Eager Beaver
A General Game Player

André Doser, Florian Geißer, Philipp Lerche, and Tim Schulte

Foundations of Artificial Intelligence
University of Freiburg
{dosera,geisserf,lerchep,schultet}@tf.uni-freiburg.de

Abstract. General game playing is the research field of being able to play multiple different kinds of games with one AI. We present Eager Beaver, a general game player based on Propositional Networks with dynamic code generation and an enhanced Upper Confidence Bounds applied to Trees (UCT) algorithm as an approach to solve this problem. We ran an evaluation study against Centurio [7], another UCT player and show the results of various UCT extensions used during this competition.

1 Introduction

Since the beginning of research in Artificial Intelligence, solving games has been an important area of interest. One complex, well-known game is chess, which consists of around 10^{47} different possible states. In 1997 the chess computer Deep Blue was the first AI to defeat the reigning chess world champion Kasparov. However, while Deep Blue is nearly a perfect chess player it is not able to play a simple game of Tic-tac-toe. Playing multiple different kinds of games with one AI is the focus in the research field of general game playing (GGP). To provide a uniform set of game rules, Love et al. [6] introduced the so called Game Description Language (GDL). With these rules a player is able to play any finite, discrete and deterministic[1] multi-player game of complete information without any game-specific algorithm. After receiving the game rules, each player has a limited time for preparation once a game and a limited time to choose its move each round. Since 2005, a yearly general game playing competition [4] takes place, to promote the research area of general game playing and contribute new ideas. In the first years, the traditional approach was to use a minimax-based game tree search combined with automatically learned heuristic evaluation functions. Nowadays, most of the players use the UCT algorithm [5]. The efficiency of the algorithms depends on the ability to quickly provide basic state manipulation operations like computation of the initial state, legal moves and next states. In contrast to the widely used Prolog-based reasoner we use a different, performance-capable approach: Propositional Networks [2] with dynamic code generation.

[1] In 2010 Thielscher [9] proposed GDL-II which allows game rules with incomplete and imperfect information

S. Wölfl (Ed.): Poster and Demo Track of the 35th German Conference on Artificial Intelligence (KI-2012), pp. 54-58, 2012.

This paper is organized as follows: in the next section, we introduce the general mechanics of a general game player. Afterwards, we describe the theoretical background of the main data structures and algorithms that are used by Eager Beaver. We conclude with implementation details and some benchmarks.

2 Data Structures and Algorithms

The game rules, available in GDL syntax, are parsed and forwarded to the Propositional Network or any other reasoner, that provides state manipulation operations. The state machine is an interface that allows the algorithm to receive the same responses for its queries, e.g. legal moves, independently of the used reasoner. See Figure 1 for an overview of Eager Beaver's layout.

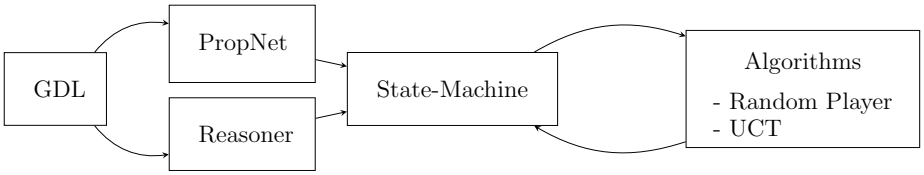

Fig. 1. Layout of Eager Beaver

2.1 Propositional Networks

In order to compute the best possible move to play in a given state, every algorithm has to rely on an efficient way to calculate upcoming legal moves for each player, their goal values (points) and the consequent states. The algorithms walk through hundreds of thousands of different game situations, therefore it is important that these basic operations are done as fast as possible. Propositional Networks are developed by the Stanford Logic Group and are an intuitive representation of GDL rules as a bipartite graph. As a consequence possible graph structures can be exploited.

A Propositional Network (PN) consists of propositions, which represent a part of the game state and can be either true or false. These propositions are connected to either boolean gates, logically connecting multiple propositions or to transitions, allowing the transition from one state into another. There are three different types of propositions[2]: *Base propositions* describe the state of a game, *input propositions* correspond to game moves and *view propositions*, determined by the GDL rules, which can be used to represent legal moves or goal values.

Figure 2 shows the PN of the Two Button game. In this single-player game the player is able to press two different buttons A and B. If one button is pressed it remains pressed for the rest of the game. The goal is to press both buttons. The two base propositions *(pressed A)* and *(pressed B)* describe the whole state of this

[2] In this year's upcoming competition GDL rules are enhanced by base and input relations, thus making it easier to compute the different propositions.

game: either no button is pressed, A is pressed, B is pressed or both buttons are pressed. If the player presses button A the value of the input proposition *(does player (press A))* becomes true. Its value is propagated through the network, so the *OR*, *VIEW* and *TRANSITION* node values become true. Remember that transitions allow the transition from one state into another. Since the input of the base proposition *(pressed A)* is true our new state consists of *(pressed A)*. Note that the view propositions *VIEW* are just for a correct PN representation, because boolean gates have to be connected to propositions and the input of transitions has to be a proposition.

To provide an efficient data structure for the aforementioned computations, Eager Beaver uses PNs extended by dynamic code generation, thus gaining a remarkable performance increase compared to the standard reasoner. For a PN we generate code in a way that redundant logic evaluations are omitted. For each proposition a method is generated that propagates its truth value by calling other methods. Furthermore boolean gates are represented as a byte array, thus we can efficiently save and load their values. By calling only methods of relevant propositions we avoid multiple computations of the same truth values. When we want to perform a basic operation, such as calculating the current goal values, we call the corresponding method in the generated class. This method runs very fast, since all it does is comparing and setting boolean values. We want to mention here that games with tens of thousands of propositions can be a problem for generating code with Java as underlying language, because Java has an upper bound of 65536 methods and 65536 characters per class. One solution is to split the proposition updates into multiple classes, like one class for all base proposition updates, one for all view proposition updates and so on.

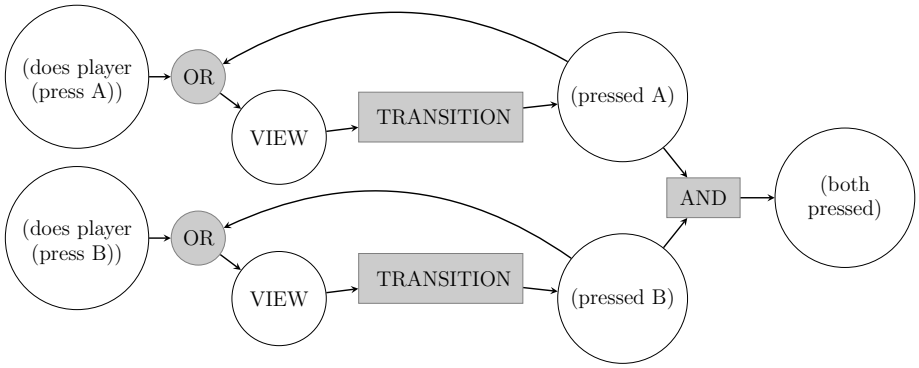

Fig. 2. A PN representation of the Two Button game [2]

2.2 UCT Algorithm

In the last years, the Upper Confidence Bounds applied to Trees (UCT) algorithm has prevailed for most successful players. The idea for this technique arose

from the UCB1 algorithm proposed by Auer et al. [1] and is, in contrast to former concepts, based on a Monte-Carlo approach. After receiving a request to select a move a^* in a given game state, the algorithm gradually builds up a tree whose spread is biased by the UCB1 formula

$$a^* = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \mathcal{Q}(s,a) + C\sqrt{\frac{ln\, N(s)}{N(s,a)}} \right\},$$

where $\mathcal{Q}(s,a)$ denotes the average score for action a in state s, $N(s)$ the accumulated visits of the parent state s, $N(s,a)$ the accumulated visits of the successor s' of s through a and C the constant for the weighting the UCT-bonus. Since it is generally impossible to represent the whole state space of a game, the UCT algorithm resorts on random playouts, namely Monte-Carlo simulations.

The algorithm consists of four fundamental phases which are consecutively executed until the timeout is reached. In the *Selection-Phase* the tree is recursively traversed by using the UCB1 formula, which tries to find a balance between exploiting the most promising moves and exploring less encouraging moves. After finding a game state which hasn't been inspected before, the *Expansion-Phase* starts and its successor states are computed and added to the game tree. In the *Simulation-Phase*, one successor is selected and random simulations are run, thus obtaining a terminal state and receiving scores for each player. These scores are propagated through the tree in the *Backpropagation-Phase*. After the timeout is reached the most promising move, i.e. the move with the most visits, is selected from the game tree.

Besides the plain UCT algorithm, Eager Beaver uses two major extensions: MAST and RAVE [3]. Furthermore the implementation allows to steer the move selection by adaptation of several constants, e.g. updating the UCB1 constant every rollout depending on the current node's wins and visits.

3 Results

Eager Beaver is based on the GGP-Base framework by Sam Schreiber [8], which is written in Java 1.6 and provides several basic features, such as server communication via HTTP. Compared to the framework's default reasoner, the implementation of the Propositional Networks led to a significant increase in runtime performance. For example, in case of *connect-4*, state machine queries are computed around 25 times faster with Propositional Networks. For our benchmarks we competed against the general game player *Centurio* V2.1 on an Athlon SUN Fire X2200 M2 x64 with 2,3GHz per core (8 cores available in total) and 32GB RAM (although we used only 64MB). Each player received one core. We let Eager Beaver play with three different configurations (plain UCT, UCT with MAST and UCT with RAVE) against Centurio using default settings with 15 seconds lap time and 45 seconds preparation time. It seems that for the three chosen games, the increased computational costs of the UCT extensions didn't pay off compared to the higher number of plain UCT simulations.

For the games we chose Tic-tac-toe, Connect 4 and Breakthrough Cylinder. Each game and configuration was played 1400 times. See results in Figure 3.

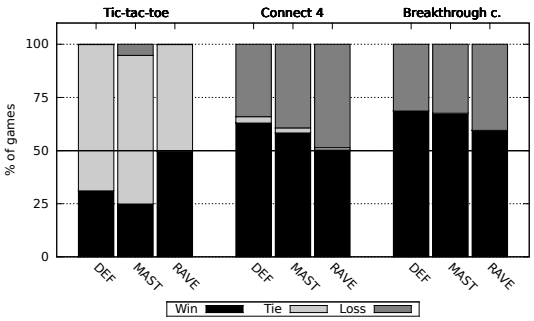

Fig. 3. Three different games vs Centurio

4 Conclusion

According to the present literature our implementation of Propositional Networks with dynamic code generation seems to be unique so far. The results emphasize that this led to a great performance increase. Together with our UCT implementation Eager Beaver was able to defeat Centurio in the majority of the matches. The default UCT configuration was the most successful setting with which Eager Beaver seems to outperform Centurio. Although the field of general game playing is young, there are more improvements, which we could include in our player, like Propositional Network Factoring [2] or other UCT enhancements.

We will participate in the next general game playing competition[3], starting July 22 this year. Moreover the source code of Eager Beaver is available at `https://bitbucket.org/tasse/eager-beaver`.

References

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.
[2] Evan Cox, Eric Schkufza, Ryan Madsen, and Michael R. Genesereth. Factoring general games using propositional automata. *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 2009.
[3] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, 25(1):9–16, 2011.
[4] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
[5] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
[6] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group, March 2008.
[7] Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, Java- and ASP-based. *KI*, 25(1):17–24, 2011.
[8] Sam Schreiber. The general game playing base package, 2010.
[9] Michael Thielscher. GDL-II. *KI*, 25(1):63–66, 2011.

[3] For more information see `http://games.stanford.edu`.