*, 1990]. Features that are conditionally independent of one another naturally tend toward smaller absolute weight connections during the learning process. In sparse or object-oriented environments the neural networks behind our model learner can shrink substantially in complexity, as we were able to prune most of the weights in our networks without hurting performance in our mini-game experiments.

5.1 Experiments: Modularization

Our third mini-game, *Flier Catcher*, involves several independent objects and requires some forward planning to achieve high rewards. The agent controls the circle on the left-most column of a grid and can only move up or down. The rectangular fliers continue to spawn randomly on the right-most column and fly straight to the left at fixed speed. The agent collects rewards by touching the fliers with the circle.

After training the agent to perform well on this game, we used a simple pruning method of removing the smallest weights by absolute value from the predictor and conditioner neural networks. Just under 90% of the connections were pruned and the now sparser networks were retrained without significantly reducing the rewards subsequently collected.

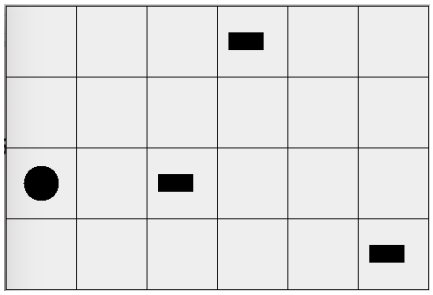


Figure 10: Flier Catcher

In Figure 11, darker boxes represent the absolute strength of all weights in paths from inputs (x-axis) to outputs (y-axis) before pruning. Figure 12 shows the same but after pruning. We can note that despite losing the vast majority of its neural connections, the agent retains knowledge of the important object relationships in the game: the valid player positions, the effect of actions on player position, and the leftward movement of the fliers.

6 Discussion

We have outlined how a model learning system based on neural networks can learn representations for approximately sampling from state transition probability functions. With the ability to sample from an environment's transitions, an agent can perform the inferences and predictions necessary for planning. And the ability to learn from observation the necessary parameters for sampling is useful in problems where the agent does not know the true model of its environment, such as games where the player is not given all the exact rules.

The approach we use separates the problem of model learning from the problems of planning and goal-setting. The separation of cognitive modules each learning its own set of parameters can lead to reduced complexity and provide reusable parts. We have also seen how the implementation of the model learner in a neural network architecture can further facilitate modularization and object-orientation of knowledge via weight pruning.

We hope the ideas put forward here will lead to further development of model learning methods. So far there seems to be limited exploration of neural network techniques specialized for model learning, though the approach seems promising. This paper is only an initial examination of one idea, and more testing is necessary to further illuminate its strengths and weaknesses. We also believe the following specific paths from here may be especially fruitful:

- Applying other generative neural methods such as RBM belief networks and NADE-based techniques
- Developing ways to efficiently estimate model error
- Testing on a wider variety of stochastic games
- Testing on environments with continuous state variables
- Investigating bottom-up approaches to achieving modularization and parameter sparsity

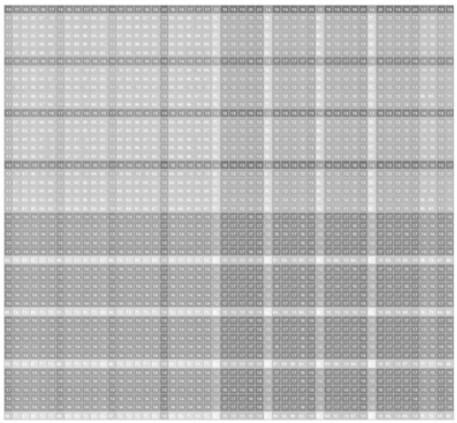


Figure 11: Network input-to-output sum of absolute weights before weight pruning

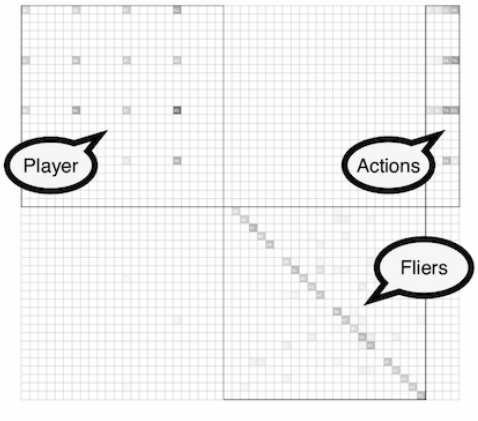


Figure 12: Network input-to-output sum of absolute weights after weight pruning

References

[Barto, 1998] Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[Bengio and Bengio, 1999] Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multilayer neural networks. In *NIPS*, volume 99, pages 400–406, 1999.

[Browne *et al.*, 2012] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[Cheng *et al.*, 2006] Haibin Cheng, Pang-Ning Tan, Jing Gao, and Jerry Scripps. Multistep-ahead time series prediction. In *Advances in Knowledge Discovery and Data Mining*, pages 765–774. Springer, 2006.

[Diuk *et al.*, 2008] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247. ACM, 2008.

[Garcez *et al.*, 2008] Artur SD'Avila Garcez, Luis C Lamb, and Dov M Gabbay. *Neural-symbolic cognitive reasoning*. Springer Science & Business Media, 2008.

[Gläscher *et al.*, 2010] Jan Gläscher, Nathaniel Daw, Peter Dayan, and John P O'Doherty. States versus rewards: dissociable neural prediction error signals underlying model-based and model-free reinforcement learning. *Neuron*, 66(4):585–595, 2010.

[Gregor *et al.*, 2013] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep autoregressive networks. *arXiv preprint arXiv:1310.8499*, 2013.

[Hinton and Sejnowski, 1986] Geoffrey E Hinton and Terrance J Sejnowski. Learning and relearning in boltzmann machines. *MIT Press, Cambridge, Mass*, 1:282–317, 1986.

[Hornik *et al.*, 1989] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[Justel and Peña, 1996] Ana Justel and Daniel Peña. Gibbs sampling will fail in outlier problems with strong masking. 5(2):176–189, June 1996.

[Kearns *et al.*, 2002] Michael Kearns, Yishay Mansour, and Andrew Y Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

[Larochelle and Murray, 2011] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. *Journal of Machine Learning Research*, 15:29–37, 2011.

[LeCun *et al.*, 1990] Yann LeCun, J. S. Denker, S. Solla, R. E. Howard, and L. D. Jackel. Optimal brain damage. In David Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS 1989)*, volume 2, Denver, CO, 1990. Morgan Kaufman.

[Moriguchi and Lipson, 2011] Hirotaka Moriguchi and Hod Lipson. Learning symbolic forward models for robotic motion planning and control. In *Proceedings of European conference of artificial life (ECAL 2011)*, pages 558–564, 2011.

[Murphy, 2002] Kevin Patrick Murphy. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, Berkeley, 2002.

[Powell, 2007] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[Reed, 1993] Russell Reed. Pruning algorithmsa survey. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 4(5), 1993.

[Reisinger *et al.*, 2004] Joseph Reisinger, Kenneth O Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 69–81. Springer, 2004.

[Richard and Lippmann, 1991] Michael D Richard and Richard P Lippmann. Neural network classifiers estimate bayesian a posteriori probabilities. *Neural computation*, 3(4):461–483, 1991.

[Rothkopf and Ballard, 2010] Constantin A Rothkopf and Dana H Ballard. Credit assignment in multiple goal embodied visuomotor behavior. *Embodied and grounded cognition*, page 217, 2010.

[Rumelhart *et al.*, 1988] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[Singh *et al.*, 2010] Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *Autonomous Mental Development, IEEE Transactions on*, 2(2):70–82, 2010.

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

[Sutskever and Hinton, 2007] Ilya Sutskever and Geoffrey E Hinton. Learning multilevel distributed representations for high-dimensional sequences. In *International Conference on Artificial Intelligence and Statistics*, pages 548–555, 2007.