Generic Heuristic Approach to General Game Playing

Jacek Mańdziuk[1] and Maciej Świechowski[2]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland; j.mandziuk@mini.pw.edu.pl
[2] Phd Studies at Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland; m.swiechowski@ibspan.waw.pl

Abstract. General Game Playing (GGP) is a specially designed environment for creating and testing competitive agents which can play variety of games. The fundamental motivation is to advance the development of various artificial intelligence methods operating together in a previously unknown environment. This approach extrapolates better on real world problems and follows artificial intelligence paradigms better than dedicated single-game optimized solutions. This paper presents a universal method of constructing the heuristic evaluation function for any game playable within the GGP framework. The algorithm embraces distinctive discovery of candidate features to be included in the evaluation function and learning their correlations with actions performed by the players and the game score. Our method integrates well with the UCT algorithm which is currently the state-of-the-art approach in GGP.

Keywords: General Game Playing, Heuristic Evaluation, Automatic Learning, Monte Carlo Simulations.

1 Introduction

Computer systems able to play a particular game such as chess or checkers have always been in the interest of Artificial Intelligence (AI). One of the most prominent examples is Deep Blue [1] - chess-playing machine which successfully challenged Garri Kasparov. Such programs, however, are equipped with game specific knowledge and heavily rely on computational power rather than intelligent behavior. General Game Playing (GGP) represents a new trend in AI focused on the ability of playing many different games previously unknown to the playing system. Given the game rules written in the so-called GDL (Game Description Language) [2] a playing agent takes various actions towards learning and mastering the game. This includes analysis of the game rules, application of various learning and searching mechanisms, logic-based reasoning methods, efficient knowledge representation and many other techniques [3–5]. Integration of all these elements formulates an interesting and challenging research task. Before playing a game an agent is a *Tabula Rasa* - no game specific features should be assumed *a priori*. GGP took its name from the competition proposed

by Stanford Logic Group in 2005. It is held annually at AAAI (or IJCAI in 2011) conferences. Playing environment contains central unit called Gamemaster and remote playing agents called Game Players. Game Players communicate via http protocol with the Gamemaster only whose role is to provide players with the rules in the GDL, running the game, sending control messages and receiving responses. Gamemaster also includes its own GDL reasoning mechanism in order to validate legality of the players' moves and updates the state. If an agent responded with an illegal move, a random move would be selected for them. Each agent is then notified about moves performed by other players. The contest features two timers: a move clock and a start clock. The first one counts time available for notifying the Gamemaster about selected action and the latter represents time for preparation before the actual start of the game. Hence, the start clock defines room for application of various pre-game learning strategies.

1.1 The Class of Considered Games

Any game which is finite, deterministic and synchronous can be played within GGP framework. The term finite should be understood as finite number of players and available actions (moves) in any game state and finite number of states. One distinguished state is marked as initial and at least one as terminal. Each terminal state has goal values defined for each player. Goal values range from 0 to 100. Due to deterministic nature of the game a state can change only as a result of performed move and there is no randomness. Players perform moves simultaneously (synchronously) during the update phase, but turn-based games can be easily simulated with the use of no-operation moves. In this scenario, for all players but one players the no-operation move is the only legal move available for them in the current state.

1.2 Game Description Language

GDL is a formal first-order logic language with the structure strictly following Datalog [2], which in turn is a subset of Prolog. Terms used in game descriptions compose sentences that are true in particular states. There are a few distinguished keywords which cannot be redefined. As an example a partial listing of Tic-Tac-Toe game written in GDL is presented below.

```
 (role xplayer) (role oplayer)
(init (cell 1 1 b)) (init (cell 1 2 b))
   ***
(init (cell 3 3 b))
(init (control xplayer))
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
   ***
(<= (next (control oplayer)   (true (control xplayer)))
(<= (row ?m ?x)
```

```
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
   ***
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (legal xplayer noop)   (true (control oplayer)))
(<= (goal xplayer 100)    (line x))
(<= (goal xplayer 50)
    (not (line x))
    (not (line o))
    (not open))
(<= (goal xplayer 0)   (line o))
   ***
(<= terminal   (line x)) (<= terminal   (line o)) (<= terminal (not
open))
```

[A subset of Tic-Tac-Toe game definition downloaded from Dresden GGP Server [6]]
A complete set of keywords consists of the following elements: *role, init, true, does, next, legal, goal, terminal, distinct.* They are used to define the initial state, legal moves, state update procedure as well as game terminal states and goals accomplishment. A more detailed description of all keywords can be found in [2]. There are also logical operators available in GDL, such as *not, or,* and <=. A special symbol ? is used to make an argument a variable - in this case a set of symbols satisfying the truth condition is to be calculated. For example: relation (cell ?m 1 ?x) has two variable arguments (?m ?x) and one constant (1). Negation and recursion, in restricted form, are both part of the language too.

2 State-of-the-art

GGP annual competition provides an environment for testing the strength of game playing algorithms. Last years were dominated by two winning approaches: CadiaPlayer (2007, 2008) presented in [7] and Ary (2009, 2010) described in [8]. Both agents rely on performing Monte Carlo simulations (MCS) aimed at learning the game and incrementally building the game tree. This solution was inspired by Go playing agents [9]. MCS perform random play from the current state to the terminal state. The goal value is then obtained and stored in the current node of the tree. Storing the entire tree in memory using all visited nodes on simulation paths would quickly exceed the available memory. Therefore, only one node, representing the first action, is added after a single simulation [7]. The most popular variant of MCS is known as Upper Confidence Bounds Applied for Trees (UCT) method [9] which efficiently keeps balance between exploration and exploitation. As the name suggests, the UCT method is a generalization of the Upper Confidence Bounds (UCB) [10] which can be used, for example, to learn the payoff distribution of slot machines in a casino. The goal of UCT is to perform MCS as wisely as possible taking advantage of the knowledge acquired

so far. In each node the algorithm checks if all possible moves in the associated position have already been tried at least once during simulation (therefore they possess initial MC estimations). If not, one of the unvisited child nodes is chosen at random. Otherwise (i.e. in case all successors of the current node have been visited at least once), the move a^* is chosen according to the following rule:

$$a^* = arg \max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{ln\,[N(s)]}{N(s,a))}} \right\} \quad (1)$$

where a - is an action; s - the current state; *A(s)* - a set of actions available in state s; *Q(s,a)* - an evaluation of performing action a in state s; *N(s)* - a number of previous visits of state s; *N(s,a)* - the number of times an action has been sampled in state s; C - a coefficient defining a degree to which the bonus (second component) should be considered. UCT-based players, such as the above-mentioned Ary and Cadia build a game tree gradually. Each node stores the average payoff, obtained by those MCS, in which it was visited on the path of play. Simulations are not terminated when the start time elapses but continue through the entire GGP episode. During the actual game, if a player chooses an action stored in a node, that node becomes a new tree root (i.e. all higher branches are deleted, because they are not needed anymore). A new simulation always runs from the current game state. Most of the UCT based players share the basic idea described above but differ by employing specific search control mechanism to optimize tree exploration [11]. .

3 Automatic Construction of the Evaluation Function

One of the potential enhancements of purely simulation-based UCT implementation is combining it with the use of some kind of evaluation function. Due to wide variety of GGP games it is hardly (if at all) possible to design such a generally-applicable function *a priori* and only tune its coefficients for a particular game. Despite the above difficulties, several researchers have explored this possibility and some heuristic approaches to General Game Playing used predefined candidates for the evaluation procedure. ClunePlayer [12] considered mobility, payoff and termination stability. Fluxplayer [13], harnessed predefined syntactic templates for common game features like board definition or successor relation. Fluxplayer's heuristic construction mechanism is an extension to the idea derived earlier in [15]. Another noteworthy approach, adopted by OGRE player [16], focuses mainly on board games. It uses the so-called evaluators. The game structure evaluators are *distance-initial*, *distance-to-target*, *count-pieces* and *occupied-columns* whereas game definition evaluators are *count-moves*, *depth*, *exact*, *pattern* and *purse* [16]. OGRE came 4-th out of 12 entrants in 2006 competition winning 34% of the matches [17].

Generally speaking, all the above-mentioned methods were geared towards standard two players board games and, in random environment, they tend to lose againts UCT-based players. Our approach constructs features that are completely independent of particular game definition. The key difference is that no

predefined templates are present. Its underpinning idea is related to identification of meaningful numbers from the symbol representation. Before going into details let us introduce database-like vocabulary used to describe the elements of GDL.

- A ***row*** is a complete term in GDL that describes the game fact in a particular state. By a fact we mean a statement which is true. A row consists of a name of the fact and its arguments (called ***symbols***).
- A ***table*** is a name of the row; in our example it is *cell*. ***TableRows*** are all rows sharing a common name;
- A ***column*** is a set of symbols at a fixed position in rows' argument lists.

The proposed algorithm for automatic construction of the evaluation function for a given GGP game consists of three phases: selection, construction and play.

3.1 Selection Phase

The aim of this phase is to select the candidate features for heuristic function. The main idea is to track cardinality of three kinds of objects:

- For a given table count its rows (***TableRows***);
- For a given table, column index and symbol count the symbol occurrences in the corresponding column in the table (***ColumnSymbols***);
- For a given table, column index and symbol extract the set of rows with the matching symbol in the corresponding column in a given table ***(SymbolRows)***.

The essential part of the algorithm is the way the features are counted. This issue is described in detail below.

Selection phase - step 1 - parallel simulations In the first selection phase TableRows and ColumnSymbols are found. Not all of them are selected but only those with **occurrence count varying** during the game in a manner which depends on the performed moves. It means that, if for a current game state all legal moves produce pairwise equal changes to the object's quantity, then such occurrence is neglected by the algorithm. The motivation behind this constraint is to discard all features a player has no impact on (such as counters, timers, control, board cells count etc.) and focus on real move consequences. In order to perform the selection, N simulations with random move making are launched in parallel. Parameter N can be tuned depending on how much time is available. Tests show that 3-4 parallel simulations are usually sufficient. This phase terminates when there is only one or none unfinished simulations left. At each simulation step TableRows and ColumnSymbols are counted independently for each simulation and their counters are tested against each other. If a difference occurs, an object (a table with all its TableRows or ColumnSymbol) is marked as changing and excluded from further tests. It is important to note that a difference is computed only between the same steps of each of (different) simulations. No difference is computed between consecutive steps. Objects marked as changing are stored for further use.

Selection phase - step 2 - extracting symbols In the second step of selection phase, only one complete random simulation is performed. Let (s_1, s_2, s_3, ..., s_{n-1}, s_n) represent consecutive states present during the simulated game. After reaching each state s_i two additional random moves are simulated that lead to hypothetical states s_{ij} and s_{ik}. The difference between the two new states s_{ij} and s_{ik} is analyzed from the heuristic selection viewpoint. The main simulation continues as if it was not affected by the two moves and SymbolRows sets are constructed for each symbol. The general idea behind this part of the algorithm is to filter symbols which express the most important features within the relation. The most important features are usually dynamic and the rest of symbols which they appear with, represent their properties which vary from state to state. Therefore features supposed to play important role will change their set of properties often. The following measure of symbols' variation was used:

$$val = 1 - \frac{2 * |A_{ij} \cap A_{ik}|}{|A_{ij}| + |A_{ik}|} \tag{2}$$

where A_{ij} and A_{ik} denote SymbolRows for a particular symbol in states s_{ij} and s_{ik}. Formula (2) is used to calculate variation of each symbol during the selection step. The most varying symbol (with the highest computed value) at each step is marked as changing and becomes a candidate for further heuristic weighting. If a selected symbol has already been marked before, this new selection is ignored.

3.2 Construction of a Heuristic Function

During the selection phase some objects marked as changing are captured. These can either be TableRows, ColumnSymbols or SymbolRows. The occurrence count is correlated with the actions selected by a player during the game. The purpose of the construction phase is to approximate the correlation factor by assigning weights to the counters. Here come MCS with preferable UCT enhancement which are run until the time is up. These MCS are used to assign weights to discovered elements of the evaluation function. The following pseudocode describes the weight-learning phase:

```
ConstructHeuristic(TimeLimit,Player)
While(currentTime < TimeLimit)
    Start a new simulation S
    While S not finished
        SavedStates->Push(S->State)
        S->Advance
    If IsSuccess(S->State,Player)
        CountOccurences(SavedStates)
      CountAverages()
      AddAverages(WinAverageList);
    Else If IsFail(S->State,Player)
    CountOccurences(SavedStates)
      CountAverages()
      AddAverages(LossAverageList);
```

```
For each heuristic object:
    Count winAverage
    Count lossAverage
  weight = C*(winAverage - lossAverage)/MaxValue
End
```

[A pseudocode of the weights-learning phase.]

Although the detailed algorithm appears to be rather complicated, its underlying idea is quite simple. During simulations the counting procedure is performed and the game-average result is computed. This value is put into collection of either 'win values' or 'loss values' depending on the game result assigned to the player for whom the heuristic is being defined. The distinction between a won and lost game can be defined in various ways. In our approach it was:

```
 AverageResult =
(MaxResult - MinResult)/2 If(Result > AverageResult)
    Win = true
Else If(Result < AverageResult)
    Loss = true;
Else //no action
    Win = Loss = false
```

[A pseudocode for determining the win or loss. MaxResult and MinResult come from the GDL definition.]

In the final step of this phase, game-average values from win and loss collections are transformed into single averages for won and lost games independently. For each counter the maximum occurrence ever (MaxValue) is monitored and used for the normalization purpose (see an example below).

```
//Data structures after a completed simulation:
CurrentGameValues = [6,4,2,0] //symbol occurrences
WinAverageValues = [4.5, 4.5]
LossAverageValues = [3,2]
MaxValue = 6
//Computing current average
CurrentAverage = (6+4+2+0)/4 = 3
//Let assume the game was won
WinAverageValues = [4.5, 4.5, 3] //3 is added
//Computing single average values for won and lost games
WinAverage = (4.5+4.5+3)/3 = 4
LossAverage = (3+2)/2 = 2.5
```

[Illustration of how the average values are computed.]

The heuristic value for each object is computed according to the following formula:

$$weight = C * \frac{WinAverage - LossAverage}{MaxValue} \tag{3}$$

where C is a constant parameter for each of the three types of heuristic elements. In the current implementation of the system we use $C = 1.0$ for both TableRows and ColumnSymbols and $C = 0.2$ for SymbolRows. Concrete rows, counted for symbols, have lesser weights in order to force them to be used if no legal action positively changes TableRows or ColumnSymbols. The final evaluation function is a linear combination of the numbers of occurrences of the elements multiplied by the computed weights.

3.3 The Use of the Heuristic Function

The evaluation method constructed as described in sections 3.1 - 3.2 takes game state as an input, performs weights calculation according to (3) and returns a single floating point value. Such automatically constructed heuristic can be taken advantage of in several ways by the playing agent. It can be used for

(1) evaluation of each legal move in order to choose the heuristically best one;
(2) incremental building of a min-max inspired game tree (in that case a distinct heuristic function is maintained for each player);
(3) sorting unplayed actions in the classic MC + UCT solution (evaluation function helps to determine which branches should be tried first);
(4) to replace the whole or part of the MC phase with certain probability in the classic MC + UCT approach [14].

In the experimental evaluation of the proposed method our focus was on facets (2). We decided to fully utilize start clock on heuristic function construction and move clock for min-max tree search. Such distribution was chosen because it is natural and easy to maintain. However, for some games a different balance might be more beneficial.

4 Empirical Results

An agent using the heuristic function was tested against reference UCT solution based on CadiaPlayer description [7] in several games downloaded from the Dresden General Game Playing Server site [6]. The main criterion for choosing games was to focus on most widely recognized games of various rules and complexity. The games included bomberman, breakthrough, checkers, chess, connnectfour, farmers, othello, pacman, tic-tac-toe, sheep and wolf and wallmaze. Each of these 11 games was played 80 times with four different pairs of clocks settings (start clock, move clock), i.e. (16T,2T), (32T,4T), (64T,8T) and (128T,16T) where T is a game-specific parameter proportional to the average time of one random simulation from the beginning to the end for particular game. The exact value depends on game complexity. In each case, in half of the games the Heuristic Player (our algorithm) was making the first move and it the remaining half of them the UCT was the initial player.

In majority of tested games interesting features were selected for the heuristics. For example high value is always assigned in chess to the TableRow **check**

Table 1. Percentage results between the Heuristic Player and UCT for short times. The interpretation is the following: Heuristic Player win ratio - UCT win ratio (the remaining games are ties). The results in favor for the Heuristic Player are bolded.

Game	**HP vs UCT**. Clocks = [16T,2T]	**HP vs UCT.** Clocks = [32T,4T]
Bomberman	6% - 94%	11% - 86%
Breakthrough	50% - 50%	50% - 50%
Checkers	**64%** - 22%	**66%** - 20%
Chess	**14%** - 0%	**35%** - 10%
Connectfour	**48%** - 40%	**45%** - 44%
Farmers	36% - 64%	32% - 68%
Othello	**50%** - 25%	29% - 49%
Pacman	**78%** - 22%	**75%** - 25%
Tic-Tac-Toe	**55%** - 25%	30% - 33%
Sheep and Wolf	**89%** - 11%	**76%** - 24%
Wallmaze	**3%** - 0%	**6%** - 0%

making the player perform checking the opponent whenever possible. Our player achieves better win ratio in almost all games for the shortest of tested times, with bomberman and farmers being the only exceptions. The results prove that the evaluation function constructed during the preparation time has a positive impact on playing quality. With the increase of time the UCT becomes a stronger opponent. It is mainly due to a greater impact of MCS performed during the move clock. With extreme time limits, most parts of the tree constructed by UCT approach will have an exact goal values fetched directly from terminal states, whereas min-max algorithm requires a full tree expansion in order to fetch at least one real goal value. This is a key difference between the methods in terms of a tree search. For the longest time settings the heuristic player remained superior in five games. Chess and checkers are games which are in favor for our method in a most significant way, whereas bomberman and farmers are

Table 2. Percentage results between the Heuristic Player and UCT for longer times. See description of Table 1.

Game	**HP vs UCT**. Clocks = [64T,8T]	**HP vs UCT.** Clocks = [128T,16T]
Bomberman	21% - 70%	14% - 77%
Breakthrough	48% - 52%	37% - 63%
Checkers	**84%** - 10%	**74%** - 16%
Chess	**45%** - 9%	**23%** - 4%
Connectfour	45% - 46%	41% - 51%
Farmers	14% - 86%	11% - 89%
Othello	38% - 49%	30% - 54%
Pacman	**55%** - 45%	**51%** - 49%
Tic-Tac-Toe	44% - 46%	24% - 58%
Sheep and Wolf	**70%** - 30%	**58%** - 42%
Wallmaze	**29%** - 25%	**34%** - 30%

games at which the UCT is undeniably better. Below we present two evaluation functions obtained for chess and bomberman, respectively.

Chess. The evaluation function primarily forces to check the opponent whenever possible and rewards having a greater number of particular pieces. Piece types have various levels of importance assigned. Moreover, board positions where a threat to adversary king's initial location (coordinates) is applicable are slightly favored. The most varying symbols are discovered as *wp, wn, wb, wq, wr, bp, bn, bb, bq, br* for *cell*. These symbols represent chess pieces. For example *wn* stands for white knight and *bq* stands for black queen.

1. **TableRows:**
   a (check, 0.8); (pawn_moved_two, -0.006); (piece_has_moved, -0.25).
   These symbols represent one-time actions that can occur after a move.
2. **ColumnSymbols:**
   a For cell at column 2: (b,-0.008); (bb, -0.08); (bn, -0.04); (bp, 0.1); (bq; -0.21); (br; -0.12); (wb, 0.29); (wn, 0.16); (wp, 0.11); (wq, 0.52); (wr; 0).
   Pieces counts (known as strength of the material) are captured here.
   b For check at column 0: (black, 0.36); (white, -0.43).
   The white player is advised to check the black player.
   c More values are discovered for all symbols in check,
   pawn_moved_two, piece_has_moved, but with little impact.
3. **SymbolRows:**
   a All rows with varying symbols that occurred during simulations, e.g. (cell a 4 wq, 0.012). Values are within the range (-0.02, 0.02).
   Particular board cells with concrete pieces are evaluated here.

Bomberman. This is an example of a game for which usually only one varying symbol is discovered. It is 1 at the first index of *blockedeast* table. It turned out that one random simulation in step 2 of the selection phase (3.1) is insufficient since, if acting randomly, a player has 50% chance of dying because of its own bomb in the first turn of the game. The only legal action is to place a bomb or move in unblocked direction. The probability of losing in a few turns drastically increases. UCT is capable of finding the safe path if given enough time.

1. **TableRows:** (location, -0.1).
2. **ColumnSymbols:**
   a For table location, column 0: (bomb0, -0.5); (bomb1, -0.11); (bomb2, -0.05); (bomb3, 0).
   b For location, columns 1 and 2: not meaningful values.
3. **SymbolRows:** (blockedeast 1 2, 0) (blockedeast 1 7, 0).

The term *location* which appeared above describes rows with the following structure *(location ?object ?x ?y)*. Its column 0 (?object) contains symbols representing object located at (?x,?y) coordinates (columns 1,2). Possible objects are bomberman, bomberwoman, bomb0, bomb1, bomb2 and bomb3. The term blockedeast means, if present, that east direction is not available at particular coordinates. The game description includes also north direction which was not selected by the algorithm. The goal of the game is to avoid bombs (of any players) and make the opponent die by a bomb.

5 Conclusion

A novel approach to building a heuristic evaluation function for General Game Playing has been presented. The proposed algorithm clearly outperforms UCT in four out of eleven games while losing visibly in three games. The introduced heuristic evaluation is constructed in a fully automatic way. Instead of using predefined categories it counts occurrences of carefully filtered elements of three different kinds. GDL description is lexical by nature and there are no ready-to-use numbers encoded (even mathematical operators must be explicitly defined for all possible arguments of lexical symbols). Our solution not only extracts numbers but also assesses their usefulness. Only elements whose values' variability is caused by a player's move are considered which is an essential idea of the algorithm. Their usefulness is further remodeled by their correlation with a game score. Experiments show that the method is well suited for games in which some 'objects' are created, destroyed or moved. Objects can be of any type like money, pieces, buildings, obstacles etc. The evaluation function can be inserted at several stages of GGP scenario. It can be used to guide MC or UCT search, which vastly improves the quality of play for games it normally performs bad at and moderately decreases the quality of play for games it has advantage in. The integration may even proceed a step further - the agent may discover, during the learning phase, whether using plain heuristic approach or guided UCT gives better results. Better strategy may by dynamically chosen for the actual play. The other way to improve the method would be to incorporate it into a complex agent featuring various heuristic functions and equipped with a feedback-based learning mechanism to choose the best suited evaluation for a currently playing game.

The main weak point of the proposed solution is that quality of the computed function greatly depends on the numbers of simulations which fall into won and lost categories. If a game lasts for a very long time or a tie is the typical result, the evaluation function may be inaccurate. Such games are also difficult to master by UCT approaches because most of the results propagated in a tree are ties. Our current work concentrates on extension of the proposed method towards automatic discovery of correlations among particular game elements (game aspects) which would allow capturing their dynamic (changing in time) mutual dependencies. This issue, in its general form, is highly challenging. To the authors' knowledge no universal solution for automatic discovery of such correlations exists, even in well-researched classical board games domain (chess, checkers, Go, Othello, etc.).

Acknowledgments. Maciej Świechowski would like to thank Foundation for Polish Science under International PhD Projects in Intelligent Computing. Project financed from The European Union within the Innovative Economy Operational Programme 2007-2013 and European Regional Development Fund.

References

1. Newborn, M.: Kasparov versus Deep Blue: Computer Chess Comes of Age, Springer-Verlag (1997)
2. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical Report LG-2006-01 (2006) Available at: `http://games.stanford.edu`
3. Genesereth, M., Love, N.: General Game Playing: Overview of the AAAI competition, AI Magazine, vol. 26, 62-72 (2005)
4. Walędzik, K., Mańdziuk, J.: CI in general game playing: to date achievements and perspectives, ICAISC'10 Proc. 10th International Conference on Artificial Intelligence and Soft Computing: Part II, Springer-Verlag Berlin, Heidelberg (2010)
5. Mańdziuk, J.: Knowledge-Free and Learning-Based Methods in Intelligent Game Playing, vol 276 of Studies in Computational Intelligence, Springer-Verlag, chapter 14.5 (2010)
6. Dresden GGP Server: `http://euklid.inf.tu-dresden.de:8180/ggpserver`
7. Bjornsson, Y., Finnsson, H.: CadiaPlayer: A Simulation-Based General Game Player, IEEE Transactions on Computational Intelligence and AI in Games, Vol. 1, No. 1., pp. 4-15 (2009)
8. Mėhat, J., Cazenave, T.: Ary, a General Game Playing Program, Board Games Studies Colloqium, Paris (2010)
9. Gelly, S., Wang, Y.: Exploration and Exploitation in Go: UCT for Monte-Carlo Go, 20th Annual Conference on Neural Information Processing Systems NIPS (2006)
10. Auer, P.: Using upper confidence bounds for online learning, FOCS '00 Proceedings of the 41st Annual Symposium on Foundations of Computer Science (2000)
11. Bjornsson, Y., Finnsson, H.: Simulation Control in General Game Playing Agents, In. Proc. IJCAI-09 Workshop on General Game Playing (GIGA'09) (2009)
12. Clune, J.: Heuristic evaluation functions for general game playing. Proc. AAAI Nat. Conf. on Artificial Intelligence, Vancouver, AAAI Press, pp. 1134-1139 (2007)
13. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, AAAI Press 1191-1196 (2007)
14. Walędzik K., Mańdziuk J.: Multigame playing by means of UCT enhanced with automatically generated evaluation functions, 4th Conf. on Artificial General Intelligence, Mountain View, CA, LNAI vol. 6830, 327-332, 2011
15. Kuhlman, G., Dresner K., Stone P.: Automatic Heuristic Construction in a Complete General Game Player. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence, pp. 1457-1462 (2006)
16. Kaiser, D.: Automatic Feature Extraction for Autonomous General Game Playing Agents. In: Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multiagent Systems, (2007)
17. Love, N.:2006 General Game Playing Competition Results: `http://euklid.inf.tu-dresden.de:8180/ggpserver` accessed, (2006)