First-Order Logic with Counting for General Game Playing

Łukasz Kaiser
CNRS & LIAFA
Paris

Łukasz Stafiniak
Institute of Computer Science
University of Wrocław

Abstract

General Game Players (GGPs) are programs which can play an arbitrary game given only its rules and the Game Description Language (GDL) is a variant of Datalog used in GGP competitions to specify the rules. GDL inherits from Datalog the use of Horn clauses as rules and recursion, but it too requires stratification and does not allow to use quantifiers. We present an alternative formalism for game descriptions which is based on first-order logic (FO). States of the game are represented by relational structures, legal moves by structure rewriting rules guarded by FO formulas, and the goals of the players by formulas which extend FO with counting. The advantage of our formalism comes from more explicit state representation and from the use of quantifiers in formulas. We show how to exploit existential quantification in players' goals to generate heuristics for evaluating positions in the game. The derived heuristics are good enough for a basic alpha-beta agent to win against state of the art GGP.

Introduction

The goal of General Game Playing (GGP) is to construct a program able to play an arbitrary game on an expert level without additional information about the game. Currently, the focus in GGP is put on playing non-stochastic games with perfect information, such as Chess or Checkers, while extensions to games with imperfect information and stochastic behavior, such as Poker, have been developed only recently (Thielscher 2010). Even for perfect-information zero-sum games, the task of a general game player is a formidable one, as evidenced by the introduction of various new techniques in each annual AAAI GGP Competition.

During the GGP competition, games are specified in the Game Description Language (GDL), which is a variant of Datalog, cf. (Genesereth and Love 2005). A successful GGP agent must reason about the rules of the game and extract from them game-specific knowledge, such as heuristic functions used to evaluate positions during the game. To facilitate the creation of good general players, GDL was designed as a high-level, declarative language. Still, in recent years (see GGP Competition results 07-10[1]) the players which only derive evaluation functions based on reasoning, cf. (Clune 2008), have often lost against Monte-Carlo based players, which rely far less on symbolic deduction and more on optimized tree-searching with Monte-Carlo simulations for evaluating positions (Finnsson and Björnsson 2008).

In this paper, we present another formalism for game description, which is even higher-level and more declarative than GDL, and we exploit it to beat a state of the art GGP player using only minimax search with logically derived heuristics for position evaluation. The introduced description language also allows to specify games in a more compact and, in our opinion, more natural way than GDL. The state of a game in our formalism is represented by a relational structure and legal moves are given by structure rewriting rules guarded by formulas of first-order logic (FO). The payoffs the players receive at the end are described by counting terms, which extend FO in a way similar to the one presented in (Grädel and Gurevich 1998). These terms can be manipulated easier than GDL descriptions of payoffs.

From the counting terms in the payoffs and the constraints of the moves we derive heuristics for evaluating game positions. These heuristics are good enough for a basic game player, Toss, which only performs a minimax search with alpha-beta pruning, to beat a state of the art GGP agent, Fluxplayer. Fluxplayer has ranked among the top 5 in each GGP competition in the last 5 years and was the only one of the top players we could access online for testing. While Toss does not win in all the games we tested, it generally plays on par with Fluxplayer in games not well-suited for simple minimax (e.g. in Connect5) and decisively outperforms Fluxplayer in other games (e.g. in Breakthrough).

Relational Structure Rewriting

The state of the game is represented in our formalism by a finite relational structure, i.e. a labeled directed hypergraph. A relational structure $\mathfrak{A} = (A, R_1, \ldots, R_k)$ is composed of a universe A (denoted by the same letter, no fraktur) and a number of relations. We write r_i for the arity of R_i, so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \ldots, R_k\}$.

The moves of the players are described by *structure rewriting rules*, a generalized form of term and graph rewriting. Structure rewriting has been introduced in (Rajlich 1973), for games in (Kaiser 2009), and is most recognized in graph rewriting and software engineering communities, where it is regarded as easy to understand and verify.

In our setting, a rule $\mathfrak{L} \to_s \mathfrak{R}$ consists of two finite relational structures, $\mathfrak{L}$ and $\mathfrak{R}$, over the same signature, and of

[1] cadia.ru.is/wiki/public:cadiaplayer:main

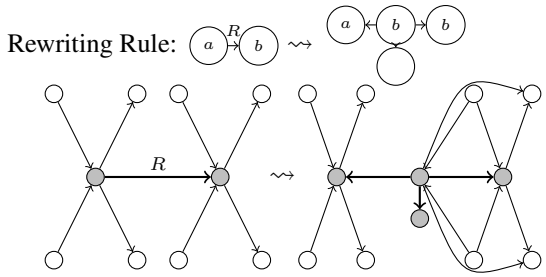

Figure 1: Rewriting rule and its application to a structure.

a partial function $s : \mathfrak{R} \to \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$. With each rule, we will also associate a set of relations symbols τ_{e} to be embedded exactly, as described below.

Definition 1. Let $\mathfrak{A}$ and $\mathfrak{B}$ be two relational structures over the same signature. A function $f : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is a τ_{e}-*embedding* if f is injective, for each $R_i \in \tau_{\mathrm{e}}$ it holds that

$$(a_1, \ldots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \ldots, f(a_{r_i})) \in R_i^{\mathfrak{B}},$$

and for $R_j \notin \tau_{\mathrm{e}}$ it holds that

$$(a_1, \ldots, a_{r_j}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \ldots, f(a_{r_j})) \in R_j^{\mathfrak{B}}.$$

A τ_{e}-*match* of the rule $\mathfrak{L} \to_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is a τ_{e}-embedding $\sigma : \mathfrak{L} \hookrightarrow \mathfrak{A}$.

Definition 2. The result of an application of $\mathfrak{L} \to_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ, denoted $\mathfrak{A}[\mathfrak{L} \to_s \mathfrak{R}/\sigma]$, is a relational structure $\mathfrak{B}$ with universe $(A \backslash \sigma(L)) \dot\cup R$, and relations given as follows. A tuple $(b_1, \ldots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{R}$ already, $(b_1, \ldots, b_{r_i}) \in R_i^{\mathfrak{R}}$, or there exists a tuple in the previous structure, $(a_1, \ldots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each j either $a_j = b_j$ or $a_j = \sigma(s(b_j))$, i.e. either the element was there before or it was matched and b_j is the replacement as specified by the rule. Moreover, if $R_i \in \tau_{\mathrm{e}}$ then we require in the second case that at least one b_l was already in the original structure, i.e. $b_l = a_l$ for some $l \in \{1, \ldots, r_i\}$.

To understand this definition it is best to consider an example, and one in which elements are both added and copied is presented in Figure 1. The labels a and b on the right-hand side of the rewriting rule depict the partial function s.

Logic and Constraints

The logic we use for specifying properties of states is an extension of first-order logic with real-valued terms and counting operators, cf. (Grädel and Gurevich 1998).

Syntax. We use first-order variables $x_1, x_2, \ldots$ ranging over elements of the structure, and we define formulas φ and real-valued terms ρ by the following grammar ($n, m \in \mathbb{N}$).

$$\begin{aligned} \varphi &:= R_i(x_1, \ldots, x_{r_i}) \mid x_i = x_j \mid \rho < \rho \\ &\quad \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi\,, \\ \rho &:= \frac{n}{m} \mid \rho + \rho \mid \rho \cdot \rho \mid \chi[\varphi] \mid \sum_{\overline{x}\mid\varphi} \rho\,. \end{aligned}$$

Semantics. Most of the above operators are defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of two real-valued terms, and $\exists x\, \varphi(x)$ means that there exists an element a in the universe such that $\varphi(a)$ holds. Among less known operators, the term $\chi[\varphi]$ denotes the characteristic function of φ, i.e. the real-valued term which is 1 for all assignments for which φ holds and 0 for all other. The term $\sum_{\overline{x}\mid\varphi} \rho$ denotes the sum of the values of $\rho(\overline{x})$ for all assignments of elements of the structure to $\overline{x}$ for which $\varphi(\overline{x})$ holds. Note that these terms can have free variables, e.g. the set of free variables of $\sum_{\overline{x}\mid\varphi} \rho$ is the union of free variables of φ and free variables of ρ, minus the set $\{\overline{x}\}$.

The logic defined above is used in structure rewriting rules in two ways. First, it is possible to define a new relation $R(\overline{x})$ using a formula $\varphi(\overline{x})$ with free variables contained in $\overline{x}$. Defined relations can be used on left-hand sides of structure rewriting rules, but are not allowed on right-hand sides. The second way is to add *constraints* to a rule. A rule $\mathfrak{L} \to_s \mathfrak{R}$ can be constrained using two sentences (i.e. formulas without free variables): φ_{pre} and φ_{post}. In φ_{pre} we allow additional constants l for each $l \in \mathfrak{L}$ and in φ_{post} special constants for each $r \in \mathfrak{R}$ can be used. A rule $\mathfrak{L} \to_s \mathfrak{R}$ with such constraints can be applied on a match σ in $\mathfrak{A}$ only if the following holds: At the beginning, the formula φ_{pre} must hold in $\mathfrak{A}$ with the constants l interpreted as $\sigma(l)$, and, after the rule is applied, the formula φ_{post} must hold in the resulting structure with each r interpreted as the newly added element r (cf. Definition 2).

Structure Rewriting Games

One can in principle describe a game simply by providing a set of allowed moves for each player. Still, in many cases it is natural to specify the control flow directly. For this reason, we define games as labeled graphs as follows.

Definition 3. A *structure rewriting game* with k players is a finite directed graph in which each vertex, called *location*, is assigned a player from $\{1, \ldots, k\}$ and k real-valued *payoff terms*, one for each player. Each edge of the graph represents a possible move and is labeled by a tuple

$$(\mathfrak{L} \to_s \mathfrak{R}, \tau_{\mathrm{e}}, \varphi_{\mathrm{pre}}, \varphi_{\mathrm{post}}),$$

which specifies the rewriting rule to be used with relations to be embedded and a pre- and post-condition. Multiple edges with different labels are possible between two locations.

Play Semantics. A play of a structure rewriting game starts in a fixed initial location of the game graph and in a state given by a starting structure. The moving player chooses an edge and a match allowed by the label of the edge such that it can be applied, i.e. both the pre- and the post-condition holds. The play proceeds to the location to which the edge leads and the new state is the structure after the application of the rule on the chosen match. If in some location and state it is not possible to apply any of the rules on the outgoing edges, either because no match can be found or because of the constraints, then the play ends. Payoff terms from that location are evaluated on the state and determine the outcome of the game for all players.

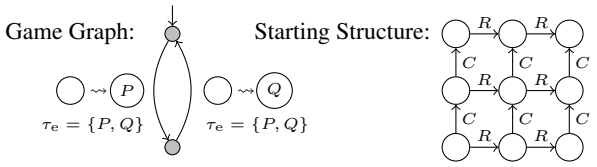

Figure 2: Tic-tac-toe as a structure rewriting game.

Example 4. Let us define Tic-tac-toe in our formalism. The starting structure has 9 elements connected by binary row and column relations, R and C, as depicted on the right in Figure 2. To mark the moves of the players we use unary relations P and Q, representing crosses and circles. The allowed move of the first player is to mark any unmarked element with P and the second player can mark with Q. Thus, there are two locations (gray) in the game graph, representing which player's turn it is, and two corresponding rules, both with one element on each side (left in Figure 2).

Observe that in both rules we require P and Q to be embedded. Note (see Definitions 1 and 2) that this ensures that no player can make a move where someone has already moved before. The two diagonals can be defined by

$$D_A(x,y) = \exists z(R(x,z) \wedge C(z,y)),$$
$$D_B(x,y) = \exists z(R(x,z) \wedge C(y,z))$$

and a line of three by

$$L(x,y,z) = (R(x,y) \wedge R(y,z)) \vee (C(x,y) \wedge C(y,z)) \vee (D_A(x,y) \wedge D_A(y,z)) \vee (D_B(x,y) \wedge D_B(y,z)).$$

Using these definitions, we specify the goal of the first player

$$\varphi = \exists x,y,z\,\big(P(x) \wedge P(y) \wedge P(z) \wedge L(x,y,z)\big)$$

and the goal of the other player by an analogous formula φ' in which P is replaced by Q. The payoff terms are $\chi[\varphi] - \chi[\varphi']$ for the first player and $\chi[\varphi'] - \chi[\varphi]$ for the other, and to ensure that the game ends when one of the players has won, we take as a precondition of each move the negation of the goal formula of the other player.

The complete code for Tic-tac-toe, with the starting structure in a special syntax we use for grids with row and column relations, is given in Figure 3. Note that we write `:(`φ`)` for $\chi[\varphi]$ and e.g. `Q:1 {}` for an empty relation Q of arity 1. Please refer to (Toss) for the complete input syntax and definitions of other, more complex games, e.g. Chess.

Type Normal Form

To derive evaluation heuristics from payoff terms, we first have to introduce a normal form of formulas which we exploit later in the construction. This normal form is in a sense a converse to the prenex normal form (PNF), because the quantifiers are pushed as deep inside the formula as possible. A very similar normal form has been used recently in a different context (Ganzow and Kaiser 2010). For a set of formulas Φ let us denote by $\mathcal{B}^+(\Phi)$ all positive Boolean combinations of formulas from Φ, i.e. define $\mathcal{B}^+(\Phi) = \Phi \mid \mathcal{B}^+(\Phi) \vee \mathcal{B}^+(\Phi) \mid \mathcal{B}^+(\Phi) \wedge \mathcal{B}^+(\Phi)$.

```
PLAYERS X, O
REL DgA(x, y) = ex z (R(x, z) and C(z, y))
REL DgB(x, y) = ex z (R(x, z) and C(y, z))
REL Row3(x, y, z) = R(x, y) and R(y, z)
REL Col3(x, y, z) = C(x, y) and C(y, z)
REL DgA3(x, y, z) = DgA(x, y) and DgA(y, z)
REL DgB3(x, y, z) = DgB(x, y) and DgB(y, z)
REL Conn3(x, y, z) =
  Row3(x, y, z) or Col3(x, y, z) or
  DgA3(x, y, z) or DgB3(x, y, z)
REL WinP()= ex x,y,z(P(x) and P(y) and P(z)
                     and Conn3(x, y, z))
REL WinQ()= ex x,y,z(Q(x) and Q(y) and Q(z)
                     and Conn3(x, y, z))
RULE Cross:
  [a | - | - ] -> [a | P(a) | - ]
  emb P, Q pre not WinQ()
RULE Circle:
  [a | - | - ] -> [a | Q(a) | - ]
  emb P, Q pre not WinP()
LOC 0 { PLAYER X
  PAYOFF {
    X: :(WinP()) - :(WinQ());
    O: :(WinQ()) - :(WinP())
  }
  MOVES [Cross -> 1] }
LOC 1 { PLAYER O
  PAYOFF {
    X: :(WinP()) - :(WinQ());
    O: :(WinQ()) - :(WinP())
  }
  MOVES [Circle -> 0] }
MODEL [ | P:1 {}; Q:1 {} | ] "

        .  .  .

        .  .  .

        .  .  .
"
```

Figure 3: Tic-tac-toe in our game description formalism.

Definition 5. A formula is in TNF if and only if it is a positive Boolean combination of formulas of the following form

$$\tau = R_i(\overline{x}) \mid \neg R_i(\overline{x}) \mid x = y \mid x \neq y \mid \exists x \mathcal{B}^+(\tau) \mid \forall x \mathcal{B}^+(\tau)$$

satisfying the following crucial constraint: in $\exists x\, \mathcal{B}^+(\{\tau_i\})$ and $\forall x\, \mathcal{B}^+(\{\tau_i\})$ the free variables of *each* τ_i appearing in the Boolean combination *must contain* x.

We claim that for each formula φ there exists an equivalent formula ψ in TNF, and the procedure `TNF`(φ) computes ψ given φ in negation normal form. Note that it uses sub-procedures `DNF` and `CNF` which, given a Boolean combination of formulas, convert it to disjunctive or respectively conjunctive normal form.

As an example, consider $\varphi = \exists x\big(P(x) \wedge (Q(y) \vee R(x))\big)$; This formula is not in TNF as $Q(y)$ appears under $\exists x$, and

$\texttt{TNF}(\varphi) = \big(Q(y) \wedge \exists x P(x)\big) \vee \exists x \big(P(x) \wedge R(x)\big)$.

Procedure TNF (φ)

case φ *is a literal* **return** φ;
case $\varphi = \varphi_1 \vee \varphi_2$ **return** $\texttt{TNF}(\varphi_1) \vee \texttt{TNF}(\varphi_2)$;
case $\varphi = \varphi_1 \wedge \varphi_2$ **return** $\texttt{TNF}(\varphi_1) \wedge \texttt{TNF}(\varphi_2)$;
case $\varphi = \exists x\, \psi$
  Let $\texttt{DNF}(\texttt{TNF}(\psi)) = \bigvee_i (\bigwedge_j \psi^i_j)$
  and $F_i = \{j \mid x \in \text{FreeVar}(\psi^i_j)\}$;
  return $\bigvee_i \Big(\bigwedge_{j \notin F_i} \psi^i_j \wedge \exists x (\bigwedge_{j \in F_i} \psi^i_j) \Big)$;
case $\varphi = \forall x\, \psi$
  Let $\texttt{CNF}(\texttt{TNF}(\psi)) = \bigwedge_i (\bigvee_j \psi^i_j)$
  and $F_i = \{j \mid x \in \text{FreeVar}(\psi^i_j)\}$;
  return $\bigwedge_i \Big(\bigvee_{j \notin F_i} \psi^i_j \vee \forall x (\bigvee_{j \in F_i} \psi^i_j) \Big)$;

Theorem 6. $\texttt{TNF}(\varphi)$ *is equivalent to* φ *and in TNF.*

The proof of the above theorem is a simple argument by induction on the structure of the formula, so we omit it here. Instead, let us give an example which explains why it is useful to compute TNF for the goal formulas.

Example 7. As already defined above, the payoff in Tic-tac-toe is given by $\exists x, y, z (P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z))$. To simplify this example, let us consider the payoff given only by row and column triples, i.e.

$$\varphi = \exists x, y, z \big(P(x) \wedge P(y) \wedge P(z) \wedge \big((R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)) \big) \big).$$

This formula is not in TNF and the DNF of the quantified part has the form $\varphi_1 \vee \varphi_2$, where

$$\varphi_1 = P(x) \wedge P(y) \wedge P(z) \wedge R(x, y) \wedge R(y, z),$$
$$\varphi_2 = P(x) \wedge P(y) \wedge P(z) \wedge C(x, y) \wedge C(y, z).$$

The procedure TNF must now choose the variable to first split on (this is discussed in the next section) and pushes the quantifiers inside, resulting in $\texttt{TNF}(\varphi) = \psi_1 \vee \psi_2$ with

$$\psi_1 = \exists x \big(P(x) \wedge \exists y \big(P(y) \wedge R(x, y) \wedge \exists z (P(z) \wedge R(y, z)) \big) \big),$$
$$\psi_2 = \exists x \big(P(x) \wedge \exists y \big(P(y) \wedge C(x, y) \wedge \exists z (P(z) \wedge C(y, z)) \big) \big).$$

In spirit, the TNF formula is thus more "step-by-step" than the goal formula we started with, and we exploit this to generate heuristics for evaluating positions below.

Heuristics from Existential Formulas

In this section, we present one method to generate a heuristic from an existential goal formula. As a first important step, we divide all relations appearing in the signature in our game into two sorts, *fluents* and *stable relations*. A relation is called *stable* if it is not changed by any of the structure rewriting rules which appear as possible moves, all other relations are *fluent*. We detect stable relations by a simple syntactic analysis of structure rewriting rules, i.e. we check which relations from the left-hand side remain unchanged on the right-hand side of the rule. It is a big advantage of our formalism in comparison to GDL that stable relations (such as row and column relations used to represent the board) can so easily be separated from the fluents.

After detecting the fluents, our first step in generating the heuristic is to compute the TNF of the goal formula. As mentioned in the example above, there is certain freedom in the TNF procedure as to which quantified variable is to be resolved first. We use fluents to decide this — a variable which appears in a fluent will be resolved before all other variables which do not appear in any fluent literal (we choose arbitrarily in the remaining cases).

After the TNF has been computed, we change each sequence of existential quantifiers over conjunctions into a sum, counting how many steps towards satisfying the whole conjunction have been made. Let us fix a factor $\alpha < 1$ which we will discuss later. Our algorithm then changes a formula in the following way.

$$\exists x_1 \big(\vartheta_1(x_1) \wedge \exists x_2 \big(\vartheta_1(x_2, x_1) \wedge \cdots \wedge \exists x_n \big(\vartheta_n(x_n, \overline{x_i}) \big) \cdots \big) \big)$$
$$\wr$$
$$\sum_{x_1 | \vartheta_1(x_1)} \big(\alpha^{n-1} + \sum_{x_2 | \vartheta_2(x_2, x_1)} (\alpha^{n-2} + \cdots (\alpha + \sum_{x_n | \vartheta_n(x_n, \overline{x_i})} 1) \cdots) \big)$$

The sub-formulas $\vartheta_i(x_i, \overline{x})$ are in this case conjunctions of literals or formulas which contain universal quantifiers. The factor α defines how much more making each next step is valued over the previous one. When a formula contains disjunctions, we use the above schema recursively and sum the terms generated for each disjunct.

To compute a heuristic for evaluating positions from a payoff term, which is a real-valued expression in the logic defined above, we substitute all characteristic functions, i.e. expressions of the form $\chi[\varphi]$, by the sums generated for φ as described above.

Example 8. Consider the TNF of the simplified goal formula for Tic-tac-toe presented in the previous example and let $\alpha = \frac{1}{4}$. Since the TNF of the goal formula for one player has the form $\psi_1 \vee \psi_2$, we generate the following sums:

$$s_1 = \sum_{x | P(x)} \big(\frac{1}{8} + \sum_{y | P(y) \wedge R(x, y)} \big(\frac{1}{4} + \sum_{z | P(z) \wedge R(y, z)} 1 \big) \big),$$
$$s_2 = \sum_{x | P(x)} \big(\frac{1}{8} + \sum_{y | P(y) \wedge C(x, y)} \big(\frac{1}{4} + \sum_{z | P(z) \wedge C(y, z)} 1 \big) \big).$$

Since the payoff is defined by $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, i.e. with Q in place of P, the total generated heuristic has the form

$$s_1 + s_2 - s'_1 - s'_2,$$

where s'_1 and s'_2 are as s_1 and s_2 but with P replaced by Q.

Finding Existential Descriptions

The method described above is effective if the TNF of the goal formulas has a rich structure of existential quantifiers. But this is not always the case, e.g. in Breakthrough the goal formula for white has the form $\exists x \, (W(x) \wedge \neg \exists y \, C(x, y))$, because $\neg \exists y \, C(x, y)$ describes the last row which the player

is supposed to reach. The general question which presents itself in this case is how, given an arbitrary relation $R(\overline{x})$ (as the last row above), can one construct an existential formula describing this relation. In this section, we present one method which turned out to yield useful formulas at least for common board games.

First of all, let us remark that the construction we present will be done only for relations defined by formulas which do not contain fluents. Thus, we can assume that the relation does not change during the game and we use the starting structure in the construction of the existential formula.

Our construction keeps a set C of conjunctions of stable literals. We say that a subset $\{\varphi_1, \ldots, \varphi_n\} \subseteq C$ *describes* a relation $Q(\overline{x})$ in $\mathfrak{A}$ if and only if Q is equivalent in $\mathfrak{A}$ to the existentially quantified disjunction of φ_i's, i.e. if

$$\mathfrak{A} \models Q(\overline{x}) \iff \mathfrak{A} \models \bigvee_i (\exists \overline{y_i}\, \varphi_i),$$

where $\overline{y_i}$ are all free variables of φ_i except for $\overline{x}$.

Our procedure extends the conjunctions from C with new literals until a subset which describes Q is found. These extensions can in principle be done in any order, but to obtain compact descriptions in reasonable time we perform them in a greedy fashion. The conjunctions are ordered by their hit-rank, defined as

$$\texttt{hit-rank}_{\mathfrak{A},Q(\overline{x})}(\varphi) = \frac{|\{\overline{x} \in Q \mid \mathfrak{A} \models \exists \overline{y}\, \varphi(\overline{x})\}|}{|\{\overline{x} \mid \mathfrak{A} \models \exists \overline{y}\, \varphi(\overline{x})\}|},$$

where again $\overline{y} = \texttt{FreeVar}(\varphi) \setminus \overline{x}$. Intuitively, the hit-rank is the ratio of the tuples from Q which satisfy (existentially quantified) φ to the number of all such tuples. Thus, the hit-rank is 1 if φ describes Q and we set the hit-rank to 0 if φ is not satisfiable in $\mathfrak{A}$. We define the $\texttt{rank}_{\mathfrak{A},Q}(\varphi, R(\overline{y}))$ as the maximum of the $\texttt{hit-rank}_{\mathfrak{A},Q}(\varphi \wedge R(\overline{y}))$ and the $\texttt{hit-rank}_{\mathfrak{A},Q}(\varphi \wedge \neg R(\overline{y}))$. The complete procedure is summarized below.

Procedure `ExistentialDescription`$(\mathfrak{A}, Q)$

$C \longleftarrow \{\top\}$
while *no subset of* C *describes* $Q(\overline{x})$ *in* $\mathfrak{A}$ **do**
  for *a stable relation* $R(\overline{y})$*, conjunction* $\varphi \in C$ *with maximal* $\texttt{rank}_{\mathfrak{A},Q}(\varphi, R(\overline{y}))$ **do**
    $C \longleftarrow (C \setminus \{\varphi\}) \cup \{\varphi \wedge R(\overline{y}),\ \varphi \wedge \neg R(\overline{y})\}$
  end
end

Since it is not always possible to find an existential description of a relation, let us remark that we stop the procedure if no description with a fixed number of literals is found. We also use a tree-like data structure for C to check the existence of a describing subset efficiently.

Example 9. As mentioned before, the last row on the board is defined by the relation $Q(x) = \neg\exists y\, C(x, y)$. Assume that we search for an existential description of this relation on a board with only the binary row and column relations (R and C) being stable, as in Figure 2. Since adding a row literal will not change the hit-rank, our construction will be adding column literals one after another and will finally arrive, on a 3×3 board, at the following existential description: $\exists y_1, y_2(C(y_1, y_2) \wedge C(y_2, x))$. Using such formula, the heuristic constructed in the previous section can count the number of steps needed to reach the last row for each pawn, which is important e.g. in Breakthrough.

Alternative Heuristics with Rule Conditions

The algorithm presented above is only one method to derive heuristics, and it uses only the payoff terms. In this section we present an alternative method, which is simpler and uses also the rewriting rules and their constraints. This simpler technique yields good heuristics only for games in which moves are monotone and relatively free, e.g. for Connect5.

Existential formulas are again the preferred input for the procedure, but this time we put them in prenex normal form at the start. As before, all universally quantified formulas are either treated as atomic relations or expanded, as discussed above. The Boolean combination under the existential quantifiers is then put in DNF and, in each conjunction in the DNF, we separate fluents from stable relations. After such preprocessing, the formula has the following form:

$$\exists \overline{x}\, \big((\vartheta_1(\overline{x}) \wedge \psi_1(\overline{x})) \vee \cdots \vee (\vartheta_n(\overline{x}) \wedge \psi_n(\overline{x}))\big),$$

where each $\vartheta_i(\overline{x})$ is a conjunction of fluents and each $\psi_i(\overline{x})$ is a conjunction of stable literals.

To construct the heuristic, we will retain the stable subformulas $\psi_i(\overline{x})$ but change the fluent ones $\vartheta_i(\overline{x})$ from conjunctions to sums. Formally, if $\vartheta_i(\overline{x}) = F_1(\overline{x}) \wedge \cdots \wedge F_k(\overline{x})$ then we define $s_i(\overline{x}) = \chi[F_1(\overline{x})] + \cdots + \chi[F_k(\overline{x})]$, and let $\delta_i(\overline{x}) = F_1(\overline{x}) \vee \cdots \vee F_k(\overline{x})$ be a formula checking if the sum $s_i(\overline{x}) > 0$. The guard for our heuristic is defined as

$$\gamma(\overline{x}) = \big(\psi_1(\overline{x}) \vee \cdots \vee \psi_n(\overline{x})\big) \wedge \big(\delta_1(\overline{x}) \vee \cdots \vee \delta_n(\overline{x})\big)$$

and the heuristic with parameter m by

$$\sum_{\overline{x} \mid \gamma(\overline{x}) \wedge \texttt{move}(\overline{x})} \big(s_1(\overline{x}) + \cdots + s_n(\overline{x})\big)^m.$$

The additional formula $\texttt{move}(\overline{x})$ is used to guarantee that at each element matched to one of the variables $\overline{x}$ it is still possible to make a move. This is done by converting the rewrite rule into a formula with free variables corresponding to the elements of the left-hand side structure, removing all the fluents F_i from above if these appear negated, and quantifying existentially if a new variable (not in $\overline{x}$) is created in the process. The following example shows how the procedure is applied for Tic-tac-toe.

Example 10. For Tic-tac-toe simplified as before (no diagonals), the goal formula in PNF and DNF reads:

$$\exists x, y, z\, \big((P(x) \wedge P(y) \wedge P(z) \wedge R(x, y) \wedge R(y, z)) \vee (P(x) \wedge P(y) \wedge P(z) \wedge C(x, y) \wedge C(y, z))\big).$$

The resulting guard is thus, after simplification,

$$\gamma(x, y, z) = \big((R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z))\big) \wedge \big(P(x) \vee P(y) \vee P(z)\big).$$

Since the structure rewriting rule for the move has only one element, say u, on its left-hand side, and $\tau_e = \{P, Q\}$ for this rule, the formula for the left-hand side reads $l(u) = \neg P(u) \wedge \neg Q(u)$. Because P appears as a fluent in γ we remove all occurrences of $\neg P$ from l and are then left with $\texttt{move}(u) = \neg Q(u)$. Since we require that a move is possible from all variables, the derived heuristic for one player with power 4 has the form

$$h = \sum_{x,y,z \mid \gamma(x,y,z) \wedge \neg Q(x) \wedge \neg Q(y) \wedge \neg Q(z)} (\chi[P(x)] + \chi[P(y)] + \chi[P(z)])^4.$$

Since the payoff expression is $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, we use $h - h'$ as the final heuristic to evaluate positions.

Experimental Results

The described algorithms are a part of (Toss), an open-source program implementing various logic functions as well as the presented game model and a GUI for the players. Toss contains an efficient implementation of `CNF` and `DNF` conversions and formula simplifications (interfacing a SAT solver, MiniSAT), and a model-checker for FO, which made it a good platform for our purpose.

We defined several board games in our formalism and created move translation scripts to play them against a GGP agent, Fluxplayer (Schiffel and Thielscher 2007). The tests were performed on the Dresden GGP Server[2].

We played 4 different games, 20 plays each, to illustrate how our heuristics work in games of various kinds. In these plays, Toss uses a basic constant-depth alpha-beta search algorithm. For the results in Table 1, Toss was using heuristics generated with parameter $\alpha = \frac{1}{4}$ or $m = 4$ in the alternative case, and the choice of the heuristic was made based on branching – the alternative one was used for Connect5.

	Toss Wins	Fluxplayer Wins	Tie
Breakthrough	95%	5%	0%
Connect4	20%	75%	5%
Connect5	0%	0%	100%
Pawn Whopping	50%	50%	0%
Total	41.25%	32.5%	26.25%

Table 1: Toss against Fluxplayer on 4 sample games.

The final total score was 4350 vs. 3650 for Toss, but, as you can see, it strongly depends on the game played. One important problem in games with low branching, i.e. both in Connect4 and Pawn Whopping, is that all leaves in the search tree of the basic alpha-beta have the same height. Fluxplayer uses a more adaptive search algorithm, and in tests against Toss with a variable-depth search, Toss did not lose any game and the final score was 5350 vs. 2650.

[2]Toss svn release 1315 was used to play on `euklid.inf.tu-dresden.de:8180/ggpserver/`. The plays are in 4 tournaments, one for each tested game, and can be viewed online after selecting `Toss` under `Players`. The variable-depth search plays are presented in 4 additional tournaments, with "variable_depth" suffix, and this algorithm is used in Toss release 0.6.

Perspectives

We presented a formalism for describing perfect information games and an algorithm to derive from this formalism heuristics for evaluating positions in the game. These are good enough to win against a state of the art GGP player and thus prove the utility of the introduced formalism.

The use of counting FO formulas for goals and move constraints allows to derive many more heuristics than the two we presented. Probably, a weighted combination of heuristics constructed in various ways from both the payoffs and the constraints would outperform any single one by far. One of the problems with this approach is the necessity to automatically learn the weights, which on the one hand requires a lot of time, but on the other hand promises an agent which improves with each play. One technique which we would like to try to apply in this context is boosting, as in (Freund and Schapire 1995), with heuristics used as experts.

Since it is very easy to construct new interesting patterns when operating on formulas, we also plan to explore how such formulas can be used to prune the search space. We imagine for example a formula which dictates the use of strategies which only place a new stone in a distance of at most 3 from some already present one. Such pruning may not be optimal in itself, but it may be the only way to perform a deeper search in games with large branching, and maybe it can be combined with some form of broad shallow search. Generally, the use of counting FO formulas facilitates many tasks and opens many new possibilities for GGP.

References

Clune, J. E. 2008. *Heuristic Evaluation Functions for General Game Playing*. Dissertation, UCLA.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI'08*. AAAI Press.

Freund, Y., and Schapire, R. E. 1995. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT'95*, volume 904 of *LNCS*, 23–37.

Ganzow, T., and Kaiser, Ł. 2010. New algorithm for weak monadic second-order logic on inductive structures. In *CSL'10*, volume 6247 of *LNCS*, 366–380. Springer.

Genesereth, M. R., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Grädel, E., and Gurevich, Y. 1998. Metafinite model theory. *Information and Computation* 140:26–81.

Kaiser, Ł. 2009. Synthesis for structure rewriting systems. In *MFCS'09*, volume 5734 of *LNCS*, 415–427. Springer.

Rajlich, V. 1973. Relational structures and dynamics of certain discrete systems. In *MFCS'73*, 285–292.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *AAAI'07*, 1191–1196. AAAI Press.

Thielscher, M. 2010. A general game description language for incomplete information games. In *AAAI'10*, 994–999. AAAI Press.

Toss. `http://toss.sourceforge.net`.