Playing General Structure Rewriting Games

Łukasz Kaiser
Mathematische Grundlagen der Informatik
RWTH Aachen

Łukasz Stafiniak
Institute of Computer Science
University of Wrocław

Abstract

Achieving goals in a complex environment in which many players interact is a general task demanded from an AI agent. When goals of the players are given explicitly, such setting can be described as a multi-player game with complete information. We introduce a general model of such games in which states are represented by relational structures (hypergraphs), possibly with real-valued labels, and actions by structure rewriting rules. For this model, we develop an algorithm which computes rational strategies for the players. Our algorithm can be parametrized by a probabilistic evaluation function and we devise a general procedure for learning such evaluations. First tests on a few classical examples substantiate the chosen game model and our algorithm.

Introduction

As frustrated users know, a computer sometimes simply does not *want* to work. At other times, a car does not *want* to start. These phrases show how natural it is to ascribe *desires* and *goals* to active objects in an effort to understand them. Not only is it natural: it is arguably very useful as well. We encounter many complex objects in the environment, including ourselves and other people, and it is impossible to understand their function in detail. Knowing their goals and intentions, even vaguely, already allows to predict their actions to a useful degree.

In this paper, we present a modeling system in which explicitly given goals of multiple players define the dynamics. To run such a system, an algorithm making rational decisions for the players is necessary. Our main contribution is exactly such an algorithm, which gives reasonable results by default and can take probabilistic evaluation functions as an additional parameter. We also devise a general learning mechanism to construct evaluation functions. This mechanism is non-deterministic: the choice of the next hypothesis is delegated to an external function. Still, even instantiated with a very simple policy it manages to learn useful evaluations, at least in a few basic examples. Some components used in the presented algorithms may be of independent interest. One of them is a solver for an expressive logic, another one is a generalization of the Monte-Carlo playing algorithm with Upper Confidence bounds for Trees (UCT).

The UCT algorithm has already been used in a general game playing (GGP) competition. Cadia player [2], a program using UCT, won the competition in 2007 demonstrating good performance of the UCT algorithm. Sadly, both the way of representing games in the GGP framework and the examples used in the competition lack true generality: These are either board games (e.g. connect-4, chess, go) or maze games (e.g. pac-man) described in a very basic prolog-like language [4]. There is neither a way to represent continuous real-time dynamics in the GGP framework, nor a way to define probabilistic choice. Moreover, in almost all examples it is possible to distinguish a fixed board and pieces moved by the players. Thus, for programs entering the GGP competition certain narrow board-game heuristics are crucial, which reduces their applications to AGI. We give both a general game model, representing states in a way similar to generalized hypergraphs [5] already used for AGI in OpenCog, and a general algorithm which is capable to learn useful patterns in specific situations without any fixed prior heuristic.

Organization. In the first two sections, we show how to represent the state of the world in which the agents play and their actions. The first section discusses discrete dynamics and the second one specifies how continuous values evolve. In the third section we introduce the logic used to describe patterns in the states. We finalize the definition of our model of games in the fourth section. Next, we proceed to the algorithmic part: we first describe the generalized UCT algorithm for playing games and then the learning procedure for evaluation functions. Finally, we present a few experimental results and conclude with perspectives on the applications of our modeling system and algorithms in AGI projects.

Discrete Structure Rewriting

To represent a state of our model in a fixed moment of time we use finite relational structures, i.e. labelled directed hypergraphs. A relational structure $\mathfrak{A} = (A, R_1, \ldots, R_k)$ is composed of a universe A and a number of relations $R_1, \ldots, R_k$. We denote the arity of R_i by 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 dynamics of the model, i.e. the way the structure can change, is described by *structure rewriting rules*, a generalized form of term and graph rewriting. Extended graph rewriting is recently viewed as the programming model of choice for complex multi-agent systems, especially ones with real-valued components [1]. Moreover, this form of rewriting is well suited for visual programming and helps to

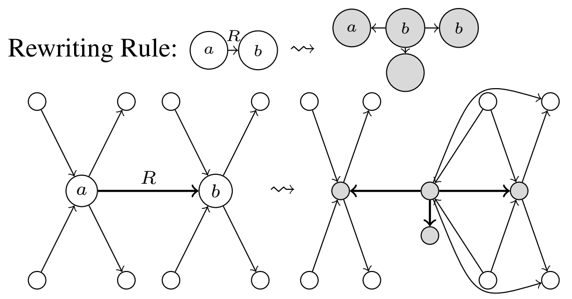

Figure 1: Rewriting rule and its application to a structure.

make the systems understandable.

In the most basic setting, a rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ consists of two finite relational structures $\mathfrak{L}$ and $\mathfrak{R}$ over the same signature and a partial function $s : \mathfrak{R} \rightarrow \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$.

Let $\mathfrak{A}, \mathfrak{B}$ be two structures, τ_{e} a set of relations symbols to be matched exactly and τ_{h} a set of relations to be matched only positively.[1] A function $f : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is a $(\tau_{\mathrm{e}}, \tau_{\mathrm{h}})$-*embedding* if f is injective, for each $R_i \in \tau_{\mathrm{e}}$ it holds that $(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \dots, f(a_{r_i})) \in R_i^{\mathfrak{B}}$, and for $R_j \in \tau_{\mathrm{h}}$ it holds that $(a_1, \dots, a_{r_i}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \dots, f(a_{r_i})) \in R_j^{\mathfrak{B}}$. A $(\tau_{\mathrm{e}}, \tau_{\mathrm{h}})$-*match* of the rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is an $(\tau_{\mathrm{e}}, \tau_{\mathrm{h}})$-embedding $\sigma : \mathfrak{L} \hookrightarrow \mathfrak{A}$. We define the result of an application of $\mathfrak{L} \rightarrow_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ as $\mathfrak{B} = \mathfrak{A}[\mathfrak{L} \rightarrow_s \mathfrak{R}/\sigma]$, such that the universe of $\mathfrak{B}$ is given by $(A \setminus \sigma(L)) \dot{\cup} R$, and the relations as follows. A tuple $(b_1, \dots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{R}$ already, $(b_1, \dots, b_{r_i}) \in R_i^{\mathfrak{R}}$, or there exists a tuple in the previous structure, $(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each i either $a_i = b_i$ or $a_i = \sigma(s(b_i))$, i.e. either the element was there before or it was matched and b_i 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_i was already in the original structure, i.e. $b_i = a_i$. To understand this definition it is best to consider an example, and one is given in Figure 1.

Continuous Evolution

To model continuous dynamics in our system, we supplement relational structures with a number of labeling functions $f_1, \dots, f_l$, each $f_i : A \rightarrow \mathbb{R}$ (A is the universe).[2] Accordingly, each rewriting rule is extended by a system of ordinary differential equations (ODEs) and a set of right-hand update equations. We use a standard form of ODEs: $f_{i,l}^k = t(f_{i,l}^0, \dots, f_{i,l}^{k-1})$, where f_i are the above-mentioned functions, l can be any element of the left-hand side structure and f^k denotes the k-th derivative of f. The term $t(\overline{x})$ is constructed using standard arithmetic functions $+, -, \cdot, /$, natural roots $\sqrt[n]{\ }$ for $n > 1$ and rational numbers $r \in \mathbb{Q}$ in addition to the variables $\overline{x}$ and a set of parameters $\overline{p}$ fixed for each rule. The set of right-hand side update equations contains one equation of the form $f_{i,r} = t(\overline{f_{i,l}})$ for each function f_i and each r from the right-hand side structure.

Let $\mathcal{R} = \{(\mathfrak{L}_i \rightarrow_{s_i} \mathfrak{R}_i, \mathcal{D}_i, \mathcal{T}_i) \mid i < n\}$ be a set of rules extended with ODEs $\mathcal{D}_i$ and update equations $\mathcal{T}_i$ as described above. Given, for each rule in $\mathcal{R}$, a match σ_i of the rule in a structure $\mathfrak{A}$, the required parameters $\overline{p_i}$ and a time bound t_i, we define the result of a *simultaneous application* of $\mathcal{R}$ to $\mathfrak{A}$, denoted $\mathfrak{A}[\mathcal{R}/\{\sigma_i, t_i\}]$, as follows.[3]

First, the structure $\mathfrak{A}$ evolves in a continuous way as given by the *sum* of all equations $\mathcal{D}_i$. More precisely, let $\mathcal{D}$ be a system of differential equations where for each $a \in \mathfrak{A}$ there exists an equation defining $f_{i,a}^k$ if and only if there exists an equation in some $\mathcal{D}_j$ for $f_{i,l}^k$ for some l with $\sigma_j(l) = a$. In such case, the term for $f_{i,a}^k$ is the sum of all terms for such l, with each $f_{i,l}^m$ replaced by the appropriate $f_{i,\sigma_j(l)}^m$. Assuming that all functions f_i and all their derivatives are given at the beginning, there is a unique solution for these variables which satisfies $\mathcal{D}$ and has all other, undefined derivatives set by the starting condition from $\mathfrak{A}$. This solution defines the value of $f_{i,a}(t)$ for each $a \in \mathfrak{A}$ at any time moment t.

Let $t^0 = \min_{i<n} t_i$ be the lowest chosen time bound and let $i_0, \dots, i_k$ be all rules with this bound, i.e. each $t_{i_m} = t^0$. We apply each of these rules independently[1] to the structure $\mathfrak{A}$ at time t^0. Formally, the relational part of $\mathfrak{A}[\mathcal{R}/\{\sigma_i, t_i\}]$ is equal to $\mathfrak{A}[\mathfrak{L}_{i_0} \rightarrow_{s_{i_0}} \mathfrak{R}_{i_0}/\sigma_{i_0}] \cdots [\mathfrak{L}_{i_k} \rightarrow_{s_{i_k}} \mathfrak{R}_{i_k}/\sigma_{i_k}]$ and the function values $f_i(a)$ are defined as follows. If the element a was not changed, $a \in \mathfrak{A}$, then we keep the function value from the solution of $\mathcal{D}$, i.e. $f_i(a) = f_{i,a}(t^0)$. In the other case a was on the right-hand side of some rule, $a \in \mathfrak{R}_m$. Let $f_{i,a} = t(\overline{f_{j,l}})$ be the equation in $\mathcal{T}_m$ defining $f_{i,a}$. The new value of $f_i(a)$ is then computed by inserting the appropriate values for $f_{j,l}$ from the solution of $\mathcal{D}$ into $t(\overline{f_{j,l}})$, i.e. $f_i(a) = t(\overline{y_{j,l}})$ where each $y_{j,l} = f_{j,\sigma_m(l)}(t^0)$.

Example. Let us define a simple two-dimensional model of a cat chasing a mouse. The structure we use, $\mathfrak{A} = (\{c, m\}, C, M, x, y)$, has two elements c and m, unary relations $C = \{c\}$ and $M = \{m\}$ used to identify which element is which and two real-valued functions x and y. Both rewriting rules have only one element, both on the left-hand side and on the right-hand side, and the element is in C for the cat rule and in M for the mouse rule. The ODEs for both rules are of the form $x' = p_x, y' = p_y$, where p_x, p_y are parameters. The update equations just keep the left-hand side values, $x_r = x_l, y_r = y_l$. In this setting, simultaneous application of the cat rule with parameters p_x^c, p_y^c for time t^c and the mouse rule with parameters p_x^m, p_y^m for time t^m will have the following effect: The cat will move with speed p_x^c along the x-axis and p_y^c along the y-axis and the mouse analogously with p_x^m and p_y^m, both for time $t^0 = \min(t^c, t^m)$.

Logic and Constraints

The logic we use for specifying properties of states is an extension of monadic second-order logic with real-valued terms and counting operators. The main motivation for the

[1]In practice, we also allow some tuples in $\mathfrak{L}$ to be optional; this is a shorthand for multiple rules with the same right-hand side.

[2]In fact $f_i(a)$ is not in $\mathbb{R}$; it is a function $\varepsilon \rightarrow (x, x+\delta), \delta < \varepsilon$.

[3]Assume no two intersecting rules have identical time bounds.

choice of such logic is *compositionality*: To evaluate a formula on a large structure $\mathfrak{A}$ which is composed in a regular way from substructures $\mathfrak{B}$ and $\mathfrak{C}$ it is enough to evaluate certain formulas on $\mathfrak{B}$ and $\mathfrak{C}$ independently. Monadic second-order logic is one of the most expressive logics with this property and allows to define various useful patterns such as stars, connected components or acyclic subgraphs.[4]

In the syntax of our logic, we use first-order variables $(x_1, x_2, \ldots)$ ranging over elements of the structure, second-order variables $(X_1, X_2, \ldots)$ ranging over *sets* of elements, and real-valued variables $(\alpha_1, \alpha_2, \ldots)$ ranging over $\mathbb{R}$, and we distinguish boolean formulas φ and real-valued terms ρ:

$$\varphi := R_i(x_1, \ldots, x_{r_i}) \mid x_i = x_j \mid x_i \in X_j \mid \rho <_\varepsilon \rho \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \neg\varphi \mid \exists x_i \varphi \mid \forall x_i \varphi \mid \exists X_i \varphi \mid \forall X_i \varphi \mid \exists \alpha_i \varphi \mid \forall \alpha_i \varphi,$$

$$\rho := \alpha_i \mid f_i(x_j) \mid \rho \dotplus \rho \mid \chi[\varphi] \mid \min_{\alpha_i} \varphi \mid \textstyle\sum_{\overline{x}|\varphi} \rho \mid \prod_{\overline{x}|\varphi} \rho.$$

Semantics of most of the above operators is defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of real-valued terms, and $\exists X \varphi(X)$ means that there exists a set of elements S such that $\varphi(S)$ 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. To evaluate $\min_{\alpha_i} \varphi$ we take the minimal α_i for which φ holds (we allow $\pm\infty$ as values of terms as well). The terms $\sum_{\overline{x}|\varphi} \rho$ and $\prod_{\overline{x}|\varphi} \rho$ denote the sum and product 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 both these terms can have free variables, e.g. the set of free variables of $\sum_{\overline{x}|\varphi} \rho$ is the union of free variables of φ and free variables of ρ, minus the set $\{\overline{x}\}$. Observe also the ε in $<_\varepsilon$: the values $f(a)$ are given with arbitrary small but non-zero error (cf. footnote 2) and $\rho_1 <_\varepsilon \rho_2$ holds only if the upper bound of ρ_1 lies below the lower bound of ρ_2.

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 three sentences (i.e. formulas without free variables): φ_{pre}, φ_{inv} and φ_{post}. In both φ_{pre} and φ_{inv} 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 $\sigma(l)$. Later, during the whole continuous evolution, the formula φ_{inv} must hold in the structure $\mathfrak{A}$ with continuous values changed as prescribed by the solution of the system $\mathcal{D}$ (defined above). Finally, the formula φ_{post} must hold in the resulting structure after rewriting. During simultaneous execution of a few rules with constraints and with given time bounds t_i, one of the invariants φ_{inv} may cease to hold. In such case, the rule is applied at that moment of time, even before $t^0 = \min t_i$ is reached — but of course only if φ_{post} holds afterwards. If φ_{post} does not hold, the rule is ignored and time goes on for the remaining rules.

[4]We provide additional syntax (shorthands) for useful patterns.

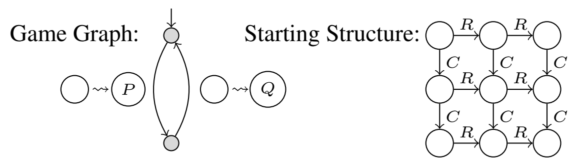

Figure 2: Tic-tac-toe as a structure rewriting game.

Structure Rewriting Games

As you could judge from the cat and mouse example, one can describe a structure rewriting game simply by providing a set of allowed rules for each player. Still, in many cases it is necessary to have more control over the flow of the game and to model probabilistic events. For this reason, we use labelled directed graphs with probabilities in the definition of the games. The labels for each player are of the form:

$$\lambda = (\mathfrak{L} \to_s \mathfrak{R}, \mathcal{D}, \mathcal{T}, \varphi_{\mathrm{pre}}, \varphi_{\mathrm{inv}}, \varphi_{\mathrm{post}}, I_t, \overline{I_p}, \mathrm{m}, \tau_{\mathrm{e}}).$$

Except for a rewriting rule with invariants, the label λ specifies a time interval $I_t \subseteq [0, \infty)$ from which the player can choose the time bound for the rule and, if there are other continuous parameters $p_1, \ldots, p_n$, also an interval $I_{p_j} \subseteq \mathbb{R}$ for each parameter. The element $\mathrm{m} \in \{1, *, \infty\}$ specifies if the player must choose a single match of the rule ($\mathrm{m} = 1$), apply it simultaneously to all possible matches ($\mathrm{m} = \infty$, useful for modeling nature) or if any number of non-intersecting matches might be chosen ($\mathrm{m} = *$); τ_{e} tells which relations must be matched exactly (all other are in τ_{h}).

We define a *general structure rewriting game* with k players as a directed graph in which each vertex is labelled by k sets of labels denoting possible actions of the players. For each k-tuple of labels, one from each set, there must be an outgoing edge labelled by this tuple, pointing to the next location of the game if these actions are chosen by the players. There can be more than one outgoing edge with the same label in a vertex: In such case, all edges with this label must be assigned probabilities (i.e. positive real numbers which sum up to 1). Moreover, an end-point of an interval I_t or I_p in a label can be given by a parameter, e.g. x. Then, each outgoing edge with this label must be marked by $x \sim \mathcal{N}(\mu, \sigma)$, $x \sim \mathcal{U}(a, b)$ or $x \sim \mathcal{E}(\lambda)$, meaning that x will be drawn from the normal, uniform or exponential distribution (these 3 chosen for convenience). Additionally, in each vertex there are k real-valued terms of the logic presented above which denote the payoff for each player if the game ends at this vertex.

A play of a structure rewriting game starts in a fixed first vertex of the game graph and in a state represented by a given starting structure. All players choose rules, matches and time bounds allowed by the labels of the current vertex such that the tuple of rules can be applied simultaneously. The play proceeds to the next vertex (given by the labeling of the edges) in the changed state (after the application of the rules). If in some vertex and state it is not possible to apply any tuple of rules, either because no match is found or because of the constraints, then the play ends and payoff terms are evaluated giving the outcome for each player.

Example. Let us define tic-tac-toe in our framework. The state of the game is represented by a structure with 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. 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 states 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). The two diagonal relations can be defined by $D_1(x, y) = \exists z(R(x, z) \wedge C(z, y))$ and $D_2(x, y) = \exists z(R(x, z) \wedge C(y, z))$ and a line of three by $L(x, y, z) = (R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)) \vee (D_1(x, y) \wedge D_1(y, z)) \vee (D_2(x, y) \wedge D_2(y, z))$. Using this definitions, the winning condition for the first player is given by $\varphi = \exists x \exists y \exists z(P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z))$ and for the other player analogously with Q. 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 winning condition of the other player.

Playing the Games

When playing a game, players need to decide what their next move is. To represent the preferences of each player, or rather her expectations about the outcome after each step, we use *evaluation games*. Intuitively, an evaluation game is a statistical model used by the player to assess the state after each move and to choose the next action. Formally, an evaluation game $\mathcal{E}$ for $\mathcal{G}$ is just *any* structure rewriting game[5] with the same number of players and with extended signature. For each relation R and function f used in $\mathcal{G}$ we have two symbols in $\mathcal{E}$: R and R_{old}, respectively f and f_{old}.

To explain how evaluation games are used, imagine that players made a concurrent move in $\mathcal{G}$ from $\mathfrak{A}$ to $\mathfrak{B}$ in which each player applied his rule $\mathfrak{L}_i \rightarrow_{s_i} \mathfrak{R}_i$ to certain matches. We construct a structure $\mathfrak{C}$ representing what happened in the move as follows. The universe of $\mathfrak{C}$ is the universe of $\mathfrak{B}$ and all relations R and functions f are as in $\mathfrak{B}$. Further, for each $b \in \mathfrak{B}$ let us define the corresponding element $a \in \mathfrak{A}$ as either b, if $b \in \mathfrak{A}$, or as $s_i(b)$, if b was in some right-hand side structure $\mathfrak{R}_i$ and replaced a. The relation R_{old} contains the tuples $\overline{b}$ which replaced some tuple $\overline{a} \in R^{\mathfrak{A}}$. The function $f_{\mathrm{old}}(b)$ is equal to $f_{\mathrm{old}}(a)$ (evaluated in $\mathfrak{A}$) if b replaced a and it is 0 if b did not replace any element. We use $\mathfrak{C}$ as the starting structure for the evaluation game $\mathcal{E}$. This game is then played (as described below) and the outcome of $\mathcal{E}$ is used as an assessment of the move $\mathfrak{C}$ for each player.

As you can see above, the evaluation game $\mathcal{E}$ is used to predict the outcomes of the game $\mathcal{G}$. This can be done in many ways: In one basic case, no player moves in the game $\mathcal{E}$ — there are only probabilistic nodes and thus $\mathcal{E}$ represents just a probabilistic belief about the outcomes. In another basic case, $\mathcal{E}$ returns a single value — this should be used if the player is sure how to assess a state, e.g. if the game ends there. In the next section we will construct evaluation games in which players make only trivial moves depending on certain formulas — in such case $\mathcal{E}$ represents a more complex probability distribution over possible payoffs. In general, $\mathcal{E}$ can be an intricate game representing the judgment process of the player. In particular, note that we can use $\mathcal{G}$ itself for $\mathcal{E}$, but then without evaluation games any more to avoid circularity. This corresponds to a player simulating the game itself as a method to evaluate a state.

We know how to use an evaluation game $\mathcal{E}$ to get a payoff vector (one for each player) denoting the expected outcome of a move. These predicted outcomes are used to choose the action of player i as follows. We consider all discrete actions of each player and construct a matrix defining a normal-form game in this way. Since we approximate ODEs by polynomials symbolically, we keep the continuous parameters playing $\mathcal{E}$ and get the payoff as a piecewise polynomial function of the parameters. This allows to solve the normal-form game and choose the parameters optimally. To make a decision in this game we use the concept of iterated regret minimization (over pure strategies), well explained in [7].

The regret of an action of one player when the actions of the other players are fixed is the difference between the payoff of this action and the optimal one. A strategy minimizes regret if it minimizes the maximum regret over all tuples of actions of the other players. We iteratively remove all actions which do not minimize regret, for all players, and finally pick one of the remaining actions at random. Note that for turn-based games this corresponds simply to choosing the action which promises the best payoff. In case no evaluation game is given, we simply pick an action randomly and the parameters uniformly, which is the same as described above if the evaluation game $\mathcal{E}$ always gives outcome 0.

With the method to select actions described above we can already play the game $\mathcal{G}$ in the following basic way: Let all players choose an action as described and play it. While we will use this basic strategy extensively, note that, in case of poor evaluation games, playing $\mathcal{G}$ like this would normally result in low payoffs. One way to improve them is the Monte-Carlo method: Play the game in the basic way K times and, from the first actions in these K plays, choose the one that gave the biggest average payoff. Already this simple method improves the play considerably in many cases. To get an even better improvement we simulateously construct the UCT tree, which keeps track of certain moves and associated confidence bounds during these K plays.

A node in the UCT tree consists of a position in the game $\mathcal{G}$ and a list of payoffs of the plays that went through this position. We denote by $n(v)$ the number of plays that went through v, by $\overline{\mu}(v)$ the vector of average payoffs (for each of the players) and by $\overline{\sigma}(v)$ the vector of square roots of variances, i.e. $\sigma_i = \sqrt{\sum_{p_i}(p_i^2)/n - \mu_i^2}$ if p_i are the recorded payoffs for player i. First, the UCT tree has just one node, the current position, with an empty set of payoffs. For each of the next K iterations the construction of the tree proceeds as follows. We start a new play from the root of the tree. If we are in an internal node v in the tree, i.e. in one which already has children, then we play a regret minimizing strategy (as discussed above) in a normal-form game with payoff matrix given by the vectors $\overline{\mu'}(w)$ defined as follows. Let $\sigma_i'(v) = \sigma_i(v)^2 + \Delta \cdot \sqrt{\frac{2\ln(n(v)+1)}{n(w)+1}}$ be the upper confidence bound on variance and to scale it let $s_i(v) =$

[5]In fact it is not a single game $\mathcal{E}$ but one for each vertex of $\mathcal{G}$.

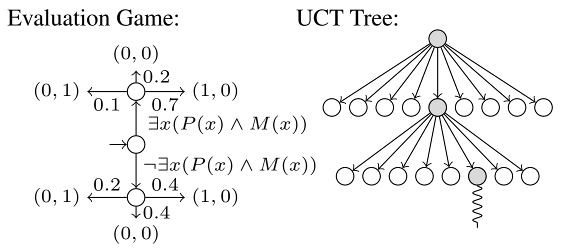

Figure 3: Evaluation game for tic-tac-toe and a UCT tree.

$\min(1/4, \sigma_i'(v)/\Delta)$, where Δ denotes the payoff range, i.e. the difference between maximum and minimum possible payoff. We set $\mu_i'(w) = \mu_i(w) + C \cdot \Delta \cdot \sqrt{\frac{\ln(n(v)+1)}{n(w)+1} s_i(v)}$. The parameter C balances exploration and exploitation and the thesis [3] gives excellent motivation for precisely this formula (UCB1-TUNED). Note that for turn-based games, when player i moves, we select the child w which maximizes $\mu_i'(w)$. When we arrive in a leaf of the UCT tree, we first add all possible moves as its children and play the evaluation game a few ($E > 1$) times in each of them. The initial value of $\overline{\mu}$ and $\overline{\sigma}$ is computed from these evaluation plays (both must be set even if $n = 0$). After the children are added, we select one and continue to play with the very basic strategy: Only the evaluation game is used to choose actions and the UCT tree is not extended any more in this iteration. When this play of $\mathcal{G}$ is finished, we add the received payoff to the list of recorded payoffs of each node on the played path and recalculate $\overline{\mu}$ and $\overline{\sigma}$. Observe that in each of the K iterations exactly one leaf of the UCT tree is extended and all possible moves from there are added. After the K-th iteration is finished, the action in the root of the UCT tree is chosen taking into account only the values $\overline{\mu}$ of its children.

Example. Consider the model of tic-tac-toe presented previously and let the formula $M(x) = \exists y\, C(x,y) \wedge \exists y\, C(y,x) \wedge \exists y\, R(x,y) \wedge \exists y\, R(y,x)$ express that x is the position in the middle of the board. In Figure 3 we depicted a simple evaluation game, which should be interpreted as follows. If the first player made a move to the middle position, expressed by $\exists x(P(x) \wedge M(x))$, then the probability that the first player will win, i.e. of payoff vector $(1,0)$, is 0.7. The probability that the second player will win is 0.1 and a draw occurs with probability 0.2. On the other hand, if the first player did not move to the middle, then the respective probabilities are 0.4, 0.2 and 0.4. When the construction of the UCT tree starts, a payoff vector is assigned to the state after each of the 9 possible moves of the first player. The payoff vector is one of $(1,0), (0,1)$ and $(0,0)$ and is chosen randomly with probabilities $0.7, 0.1, 0.2$ for the middle node in the UCT tree and with probabilities $0.4, 0.2, 0.4$ for all other 8 nodes, as prescribed by the evaluation game. The first iteration does not expand the UCT tree any further. In the second iteration, if the middle node is chosen to play, then its 8 children will be added to the UCT tree. The play in this iteration continues from one of those children, as depicted by the snaked line in Figure 3.

Learning Evaluation Games

Even during a single play of a game $\mathcal{G}$ we construct many UCT trees, one for each move of each player, as described above. A node appearing in one of those trees represents a state of $\mathcal{G}$ and a record of the payoffs received in plays from this node. After each move in $\mathcal{G}$, we collect nodes from the UCT tree from which a substantial number of plays were played, i.e. which have n bigger than a confidence threshold N. These nodes, together with their payoff statistics, are used as a training set for the learning procedure.

The task of the learning mechanism started on a training set is to construct an evaluation game $\mathcal{E}$, preferably as simple as possible, which, started in a state from the training set, gives payoffs with a distribution similar to the one known for that state. Observe that the game $\mathcal{E}$ constructed for a training set from some plays of $\mathcal{G}$ is therefore a simplified probabilistic model of the events which occured during simulated plays of $\mathcal{G}$. Further — a game $\mathcal{E}$ costructed from plays of $\mathcal{G}$ which already used an evaluation game $\mathcal{E}'$ models plays between players who already "know" $\mathcal{E}'$. Note how this allows incremental learning: each evaluation game can be used to learn a new one, modeling plays between smarter players.

We present a non-deterministic construction of candidate evaluation games, i.e. we provide only the basic operations which can be used and leave the choice to an external function. Still, as we show next, even very simple choices can produce useful evaluation games. During the whole construction process the procedure maintains a few sets: a set G of evaluation games, a set S of structures, a set T of equation terms, and a set Φ of formulas and real-valued terms. Initially the set G contains at least a trivial game, S a trivial structure and T and Φ may be empty. The learning procedure constructs new games, structures, terms and formulas until, at some point, it selects one game from G as the result.

The rules for adding new formulas and terms to Φ closely resemble the syntax of our logic presented before. For each syntactic rule we allow to add to Φ its result if the required formulas and terms are already in Φ. For example, we can add the literal $P(x)$ to an empty set Φ, then add $Q(x)$, create $P(x) \wedge Q(x)$, and finally use the existential quantifier to create $\exists x(P(x) \wedge Q(x))$. The rules for construction of equational terms are analogous. For structures, we allow to add a single element or a single relation tuple to a structure from S and to take disjoint sum of two structures from S. Finally for games we allow compositional rules similar to the constructions in Parikh's Game Logic, cf. [10].

Clearly, the rules above are very general and it is up to the external function to use them to create a good evaluation game. In our first experiments, we decided to focus on a very simple heuristic which does not create any structures or equations. It uses only formulas and probabilistic vertices and the payoff is always one of the vectors already occuring in the training set. Moreover, we do not allow arbitrary formulas but only existentially quantified conjunctions of literals. Evaluation games created by our function have thus similar form to the one presented on the left side of Figure 3. To decide which formula to add next, our function extends formulas already kept in Φ by a literal and keeps the one which is the best selector. This is very similar to the

	White uses $\mathcal{E}$	Black uses $\mathcal{E}$
Gomoku	78%	82%
Breakthrough	77%	73%

Table 1: Playing with a learnt evaluation against pure UCT.

Apriori algorithm for frequent itemset mining, just instead of items we use literals and a set of literals is understood as their conjunction, existentially quantified. Transactions in this sense are sets of states with average payoff in a specific interval. The found formulas are then used as constraints of a transition to a probabilistic node which is constructed so as to model the distribution of payoffs in the states from the training set which satisfy this formula. (cf. Figure 3).

Experimental Results

The described algorithms are a part of Toss (`toss.sourceforge.net`), an open-source program implementing the presented game model and a GUI for the players.[6] To construct a scalable solver for our logic we used a SAT solver (MiniSAT) to operate on symbolic representations of MSO variables and implmented a quantifier elimination procedure for real numbers based on Muchnik's proof. The UCT algorithm and the rewriting engine were implemented in OCaml and the GUI in Python using the Qt4 library. In Toss, we defined a few board games and some systems with continuous dynamics. In this paper, we present preliminary results for Breakthrough and Gomoku, two games often used to evaluate game playing programs.

The strength of the UCT algorithm has been evaluated before: it is respectable, but can only go so far without any evaluation function. We used our learning procedure to get an evaluation game for both Breakthrough and Gomoku. Only top-performing formulas were selected, in case of Breakthrough it was one simple formula meaning "beat if possible" and for Gomoku the formulas suggested to put stones near the already placed ones. While these formulas are basic, we present in Table 1 the percentage of plays won by a UCT player using the evaluation game against an opponent using none. As you can see, this is a significant majority of cases — even playing black in Breakthrough and white in Gomoku, i.e. starting second, which is a weaker position.

Perspectives

We presented a general model of games and an algorithm which can both play the game in a reasonable way and learn from past plays. There are several phenomena which we did not include in our model. On the side of games, we allowed neither imperfect nor incomplete information, so players must fully know the game and its state, which is not realistic. On the modeling side, relational structures give no direct way to represent hierarchies, which should be improved as well. For practical use of our system on larger examples it is also important to introduce a type system, which should be integrated with our logic. We started to investigate this problem from the theoretical side in [8]. To improve the learning procedure, we plan to investigate classical learning algorithms (e.g. C4.5), and recent program induction methods (e.g. [9]). These can hopefully find new rewriting rules and in this way generate novel evaluation games. One could also try to analyze UCT using higher-order probabilities [6].

Even with the drawbacks mentioned above, the model we presented is, to our knowledge, the most general kind of games for which a playing algorithm is implemented. In addition to this generality, we have chosen a hypergraph representation for states, which is already used in AGI projects. Our playing algorithm is based on the upper confidence bounds method, which is not only established for board games but also scales to other domains, e.g. robotic visual learning [11]. Thus, it could be opportune to use our system as a basis for an AGI project and we look forward to cooperating with AGI system builders on such integration.

[6]Currently released version 0.3 of Toss does not support all the features we described. Our tests were conducted on a recent version in code repository and will be included in the next release.

References

[1] S. Burmester, H. Giese, E. Münch, O. Oberschelp, F. Klein, and P. Scheideler. Tool support for the design of self-optimizing mechatronic multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 10(3):207–222, 6 2008.

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

[3] S. Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. Dissertation, University Paris-Sud 11, 2007.

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

[5] B. Goertzel. Patterns, hypergraphs and embodied general intelligence. In *Proc. of IJCNN'06*, pages 451–458, 2006.

[6] B. Goertzel, M. Ikle, and I. L. Freire Goertzel. *Probabilistic Logic Networks: A Comprehensive Framework for Uncertain Inference*. Springer, 2008.

[7] J. Y. Halpern and R. Pass. Iterated regret minimization: A new solution concept. In *Proc. of IJCAI'09)*, pages 153–158, 2009.

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

[9] M. Looks and B. Goertzel. Program representation for general intelligence. In *Proc. of AGI'09*, 2009.

[10] R. Ramanujam and S. Simon. Dynamic logic on games with structured strategies. In *Proc. of KR'08*, pages 49–58. AAAI Press, 2008.

[11] M. Salganicoff, L. H. Ungar, and R. Bajcsy. Active learning for vision-based robot grasping. *Machine Learning*, 23:251–278, 1996.