Knowledge Generation for Improving Simulations in UCT for General Game Playing

Shiven Sharma, Ziad Kobti, and Scott Goodwin

Department of Computer Science, University of Windsor,
401 Sunset Avenue, Windsor, ON N9C4B9, Canada
{sharmaw,kobti,sgoodwin}@uwindsor.ca
http://www.cs.uwindsor.ca

Abstract. General Game Playing (GGP) aims at developing game playing agents that are able to play a variety of games and, in the absence of pre-programmed game specific knowledge, become proficient players. Most GGP players have used standard tree-search techniques enhanced by automatic heuristic learning. The UCT algorithm, a simulation-based tree search, is a new approach and has been used successfully in GGP. However, it relies heavily on random simulations to assign values to unvisited nodes and selecting nodes for descending down a tree. This can lead to slower convergence times in UCT. In this paper, we discuss the generation and evolution of domain-independent knowledge using both state and move patterns. This is then used to guide the simulations in UCT. In order to test the improvements, we create matches between a player using standard the UCT algorithm and one using UCT enhanced with knowledge.

Key words: General Game Playing, Monte Carlo Methods, Reinforcement Learning, UCT

1 Introduction

Historically, game playing agents were designed to be good in specific games. However, even though these players excelled in games that they were designed for, they could not play any other games. General Game Playing (GGP) focuses on the creation of agents that are able to accept rules of a game, and use them to learn how to play it, eventually displaying a high level of competence in it.

A class of games for which a GGP approach was taken were *positional games* (eg. Tic-Tac-Toe, Hex and the Shannon switching games), which were formalised by [1]. A positional game can be defined by three sets, P, A, B. Set P is a set of positions; with set A and B are sets of subsets of P. Programs that are capable of accepting rules of positional games and, with practice, learn how to play the game have been developed. [1] constructed a program that is able to learn important board configurations in a 4 x 4 x 4 Tic-Tac-Toe game.

The annual General Game Playing Competition [2] organised by Stanford University has been instrumental in bringing about renewed interest in GGP.

The rules of the games are written in Game Description Language (GDL) [3]. The tournaments are controlled by the Game Manager (GM) which relays the game information to each Game Player (GP) and checks for legality of moves and termination of the game [4]. Communication between players and the GM takes place in the form of HTTP messages. Successful players have mainly focused on automatically generating heuristics based on certain generic features identified in the game. Cluneplayer [5] was the winner of the first GGP competition, followed by Fluxplayer [6]. Both these players, along with UTexas Larg [7] use automatic feature extraction. Another approach that has been taken is in [8], where transfer of knowledge extracted from one game to another is explored by means of a TD(λ) based reinforcement learner.

The main aim of this paper is to explore the creation and use of domain-independent knowledge. The knowledge is then applied to UCT search for GGP. The rest of the paper is organised as follows. In the Section 2 the UCT search mechanism if discussed briefly along with its application to GGP. In Section 3 we discuss the knowledge representation scheme. Matches are then played between a standard UCT player and a player using the knowledge and stored tree for the games large (5 x 5) Tic-Tac-Toe, Connect-4, Breakthrough and Checkers.

2 UCT and General Game Playing

UCT [9] is an extension of the UCB1 algorithm [10], and stands for *Upper Confidence Bounds applied to Trees*. The UCB1 Algorithm aims at obtaining a fair balance between exploration and exploitation in a K-Armed bandit problem, in which the player is given the option of selecting one of K arms of a slot machine (i.e. the bandit). The selection of arms is directly proportional to the total number of plays and inversely proportional to the number of plays of each arm. This is seen in the selection formula

$$\bar{X}_j + C\sqrt{\frac{2\log n}{T_j(n)}} \tag{1}$$

The arm maximising (1) is selected. $\bar{X}_j$ is the average return of arm j after n plays, and $T_j(n)$ is the number of times it has been selected. C controls the balance between exploration and exploitation of the tree. This formula is used in our player.

UCT extends UCB1 to trees. A single simulation consists of selecting nodes to descend down the tree using and using (1) and random simulations to assign values to nodes that are being seen for the first time. CADIA-Player [11] was the first General Game Player to use a simulation based approach, using UCT to search for solutions, and was the winner of the last GGP Competition. UCT has also been used in the game of Go, and the current computer world champion, MoGo [12], uses UCT along with prior game knowledge. [13] also used simulations to build a basic knowledge base of move sequence patterns in a multi-agent General Game Player. Selection of moves was done based on the average returns, and

mutation between move sequence patterns was done to facilitate exploration. An advantage of a simulation based approach to General Game Playing is that for any game, generating a number of random game sequences does not consume a lot of time, since no effort is made in selecting a good move. UCT is able to guide these random playoffs and start delivering near-optimal moves. However, even with UCT, lack of game knowledge can be a significant obstacle in more complex games. This is because in the absence of any knowledge to guide the playoffs, it takes a large number of runs of the UCT algorithm to converge upon reasonably good moves.

2.1 Structure of the Game Player

In order to make a move, the current state is set as the root node of a tree which is used by UCT. To allow for adequate exploration, the algorithm ensures that each child is visited at least once. To manage memory, every node that is visited at least once is added to a *visited table*. Once a node is reached which is not in the table, a random simulation is carried on from that node till the end of the game. The reward received is backed up from that node to the root. Algorithm 1 shows the basic method of UCT and the update of values in a 2 player zero-sum game. A number of such simulations are carried out, each starting from the root and building the tree asymmetrically.

Algorithm 1 UCTSearch(*root*)

```
1: node = root
2: while visitedTable.contains(node) do
3:     if node is leaf then
4:         return value of node
5:     else
6:         if node.children.size == 0 then
7:             node.createChildren()
8:         end if
9:         selectedChild = UCTSelect(node)
10:        node = selectedChild
11:    end if
12: end while
13: visitedTable.add(node)
14: outcome = RandomSimulation(node)
15: while node.parent ≠ null do
16:    node.visits = node.visits + 1
17:    node.wins = node.wins + outcome
18:    outcome = 1 − outcome
19:    node = node.parent
20: end while
```

The value of each node is expressed as the ratio of the number of winning sequences through it to the total number of times it has been selected. In order to select the final move to be made, the algorithm returns the child of the root having the best value.

In the next section we discuss how domain-independent knowledge can be gathered from simulations to guide them towards reasonable solutions.

3 Knowledge Creation and Evolution

Knowledge can be used to guide node selection and random simulations in UCT. [12] uses domain dependent knowledge in the form of local patterns to make the simulations more realistic. [11] uses a history heuristic for moves to provide basic knowledge, associating with each move its average win rate, irrespective of the state it was made in. [14] use TD(0) learning to learn an evaluation function for a template of local features in Go. The value function is expressed as a linear combination of all local shape features, squashed by the *sigmoid* function to constrain it to be between $[0, 1]$, thus giving it a natural interpretation as a probability of winning. Our approach to learning state-space knowledge is very similar to it, the major difference being that instead of focusing on local features, we use the entire state description.

3.1 State-Space Knowledge

States in GDL are represented as a set of ordered tuples, each of which specifies a certain feature of the state. For example, in Tic-Tac-Toe, $cell(1, 1, b)$ specifies that the cell in row 1 and column 1 is blank. Therefore, a state in Tic-tac-Toe is represented as a set of 9 such tuples, each specifying whether a cell is blank or contains an X or an O. In order to extract state-space knowledge from such tuples, we assign values to each of these tuples. Initially, the values are set to 0, thereby giving the squashed state value as 0.5 (equal probability of winning and losing). The value of a state S is calculated as

$$V(S) = \sigma \left(\sum_{t \in match(S)} \psi_t \right) \tag{2}$$

where the sigmoid function $\sigma(t)$, a special form of the logistic function, squashes the value between 0 and 1. ψ_t is the value of tuple t that is part of the description of state S.

3.2 Move Knowledge

The manner in which move knowledge is expressed is similar to that used by [11]. However, instead of simply giving a move its average win rate as a value, we factor in the depth as well, as given in [15]. Assuming a move m is made at

position p during a random simulation, and the length of the simulation is l and the outcome is r, then the value assigned to m after n plays is given as

$$V(m) = \frac{\sum_{i=1}^{n} 2^{\frac{l_i - p_i}{l_i}} \times r_i}{n} \tag{3}$$

3.3 Learning

Every time a random game is played, knowledge is updated for the role(s) that used the knowledge (examples of roles are *black* and *white* in Chess). The result of the random game is used to perform the updates. If the outcome of game n is r, the values for each move played in the sequence, $V(m)$, is updated as

$$V(m) = \frac{R_{n-1} + 2^{\frac{l-p}{l}} \times r}{n} \tag{4}$$

where R_{n-1} is the cumulative reward for $n-1$ plays (the numerator of (3)).

For each tuples t in each state of the sequence, its value ψ_t is updated as

$$\psi_t = \psi_t + \alpha \left(\frac{r - V(S)}{size_S} \right) \tag{5}$$

The learning rate, α is gradually decreased over time. $size_S$ is the number of tuples in the description of S.

3.4 Using the knowledge

Given the two forms of knowledge, $\Psi_m = V(m) + V(S_m)$ gives a measure of the effectiveness of a move, where m is the move which leads to state S_m. Random simulations use this knowledge by using $\epsilon - greedy$ selection, with a relatively high value of ϵ (this is done to prevent the simulations from becoming too deterministic).[1]

In order to select nodes that have not been selected previously (not in the *visitedTable* of Algorithm 1), UCT selects all nodes randomly, with equal probability. However, using the knowledge learnt it is possible to bias this selection in favour of nodes with higher values. Therefore, we assign these nodes an initial value based on $V(S_m)$ and $V(m)$.. This is similar to the approach taken in [16]. Node selection is then done using the standard UCT formula of (1), with $\bar{X}_j$ being replaced by Ψ_m. $N.value$ and $N.visits$ are updated as given in (1).

4 Experiments

In order to test the effectiveness of using knowledge with UCT, we played matches between a player using standard UCT, UCT_S, and a player using knowledge for simulations, UCT_K. We implemented a GGP type system on our local

[1] [16] presents an interesting discussion on the effect of different types of simulations in UCT search.

machines. The players and the manager were written in Java. For inferencing, the game rules were converted to Prolog. YProlog [17], a Prolog-style inference engine written in Java, was used. The games used for testing were 5 X 5 Tic-Tac-Toe, Connect-4 and Breakthrough, Checkers. The results for 100 games each are shown. The knowledge, once generated, was not changed. The roles for the games were randomly distributed between the UCT_S and UCT_K. The addition

Table 1. Number of wins for UCT_S and UCT_K over 100 matches per game

Game	Wins for UCT_S	Wins for UCT_K	Total Draws
Tic-Tac-Toe (large)	7	15	78
Connect-4	41	59	0
Breakthrough	32	68	0
Checkers	43	57	0

of knowledge during random simulations results in corresponding player winning the majority of matches. We will investigate the effect continuous evolution of knowledge in future experiments. We also hope to reconstruct some of the existing game players (to the best of our knowledge) and test knowledge enhanced versions of the UCT player against them, as this will give a broader perspective as to the strength of the UCT player.

5 Conclusions and Future Work

In this paper we proposed an approach for generating and evolving domain-independent knowledge for General Game Players. The idea of the history heuristic was taken from [15], and it has been used in [11]. The state knowledge representation and learning was inspired from [14]. The combined knowledge is used to guide simulations in UCT. The results of matches between a standard UCT player and a player enhanced with knowledge clearly indicate the advantages of including knowledge. Learning can be speeded up by using faster inference engines (for example, YAP [18] and B-Prolog [19]), using hashing techniques such as [20] to look up tuples, and by parallelising the simulations.

The very nature of General Game Playing makes it difficult to create and use knowledge. Given the vast variety of games that can be played, building a general framework for knowledge representation and learning is challenging. In many games, moves that are successful in certain states may not be successful in other states. Therefore, the history value of moves only acts as a simple guide for making future moves. In our representation for state knowledge, we have assumed that the tuples are independent of each other. However, in most cases it is the relationship between various state features that matters. An alternative to using linear function approximation is using non-linear function approximation techniques such as neural-networks. We are also investigating the use of Ant Colony Optimisation approaches [21] to generate random sequences and use them in conjunction with the ideas presented in this paper.

Acknowledgments. This work was funded in part by an NSERC Discovery grant.

References

1. Koffman, E.: Learning through pattern recognition applied to a class of games. In: IEEE Trans. on Systems, Man and Cybernetics, Vol SSC-4. (1968)
2. Genesereth, M., Love, N.: General game playing: Overview of the aaai competition. AI Magazine (Spring 2005) (2005)
3. Genesereth, M., Love, N.: General game playing: Game description language specification. `http://games.standford.edu`
4. `http://games.stanford.edu`
5. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence. (2007)
6. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence. (2007) 1191–1196
7. Banerjee, B., Kuhlmann, G., Stone, P.: Value function transfer for general game playing. In: ICML Workshop on Structural Knowledge Transfer for ML. (2006)
8. Banerjee, B., Stone, P.: General game playing using knowledge transfer. In: The 20th International Joint Conference on Artificial Intelligence. (2007) 672–777
9. Kocsis, L., Szepesvari, C.: Bandit based monte-carlo planning. In: European Conference on Machine Learning (ECML). (2006) 282–293
10. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite time analysis of the multi-armed bandit problem. Machine Learning, vol. 47, no. 2/3 (2002) 235–256
11. Bjornsoon, Y., H., F.: Simulation-based approach to general game playing. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI Press (2008)
12. Gelly, S., Wang, Y.: Modifications of uct and sequence-like simulations for monte-carlo go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii. (2007)
13. Sharma, S., Kobti, Z.: A multi-agent architecture for general game playing. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii. (2007)
14. Silver, D., Sutton, R., Muller, M.: Reinforcement learning of local shape in the game of go. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07). (2007)
15. Schaeffer, J.: The history heuristic and alpha-beta search enhancements in practice. IEEE Transaction on Pattern Analysis and Machine Intelligence (1989) 1203–1212
16. Gelly, S.: A Contribution to Reinforcement Learning; Application to Computer-Go. PhD thesis, University of Paris South (2007)
17. Winikoff, M. `http://www3.cs.utwente.nl/~schooten/yprolog`
18. `http://www.dcc.fc.up.pt/~vsc/Yap/`
19. `http://www.probp.com/`
20. Zobrist, A.: A new hashing method with application for game playing. Technical report 99, University of Wisconsin (1970)
21. Sharma, S., Kobti, Z., Goodwin, S.: General game playing with ants. In: The Seventh International Conference on Simulated Evolution And Learning (SEAL'08), Melbourne, Australia. (2008) (in press).