Multigame playing by means of UCT enhanced with automatically generated evaluation functions

Karol Walędzik and Jacek Mańdziuk

Faculty of Mathematic and Information Science,
Warsaw University of Technology,
Pl. Politechniki 1, 00-661 Warsaw, Poland,
{k.waledzik,j.mandziuk}@mini.pw.edu.pl

Abstract. General Game Playing (GGP) contest provides a research framework suitable for developing and testing AGI approaches in game domain. In this paper, we propose a new modification of UCT game-tree analysis algorithm working in cooperation with a knowledge-free method of building approximate evaluation functions for GGP games. The process of function development consists of two stages: *generalization* and *specification*. Both stages are performed autonomously by the system and do not require human intervention of any kind. Effectiveness of the proposed method is proved in three exemplar games selected from the GGP games repository.

1 Introduction

Playing games has always been an important part of human activities and the oldest board games still played in their original form (Go and backgammon) date back to 1,000 - 2,000 BC. Games also became a fascinating topic for Artificial Intelligence (AI) and Computational Intelligence (CI) research.

Years of research aimed at particular games led to significant achievements of artificial systems including theoretical solving of the game of checkers [15] or outplaying human top chess players with a wide margin. Majority of spectacular accomplishments of AI in games, however, adopted the *narrow-AI* approach, i.e. the winning programs were tailored for particular games and were rather examples of wonderful engineering abilities of their inventors than the hallmarks of *real* intelligence. In principle, these approaches lacked *universal learning mechanisms* (most of the top playing programs in classical board games do not apply any learning whatsoever) and *generality of approach* also known as *multigame playing ability* (none of the top programs in any board game can play any other, even relatively similar, game). These two capabilities are, on the other hand, evident hallmarks of human intelligence [12].

In this paper we adopt autonomous learning approach to building evaluation function for the General Game Playing (GGP) contest [6] (described in the following section). Our approach interweaves generalization mechanism, which

allows building a large pool of candidate features, with specification stage (which, in turn, selects a reasonable subset of pertinent features). Both stages are performed without human intervention as they are based on generally applicable heuristical meta-rules. What is more, contrary to approach of many GGP competitors [16, 9], our method operates strictly on game descriptions only without any implicit expectations about their structure or rules, i.e. it does not expect them to be played on boards, to use pieces, for players to make moves alternately etc.

The evaluation function devised based on the above described subset of features is subsequently employed by what we call a Guided UCT algorithm - our modification of the state-of-the-art UCT tree search method described in section 3. Results of simulations performed in three game domains: chess, checkers and connect-4 proved a clear upper hand of the enhanced UCT method over its plain version, even in the case of not restrictive time regime.

2 General Game Playing Competition

GGP, one of the latest and most popular approaches to the multigame playing topic, was proposed at Stanford University in 2005 in the form of General Game Playing Competition [8]. General Game Playing applications are able to interpret game rules encoded in Game Description Language (GDL) [11] statements and devise a strategy allowing them to play those games effectively without human intervention. The competition includes a wide variety of games, both known previously and devised specifically for the tournament. Contestants are expected to deal with games of varied complexity and differing numbers of players. They can include single-player puzzles, as well as both cooperative and competitive multi-player games.

2.1 Game Description Language

Game Description Language (GDL) [11] is a variant of Datalog used to describe the rules of the class of games playable within the GGP framework, i.e. finite, discrete, deterministic multi-player games of complete information. Game states are represented by sets of facts while algorithms for computing legal moves, subsequent game states, termination conditions and final scores for players are defined by logical rules. In order to make the description understandable to playing agents several distinguished keywords are required:

role(p) defining p as a player;
init(f) stating that f is true in the initial state of the game;
true(f) stating that f holds in the current game state;
does(p, m) stating that player p performs move m;
next(f) stating that f will be true in the next state (reached after all actions defined in *does* relations are performed);
legal(p, m) stating that it is legal in current state for player p to perform move m;

```
;Roles:
(role x) (role y)
;Initial state:
(init (cell 1 1 b)) (init (cell 1 2 b)) ... (init (control x))
;Rules:
(<= (next (cell ?x ?y ?player)) does (?player (mark ?x ?y)))
(<= (next (control x)) (true (control o)))
...
(<= (line ?player) (row ?x ?player))
;Legal moves:
(<= (legal ?player (mark ?x ?y)) (true (cell ?x ?y b)) (true (control ?player))) ...
;Goals:
(<= (goal ?player 100) (line ?player)) ...
;Terminal:
(<= terminal (line ?player)) ...
```

Fig. 1. A condensed Tic-Tac-Toe game description in GDL.

goal(p**,** v**)** stating that, should current state be terminal, the score of player p would be v;

terminal stating that current state is terminal.

Figure 1 presents an extract of sample description of Tic-Tac-Toe game in prefix KIF presented in [11]. As can be seen, typical game descriptions make extensive use of variables (prefixed with '?') and game-specific predicates. This form of GDL can naturally and easily be translated into Prolog (or any other logic programming language). This path is, in fact, often chosen by general game players' implementors for game engine implementation.

3 UCT

Upper Confidence bounds for Trees (UCT) is a simulation-based game playing algorithm that proved to be quite successful in case of some difficult game-based tasks, including Go [4] and GGP tournament, where it was employed by CadiaPlayer [3] - competition champion two times in a row in 2007 and 2008. UCT is a variation of a UCB1 [1] approach to solving K-armed bandit problem. K-armed bandit problem aim is finding an optimal way of interacting with K traditional gambling machines (or, alternatively, one machine with K modes of play - arms). The rewards of the machines are nondeterministic but stable (the rewards distributions remain constant in time) and pairwise independent.

UCT algorithm treats each game position as a bandit and each possible action in that position as an arm. Following the principles of UCB1, UCT advises to first try each action once and then, whenever the same position in encountered, choose action according to the following formula:

$$a* = argmax_{a \in A(s)}\{Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}}\}, \tag{1}$$

where $A(s)$ denotes the set of all actions possible in state s, $Q(s,a)$ – average return of the state-action pair so far, $N(s)$ – number of times state s has been visited by the algorithm and $N(s,a)$ - number of times action a has been selected in state s. Therefore, probability of choosing each action depends on its observed rewards and the frequency with which it has been chosen so far (with a tendency towards choosing less popular actions). This way UCT provides a balance between exploiting the actions that proved to be most rewarding and exploiting other possibilities.

In realistic cases it is of course impossible to store information about all game states in memory at the same time (games for which this is possible can easily be solved by brute-force algorithms). Actual implementation of UCT algorithm's simulation consists therefore of 2 phases (see figure 2). Game tree is built in memory iteratively. In each simulated episode, strict UCT strategy is applied only until first so-far-unvisited node is encountered. Afterwards, the rest of the simulated episode consists in plain Monte Carlo simulation, i.e. random play with no data retained in memory. Since better moves are selected more often than less promising ones, the in-memory tree is actually expanded according to a kind of best-first strategy. Additionally, each time a non-simulated move is played all information about game-states prior to it is discarded from memory.

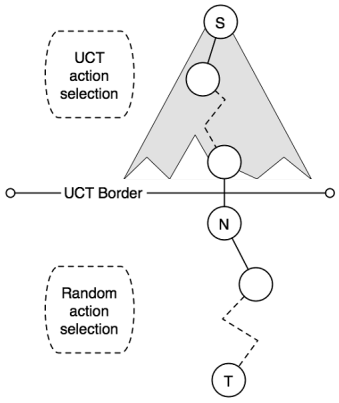

Fig. 2. Conceptual overview of a single simulation in a UCT algorithm [3]

4 Guided UCT

UCT algorithm in its basic form requires no expert game-specific knowledge, which is its huge advantage and makes it easier to apply it in many problems. We, however, investigate the possibility of augmenting UCT with **automatically inferred** game-specific state evaluation function. Approaches similar in idea, but very different in realization, have been employed by several programs making use of UCT method, e.g. aforementioned CadiaPlayer [3] and Go-playing program MoGo [5]. For the sake of clarity, we will refer to any version of our augmented UCT algorithm as Guided UCT (GUCT).

The evaluation function we aim to include in our Guided UCT algorithm is expected to return, upon being presented a valid game state, reasonable approximation of the expected score of the player it is designed for. While it is not expected to be as precise as game tree search algorithms, it must be orders of magnitude faster than any such method.

Once defined, the evaluation function $F(s)$ can be employed by the GUCT method in several ways, both in strict UCT and Monte-Carlo simulation phases. First of all, it can influence the value of state-action pair evaluation function $Q(s, a)$. The value $Q(s, a)$ can be a simple weighted average of the heuristical value provided by the evaluation function and the current simulation results, with simulations' results weight increasing with the value of $N(s, a)$. Alternatively, F can be used to pre-initialize the data stored in the game tree built by the UCT. Whenever a new node is added to the in-memory tree, it's simulations' results ($Q(s, a)$ values) are set to the value provided by the evaluation function, with $N(s)$ and $N(s, a)$ set to a fixed value depending on the level of trust in the evaluation function and required balance between the evaluation function's and simulations' influences.

The Monte-Carlo phase of the UCT routine can also be modified by inclusion of the evaluation function. Following the approach of authors of CadiaPlayer (employing *History Heuristic* as additional evaluation function), it can be used to influence the probability of selecting possible actions so that it is (in some way) proportional to their estimated value. Alternatively, the routine can be modified so that in each and every state there is a (relatively low) probability that the simulation will be stopped and the evaluation function's value returned instead. The latter approach relies on the expectation that the $F(s)$ values are much more reliable in case of positions closer to the end of game. Its advantage is that it can lead to significant improvements in algorithm speed, in case of scenarios where computing subsequent game states is time- or computationally-intensive (as is the case in GGP).

Since in all cases evaluation function F operates on states while Q on state-action pairs, there is a need to redefine the evaluation function directly used by the algorithm to operate on state-action pairs as well. One of the simplest ways to achieve that without any sophisticated analysis is to define it as $F'(s, a) = avg_{t \in N(s,a)}(F(t))$, where $N(s, a)$ denotes the set of all states directly reachable from state s by performing action a (GGP games are deterministic, but players

may perform actions simultaneously, which means that the state reached by performing action a does not depend only on the action performed.)

5 Evaluation function

Construction of evaluation function in the context of GGP framework becomes a task much more difficult than in traditional single-game applications. Since the agent is expected to be able to play any game, whose rules it is provided with, there is virtually no practical way of including any significant expert domain-specific knowledge in the program itself. The evaluation function must be automatically generated by some kind of AI-based routine.

Still, some GGP agents' developers choose to specifically tune their application towards certain classes of problems, expecting tournament organizers to be inspired by real-world human games. Such programs analyze game rule descriptions in search for potential realization of concepts such as boards (usually two-dimensional), pieces and counters [10, 16, 9].

In our application (called *Magician*), however, we decided to concentrate on developing the evaluation function in as knowledge-free a manner as possible and with as few preconceptions as possible. We construct the function as a linear combination of a number of numerical characteristics of game states called *features*. Features are by their nature game-specific and are inferred from the game rules by a set of procedures described in the following sections.

5.1 Features identification

Our approach to game state features identification was inspired by prior work in the GGP area, most notably [10], [2] and [16]. Still, while we employ some of the concepts presented in the above-mentioned papers, our approach remains unique both algorithmically (mixing the concepts in new way and enhancing them with new ideas) and conceptually (being truly preconception-free, as explained earlier, and making use of totally different search routine).

We aim to obtain features represented by expressions similar to those in GDL, e.g. (*cell* ?x ?y b). In order to find the value of such a feature in a given game state, we would attempt to find all values of ?x and ?y variables for which this expression would be true. The number of solutions to the expression is considered the feature value. In case of features with no variables their possible values would only be 0 and 1, of course.

Finding the initial set of possible features is easy: we simply need to analyze the game definition (in GDL) and extract all suitable statements directly from it. This set of expressions becomes the basis for deriving additional ones.

We start by generalizing features, i.e. replacing all constants in all expressions with variables, generating all possible combinations of variables and constants. The effect is that by starting with a single feature (*cell* 1 ? b), we would arrive at a set consisting of four expressions: (*cell* 1 ? b), (*cell* ? ? b), (*cell* 1 ? ?) and (*cell* ? ? ?).

Next, we want to specialize features, i.e. generate features containing less variables than those in the original set, e.g. being provided with a feature (*cell* 1 1 ?), we would like to also generate features (*cell* 1 1 x), (*cell* 1 1 o) and (*cell* 1 1 b). In order to do this in a reasonable way, without generating a huge number of features that would by definition always have zero value, we need to identify valid domains of each and every argument of the predicates we try to specialize.

We do it in a simplified and approximate way, according to a routine inspired by [16]. Basically, we analyze game description to identify variables present in several predicates (or as several arguments of a single predicate). Whenever we find such a case we assume the domains of the arguments to be equal. Afterwards, we analyze the GDL yet again, looking for constants and assigning them to proper domains. This approach does not guarantee that we will actually arrive at precisely the domains of the predicate arguments, but rather their supersets. Still, it has potential to significantly reduce the number of specialized features we will generate.

5.2 Features analysis

Once a set of potential features has been generated, we perform some simple simulations in order to analyze them. We start by generating a set of random game state sequences. Each such sequence starts with a random game state and consists of a small number of consecutive (or almost consecutive, i.e. with a distance of 2 or 3 plies) states resulting from random play by all players. For each sequence we also perform a small number of Monte-Carlo simulations (random plays from its last position) in order to obtain basic statistics about scores expected for each player at the end of the game.

Once this game states pool has been generated and pre-analyzed we use it to compute a number of statistics for each of the identified features. These include its average value (and average absolute value), its variation, minimum and maximum value and so on. Two of the statistics gathered at this point require more attention. Firstly, we calculate each features correlation with the expected final score for each player. Secondly, we calculate a characteristic called stability.

Stability reflects the ratio of feature's variation measured across all game states to average variation within sequences, or more precisely it is calculated according to the formula $S = TV/(TV + 10SV)$, where TV denotes the features total variance (across random game states) and SV – average variance within sequences. The idea behind this characteristic is that more stable features are more promising components of the evaluation function. If a feature's value oscillates wildly between consecutive positions, it has little prognostic value as to the final score of the game (we would arrive at different expectations each time we calculated the feature). Analogically, a feature whose value differs little between separate plays with different results is also of little use in evaluation function.

5.3 Evaluation function generation

Having identified a set of potential game state features, the last step in building a linear evaluation function is selection of the most useful of them and assigning value to each of them. While we plan to employ more advanced Computational Intelligence based approaches to this problem, as the first phase of our research we decided to employ for the task a very simplistic heuristical approach. Considering its simplicity we had low expectations of its performance. At the same time, promising results achieved with such a crude approach form a strong suggestion that choices made so far have been the right ones.

The actual procedure we use for building the evaluation function consists of 2 steps. First, we order features by the minimum of their stability and absolute value of their correlation with the final score. While these two values are not directly comparable, they both fall into the 0-1 interval and we prefer features having both these characteristics as high as possible. Afterwards, we choose the first 30 features to form the evaluation function. The linear combination of the features is created by assigning them weights equal to the product of their stabilities and correlations with the final score.

6 Experiment

6.1 Experiment setup

In order to test the quality of the GUCT algorithm in cooperation with the simple evaluation functions generated by our application, we decided to run a small competition comparing players using GUCT and UCT in 3 games of various complexity: connect-4, checkers and chess. All game definitions have been downloaded from [7] and while their rules may differ slightly from the generally accepted ones due to GDL limitations (e.g. difficulty in expressing the rule of a draw by position repetition), these differences should not be significant.

For the sake of fair comparison, both competing agents were based on the same single-threaded implementation of the UCT algorithm. GUCT player made use of the evaluation function only in the Monte-Carlo simulations phase, stopping the simulation and using the evaluation function's value as the result with the probability of 0.1 in each position - a value that guaranteed significant influence of the evaluation function on the agent's decisions, especially in the early game phases.

All in all, tournament consisted of 60 matches in total, 20 matches for each game - 4 per time limit for move of 1s, 10s, 15s, 30s and 60s. Players swapped sides after each game. GUCT player regenerated its evaluation function from scratch before each and every match. Each player was rewarded 1 point for a victory, -1 point for a loss and 0 – for a draw.

6.2 Experiment results

The tournament results for GUCT player are presented in figure 3. Please keep in mind that any score above 0 indicates player's supremacy over plain UCT

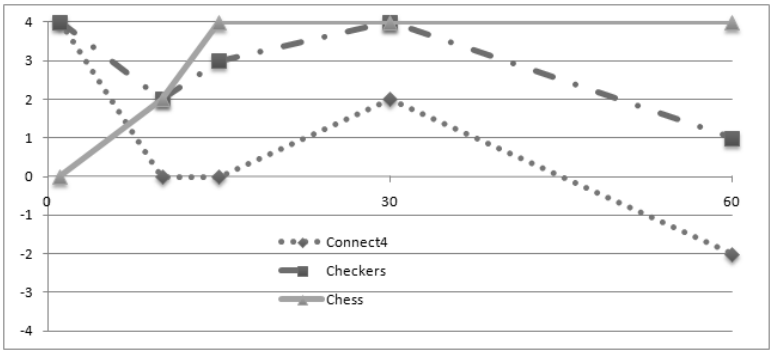

Fig. 3. *GUCT* player scores for each game depending on time allowed per move (in seconds)

approach. First and most obvious observation here is that, considering the simplicity of the evaluation function generation procedure, GUCT player fares unexpectedly well, significantly outperforming its opponent in all games.

More detailed analysis of the results leads to two interesting observations regarding the dependence of the algorithm's performance on time limit per move. Both of them can only be treated as hypotheses considering limited experimental data but anecdotal data gathered during development and preliminary testing of the system strongly supports them as well.

Firstly, in case of very low time limits and sophisticated games, results of the games are often insignificant (typically being a draw), as neither player has enough time for analysis to play in a reasonable way. This effect leads to a tie between the competitors in case of chess with 1s time limit - actually all the tournament matches ended with a draw in this case. As soon as the strict time limits are lifted, however, GUCT approach clearly shows its upper hand, winning all games with time limits of 15s and more. Even with the relatively high time limit of 1 minute per move, plain UCT player is still unable to effectively compete against its opponent.

At the same time, as the time limit per move is increased, another effect can be observed – especially in the case of simpler games. While the UCT player is able to perform more and more simulations, obtaining more and more precise results, GUCT-based agent still heavily relies on the very rudimentary evaluation function, whose quality remains constant. This leads to the UCT approach slowly gaining upper hand in the matches. This phenomenon explains poor results of our player in case of connect-4 with the highest time limit per move (although it is worth noticing that connect-4 clearly proved to be a game for which it was relatively difficult for our system to generate a reasonable evaluation function in general).

7 Conclusions

As presented above, experiments we have performed so far, strongly suggest that our approaches to both modification of the UCT tree search algorithm and automated game-independent process of creating evaluation function have high potential. Our feature-building strategy follows two principles typical for human thinking: generalization and specialization. While the former process is useful for generating new concepts by ignoring certain details of the problem aspects, the latter allows applying the concepts to specific situations and finding special cases and exceptions to the rules. It is the unique synergy of the two approaches that facilitates solving even seemingly distant and unrelated tasks.

8 Future research

At the moment, our immediate research plans include two paths of further system development. Firstly, we intend to further enhance feature generation system by including compound features defined as differences or ratios of related simple features. This way it would be possible to construct features such as difference in checkers count of both players etc.

Secondly, we are working on more sophisticated, CI-based methods of evaluation functions generation. In the immediate future, we consider employing co-evolutionary and/or Layered Learning [13] schemes. Employment of artificial neural networks instead of linear evaluation functions remains an open possibility as well.

References

1. P. Auer, N. Cesa-Bianchi, P. Fischer: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2/3), pages 235–256, 2002
2. J. Clune: Heuristic evaluation functions for General Game Playing. In Proceed- ings of the Twenty-Second AAAI Conference on Articial Intelligence (AAAI- 07), pages 1134–1139, Vancouver, BC, Canada, 2007. AAAI Press.
3. H. Finnsson, Y. Björnsson: Simulation-based approach to General Game Playing. In Proceedings of the Twenty-Third AAAI Conference on Articial Intelligence (AAAI-08), pages 259–264, Chicago, IL, 2008. AAAI Press.
4. S. Gelly, Y. Wang: Exploration exploitation in Go: UCT for Monte-Carlo Go. In Neural Information Processsing Systems 2006 Workshop on On-line trading of exploration and exploitation, 2006
5. S. Gelly, Y. Wang, R. Munos, O. Teytaud: Modification of UCT with patterns on Monte Carlo Go. Technical Report 6062, INRIA, 2006
6. General Game Playing website by Stanford University. http://games.stanford.edu/.
7. General Game Playing website by Dresden University of Technology. http://www.general-game-playing.de/.
8. M. Genesereth, N. Love: General Game Playing: Overview of the AAAI Competition. http://games.stanford.edu/competition/misc/aaai.pdf, 2005.
9. D. Kaiser: The Design and Implementation of a Successful General Game Playing Agent. In Proceedings of FLAIRS Conference, pages 110–115, 2007

10. G. Kuhlmann, K. Dresner, and P. Stone: Automatic heuristic construction in a complete General Game Player. In Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence (AAAI-06), pages 1457–1462, Boston, MA, 2006. AAAI Press.
11. N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth: General Game Playing: Game Description Language Specification. http://games.stanford.edu/language/spec/gdl_spec_2008_03.pdf, 2008.
12. J. Mańdziuk, *Knowledge-Free and Learning-Based Methods in Intelligenet Game Playing*, ser. Studies in Computational Intelligence. Berlin, Heidelberg: Springer-Verlag, 2010, vol. 276.
13. J. Mańdziuk, M. Kusiak, K. Walędzik: Evolutionary-based heuristic generators for checkers and give-away checkers. Expert Systems, 24(4): 189–211, Blackwell-Publishing, 2007.
14. J. Reisinger, E. Bahçeci, I. Karpov and R. Miikkulainen: Coevolving strategies for general game playing. In Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG'07), pages 320–327, Honolulu, Hawaii, 2007. IEEE Press.
15. J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317:1518–1522, 2007.
16. S. Schiffel, M.Thielscher: Automatic Construction of a Heuristic Search Function for General Game Playing. In Seventh IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC07).