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.

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

```
init(cell(1, 1, b)) ... init(cell(3, 3, b)).
init(control(xplayer)).
legal(W, mark(X, Y)) :- true(cell(X, Y, b)),
    true(control(W)).
legal(oplayer, noop) :-
  true(control(xplayer)).
...
next(cell(M, N, x)) :-
  does(xplayer, mark(M, N)),
  true(cell(M, N, b)).
next(control(oplayer)) :-
  true(control(xplayer)).
...
diagonal(X) :- true(cell(1, 1, X)),
  true(cell(2, 2, X)), true(cell(3, 3, X)).
goal(xplayer, 100) :- diagonal(x).
goal(xplayer, 0) :- diagonal(o).
terminal :- diagonal(x).
...
```

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: $\text{next}(X) :- F_i$
- $\mathcal{R}[\text{distinct}(a_1, a_2)] = \text{distinct}(a_1, a_2)$
- 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.

[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.

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

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

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:

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

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.

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

[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$(cell(1,1,x))$:

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

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.

$$\big(\text{true}(cell(2,2,b)) \land \text{F}\big) \lor \text{true}(cell(2,2,x)) \tag{5}$$
$$\text{true}(cell(2,2,x)) \tag{6}$$

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:

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

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

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

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:

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

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 Playout	Tree	Combi.	Relative number of simuls. 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.