Noname manuscript No.
(will be inserted by the editor)

Knowledge-Based General Game Playing

Sebastian Haufe · Daniel Michulke · Stephan Schiffel · Michael Thielscher

Received: 30 July 2010 / Accepted: 14 September 2010

Abstract Although we humans cannot compete with computers at simple brute-force search, this is often more than compensated for by our ability to discover structures in new games and to quickly learn how to perform highly selective, informed search. To attain the same level of intelligence, general game playing systems must be able to figure out, without human assistance, what a new game is really about. This makes General Game Playing in ideal testbed for human-level AI, because ultimate success can only be achieved if computers match our ability to master new games by acquiring and exploiting new knowledge. This article introduces five knowledge-based methods for General Game Playing. Each of these techniques contributes to the ongoing success of our FLUXPLAYER [15], which was among the top four players at each of the past AAAI competitions and in particular was crowned World Champion in 2006.

Keywords General game playing · Knowledge representation · Fluxplayer

This work was partially supported by *Deutsche Forschungsgemeinschaft* under project number TH 541/16-1 and by the Australian Research Council under project number FT 0991348.

S. Haufe · D. Michulke · S. Schiffel
Department of Computer Science, Dresden University of Technology, 01062 Dresden, Germany
E-mail: sebastian.haufe@tu-dresden.de

D. Michulke
E-mail: daniel.michulke@inf.tu-dresden.de

S. Schiffel
E-mail: stephan.schiffel@inf.tu-dresden.de

M. Thielscher
School of Computer Science and Engineering, The University of New South Wales, Sydney NSW 2052, Australia
E-mail: mit@cse.unsw.edu.au

1 Introduction

As a Grand Challenge for Artificial Intelligence, General Game Playing requires to combine methods from a variety of sub-disciplines, including reasoning, search, computer game playing, and learning. While simple games can be solved by brute-force search and recent research has shown that Monte-Carlo methods provide a successful form of selective blind search, moving from blind to informed search is a great endeavour in General Game Playing as it requires a player to fully automatically analyse the bare rules of unknown games in order to extract and exploit game-specific knowledge.

This article gives an overview of five essential methods for knowledge-based General Game Playing:

1. We show how to automatically generate evaluation functions for non-terminal positions.
2. We demonstrate the application of machine learning techniques to improve these evaluation functions.
3. We illustrate how a system can automatically derive new properties of games using automated theorem proving.
4. We present two specific techniques that help to reduce the complexity of the search space by mimicking the ability of humans to
   (a) detect symmetries in games and
   (b) identify subgames in composite games.

Each of these techniques contributes to the ongoing success of our FLUXPLAYER [15], which was among the top four players at each of the past AAAI competitions and in particular was crowned World Champion in 2006.

2 Generating Evaluation Functions

In virtually all interesting game the state space is too large to be searched exhaustively. It is therefore necessary to be able to evaluate intermediate states and to decide on the moves based on that evaluation. One way to evaluate non-terminal states is to use Monte-Carlo simulations of the game. In contrast, knowledge-based approaches to General Game Playing typically use a heuristic evaluation function. Heuristics are typically derived by making simplifying assumptions about the domain. The main assumption that we make is that the atomic properties of the game states can be changed independently. Of course this does not hold in general, but it is a good approximation in many games.

We have developed a method for generating a heuristic function based on the rules of the game, specifically the rules for terminal and goal states. The main advantage of our approach is that the evaluation function can be generated very fast and does not need to be trained with a lot of examples. The idea for our evaluation function is to calculate a *degree* of truth of the formulas defining predicates `goal` and `terminal` in the state to be evaluated. The values for `goal` and `terminal` are combined in such a way that terminal states are avoided as long as the goal is not fulfilled, that is, the value of `terminal` has a negative impact on the evaluation of the state if goal has a low value and a positive impact otherwise.

2.1 Evaluating Formulas

Our approach is based on fuzzy logic, which is used to assign real numbers between 0 and 1 to atoms depending on their truth value. A pair of functions T and S is then used similar to standard t-norm and t-co-norms to calculate the degree of truth of conjunctions and disjunctions, respectively. We define the evaluation of a GDL formula with respect to a game state z as follows:

$$\begin{aligned} eval(a, z) &\in [0, 1] \\ eval(f \wedge g, z) &= T(eval(f, z), eval(g, z)) \\ eval(f \vee g, z) &= S(eval(f, z), eval(g, z)) \\ eval(\neg f, z) &= 1 - eval(f, z) \end{aligned}$$

where f, g are GDL formulas, a is an atomic formula.

An atomic formula a is evaluated against state z and given a value between 0 and 1 depending on whether a holds in z and possibly additional information, e.g., the likelihood of a to stay true (or false) until the final state of the game. Values between 0.5 and 1 are used for atoms that hold in z, and values between 0 and 0.5 for atoms that do not hold in z. The exact values may be chosen according to distance metrics or other information about the game. A value of 1 is assigned to an atom a only if a is currently true and persistent until the end of the game, like markers in Tic-Tac-Toe or pieces in Quarto that cannot be moved or removed from the board. Similarly, a value of 0 is assigned to a if it is currently false and can not be fulfilled any more, e.g., because there is already another persistent marker in the same position. The additional information, like persistence of markers, boards and distance metrics can be acquired using pattern recognition on the rules of the game [15, 10], simulations of the game [15, 10, 4], or automatic theorem proving as described in Section 4 below.

As we use values < 1 for atoms that are true, we cannot simply adopt a standard continuous t-norm like $T(x, y) = x * y$, because their monotonicity implies that $eval(a_1 \wedge \ldots \wedge a_n, z)$ is near 0 for large n. In other words, the evaluation may consider the formula $a_1 \wedge \ldots \wedge a_n$ to be false even if all a_i hold in z. To overcome this problem, we use a threshold t with $0.5 < t < 1$, with the following intention: values above t denote true and values below $1 - t$ denote false. The truth function we use for conjunction is now defined as:

$$T(a, b) = \begin{cases} max(T'(a, b), t), & \text{if } min(a, b) > 0.5 \\ T'(a, b) & \text{otherwise} \end{cases}$$

where T' denotes an arbitrary standard t-norm. This function together with the associated truth function for disjunctions, $S(a, b) = 1 - T(1 - a, 1 - b)$, ensures that formulas that are true always get a value greater or equal t and formulas that are false get a value smaller or equal $1-t$. Thus the values of different formulas stay comparable. This is necessary, for example, in a game with multiple goals. The disadvantage of this definition is that T is not associative, at least in cases of continuous t-norms T', and is therefore not a t-norm itself in general. Therefore, the evaluations of semantically equivalent but syntactically different formulas can differ. However, it is possible to minimise that effect by choosing an appropriate t-norm T'.

2.2 Evaluating States

The complete heuristic evaluation function for a state z and role r in a particular game is defined as follows:

$$h(r, z) = \frac{1}{\sum_{v \in GV} v} * \sum_{v \in GV} h(r, v, z) * v$$

$$h(r, v, z) = \begin{cases} eval(goal(r, v) \vee terminal, z), & \text{if } goal(r, v) \\ eval(goal(r, v) \wedge \neg terminal, z), & \text{otherwise} \end{cases}$$

Here, GV is the domain of goal values, $goal(r, v)$ is the goal formula for the goal value v of role r, and $terminal$ is the terminal condition of the game. That means the heuristic value of a state is calculated by combining heuristics $h(r, v, z)$ for each goal value v of the domain of goal values GV weighted by the goal value v. The heuristics for each possible goal value is calculated as the evaluation of the disjunction of the goal and terminal formulas in case the goal is fulfilled, that is, the heuristics tries to reach a terminal state if the goal has been achieved. On the other hand the heuristics tries to avoid terminal states as long as the goal is not reached. The use of a disjunction in the first case and conjunction in the second case ensures a clear distinction between both cases: in the first case the evaluation yields a value > 0.5, whereas the evaluation is ≤ 0.5 in the second case.

The observant reader may have noticed that we only deal with propositional formulas in the evaluation function. In fact, before applying the evaluation function we try to ground the goal and terminal rules and remove all non-keyword predicates by unfolding them, i.e., replacing every occurrence of a predicate p by a disjunction of the bodies b of rules $p :- b$. Grounding and unfolding may fail or may be incomplete, e.g., because it takes too long, the representation gets too big, or there are recursive rules. In that case we treat subformulas with quantifiers (variables) as atomic formulas.

3 Improving Evaluation Functions

While fuzzy logic proves to be a plausible way to evaluate non-terminal states, it also suffers from two disadvantages: Since propositional fuzzy logic formulas provide no native means to weigh propositions, all constituents of a formula are implicitly assumed to be equally important, a fact that we consider too restrictive for most games. On the other hand, a fuzzy logic evaluation comes with no learning capabilities. Given that learning is a crucial ability for humans to understand and play games well, a lack thereof is a substantial disadvantage.

While these problems can be addressed by neural networks, their application brings other disadvantages. Before using them, neural networks must be trained in order to capture the behaviour of a given function — a process known to be time-consuming. Moreover, the yet-to-be-trained networks are difficult to initialise with regard to parameters as the number of neurons, the connections between them, and their initial connection weights. A bad initialisation may result in the non-convergence of the network.

Both problems, however, can be solved by employing a logic-to-neural-network transformation. Here, the structure and parameters of the neural network are directly derived from the set of propositional rules that the network is supposed to represent. In this way, we can map the goal function of a game to a neural network such that the network is usable ad-hoc while it correctly evaluates goal states and allows for a fuzzy-comparable distinction of non-terminal states.

3.1 Goal to Value Function Transformation

To transform a goal function for a specific role to an evaluation function, one first needs to obtain a propositional representation of the goal function. Again, we encounter the problem described in Section 2.2 and solve it similarly. Besides, for neural networks there are algorithms that enable correct representation of first-order logic clauses, however, up to now none of these has been successfully put into practice for a complex problem domain.

Having obtained a propositional representation, we translate it further to a neural network using the $C - IL^2P$ algorithm [5].

The algorithm represents truth and falsity using bipolar neurons with an output in the interval $(-1, 1)$. An equivalence between a propositional variable p and its corresponding neuron with output o_p is established by enforcing

$$p \leftrightarrow o_p \in (A, 1)$$
$$\neg p \leftrightarrow o_p \in (-1, -A)$$

where the parameter $A > \frac{k-1}{k+1}$ depends on the maximum number k of antecedents in any conjunction or disjunction in the propositionalised goal formula.

Neuron output values in $(-A, A)$ are guaranteed not to occur by imposing further conditions on the standard weight W for connections between two neurons; due to lack of space we have to refer the interested reader to [11] for details.

With all goal functions transformed, we can construct exactly one network for each tuple of role and goal value in the game. To obtain an evaluation function for a specific role, we normalise the output for each of the role's associated networks and aggregate the outputs of these networks weighted by the goal value the network represents. The resulting evaluation function is equivalent to the one described in [15], with the same heuristic measures applicable. The difference, however, consists in facts in a state (= propositions) being represented as input neurons, and all conjunctions and disjunction on these facts represented as hidden layer neurons.

3.2 Learning in General Games

A direct benefit of this new representation is that we can now define a learning process that is basically the opposite process of state evaluation: Since for terminal states we know the exact state value, we may also identify the network(s) that are supposed to return *true* and may assign training signal $t = 1$ to their respective output neurons. Correspondingly, all other networks not matching the state value are assigned a negative training signal. Training can now be performed by simply presenting the terminal state as network input and applying standard learning algorithms such as backpropagation.

The training can be further enhanced by using not only terminal states, but also non-terminal states. By discounting non-terminal states with increasing distance to the terminal state, we may thus resemble learning patterns as used in $TD(\lambda)$ [18,19] where the training signal for the k^{th} predecessor is discounted by λ^k with $\lambda \in [0, 1]$.

Finally, we may even refrain from discounting a state if we know the state to inevitably lead to a terminal state of the same value. If we encounter a state where optimal play leads to a winning position, we may count this state as well as already won. This type of search for optimal play is already done during the match when searching the game tree, and by reusing this information we substitute a probably erroneous discounted training signal by a provably correct one.

The application of these learning strategies allows the evaluation function to gradually adopt a bidirectional search pattern where forward game-tree search is complemented by learned preterminal state patterns.

3.3 Further Ramifications

By extracting training data directly from matches, we may use non-artificial states that occur in the course of reasonable game play and under no time constraints, as opposed to possibly invalid pseudo-states [4] or states generated from expensive random playouts.

Furthermore, the approach enables the refinement of the evaluation function at any point of time where a correct state value for some state is available. In the context of Game Playing, this amounts to the possibility to initiate a learning cycle directly after each match, or even within matches due to the monotonicity of goal values required by GDL. This allows a constant improvement of weights for non-logical features as e.g. distance evaluations or, in fact, any other proposition in the evaluation function. These weights correspond to the utility of the proposition/feature and represent a form of feedback that allows further enhancements such as continuous feature integration and removal, or a self-improving agent based on self-play.

In this way, the approach can be seen as a synergetic approach that unifies logic with neural networks and reinforcement learning. Given that the approach is very recent, however, more empirical evidence is needed in order to prove the claims and to demonstrate its full capacity.

4 Learning by Theorem Proving

The ability to form knowledge about a new game is a prerequisite for both the automated generation of search heuristics and the construction of evaluation functions for non-terminal positions. While successful general game playing systems like [10,4] do extract this kind of knowledge, they do not actually attempt to prove it; rather they generate a number of random sample matches to test a property, and then rely on the correctness of this informed guess.

The first formal approach to the formalisation and automated proving of properties employs complete search through the state transition diagram for a game [13]. However, for games that are simple enough to make this practically feasible, a general game player does not actually need game-specific knowledge because it can solve the game by exhaustive search anyway. For this reason, we have developed a local proof method that is practically feasible [16]. In case of game-specific properties that hold across all reachable states the key idea is to reduce the automated theorem proving task to a simple proof of an induction step and its base case. An example from this property class is the uniqueness of cell content in a game of Tic-Tac-Toe, which can be proved by checking the property in the initial state of the game for the base case, and by checking all direct successor states of states that itself entail the property for the induction step.

A recent extension enlarges this class to properties which need to be verified against reachable finite sequences of successive game states. An example is the fact that, again in Tic-Tac-Toe, a marked cell persists from one state to the next, which needs a lookahead to all direct successor states in order to be verified in a state. The extension incorporates ideas of an approach to solve single player games, which is especially useful for endgame position evaluation via depth restricted forward search [20]. In the following, we give an overview of syntax and semantics of the language which allows to formulate what we call *temporally extended state invariants*; we then sketch a local proof method to verify properties of this class against a given

game description; and finally we recast the main ideas of [20] in terms of our general framework.

4.1 Temporal Property Formulas

Syntax Temporally extended state invariants, in the following also called *(temporal) properties*, are formulated via essentially propositional formulas, based on the ground atoms of a given game description. Additionally, the unary connective $\bigcirc$ is allowed to refer to a successor game state. E.g., the aforementioned property of cell content persistence, here exemplarily for a board cell with coordinates $(1, 1)$ and marker x, can be formulated via the formula

$$true(cell(1, 1, x)) \supset \bigcirc true(cell(1, 1, x)).$$

It states that once cell $(1, 1)$ is marked with x in a game state, the cell keeps that content in each direct successor game state. This formula being entailed in every reachable state implies that marker x, once placed, is persistent throughout the remainder of the game. We further on refer to the maximal nesting of $\bigcirc$ in φ as the *degree of* φ, denoted by $deg(\varphi)$.

Semantics A GDL description can be interpreted as a transition system over states [17], where a state is a finite set of terms over the signature of the GDL description. A transition from state S to state S' occurs if S is non-terminal and each player performing exactly one of its legal actions in S results in S'. Entailment of a temporally extended state invariant φ in a state S then amounts to checking φ w.r.t. the partial game tree which roots in S and is built up to depth $deg(\varphi)$ via the defined transition system. E.g., the aforementioned property $true(cell(1, 1, x)) \supset \bigcirc true(cell(1, 1, x))$ is entailed in a state S if and only if either $cell(1, 1, x)$ is not contained in S, or $\bigcirc true(cell(1, 1, x))$ is entailed in S, which holds if and only if $cell(1, 1, x)$ is contained in each of the direct successor states of S.

4.2 Verification of Temporal Properties

The induction proof method to verify a temporally extended state invariant φ w.r.t. all reachable states is put into practice using Answer Set Programming. (For a thorough introduction to answer set programming see, e.g., [7].) We construct two answer set programs (ASPs) dependent on φ in order to establish proofs for a base case and an induction step. The base case shows that φ is entailed in the initial state. The induction step shows that, provided a state entails φ, each legal successor state will also entail φ. In conclusion, then, φ is entailed in all reachable states.

Base Case The ASP for the base case is constructed as follows:

(1) *Temporal GDL description:* We construct a variant of the game rules for each of the time points $i = 0, \ldots, deg(\varphi)$, omitting the rules for the initial state. In each variant we enrich each predicate with a further argument i and "glue" all time extended game rule variants together by replacing each enriched $next(f, i)$ with $true(f, i + 1)$.
(2) *State Encoding:* We include a variant of the rules for the initial state where each $init(f)$ is replaced by $true(f, 0)$.
(3) *Action Encoding:* We add rules encoding that each player performs exactly one legal action at each time point between 0 and $deg(\varphi) - 1$. As a result, each solution to the constructed ASP (1) + (2) + (3) corresponds to a finite sequence of successive game states with time horizon $deg(\varphi)$, starting in the initial state.
(4) *Property Encoding:* We encode φ such that only solutions of (1) + (2) + (3) are kept which represent φ-violating sequences of successive game states.

The proof of φ being entailed in the initial state S_{init} is then obtained via contradiction: if the ASP for the base case is inconsistent, then no sequence starting in S_{init} violates φ. Hence, S_{init} entails φ.

Induction Step The encoding for the induction step differs from the base case encoding only in that the verification is done over property formula $\varphi \supset \bigcirc\varphi$ instead of just φ. Moreover, the initial state encoding (2) is replaced by an ASP which generates the reachable states. In conclusion, if the induction step encoding is inconsistent, then every generated state entails $\varphi \supset \bigcirc\varphi$. If additionally the base case ASP is inconsistent, it follows that φ is entailed in all *reachable* states.

4.3 Solving Single Player Games

Answer Set Programming can also be used when playing Single-Player Games [20]. The main idea is to accomplish the goal of the single player p—a terminal state which maximises its outcome—by constructing an ASP such that each solution represents a sequence of legal actions for p by which the goal is reached. The construction assumes a given time horizon for the maximal length of this sequence.

A goal can be seen as the set of all states that entail property formula $\varphi = terminal \wedge goal(p, 100)$. In order to find a sequence that starts in the initial state and

ends in a state which entails φ, we construct a base case ASP for the formula

$$\varphi_t = \bigcirc^0 \neg\varphi \wedge \bigcirc^1 \neg\varphi \wedge \ldots \wedge \bigcirc^t \neg\varphi,$$

where $\bigcirc^t$ denotes t consecutive $\bigcirc$ connectives for a given time horizon t. If the constructed ASP is consistent, then each solution represents a sequence of successive game states that starts in the initial state and violates φ_t. By construction of φ_t this sequence is of length $\leq t$ and such that its last state entails φ—in other words, we obtain an action sequence that achieves the goal. If, however, the constructed ASP does not have a solution, then there exists no sequence achieving the goal within time horizon t.

4.4 Characteristics and Experiments

We have formally proved the soundness of our automated theorem proving method [21], hence it provides a reliable technique to prove valuable game properties. It is also practically useable—we briefly report on experiments in the next paragraph—but this is attended by the loss of completeness: since the exact set of reachable states is initially unknown and hard to compute in general, we construct an easily obtainable superset when encoding the ASP for an induction step. This, however, may result in phantom states which provide counter-examples for properties that are actually valid.

The incompleteness of our proof method notwithstanding, numerous experiments have shown that a variety of properties can easily be proved in practice, including the detection of board properties, uniqueness and periodicity of player control, zero-sum, turn taking, persistence and the solving of single-player games [16, 21, 20]. The computation times are in the range of seconds for many games taken from previous General Game Playing competitions, confirming that our approach provides significant assistance to a general game player.

5 Symmetry Detection

In this and the following section we present two approaches for finding properties in general games that help to reduce the complexity of the search space. The properties are symmetries and independent subgames. Both approaches have in common that they analyse the rules of the game as opposed to the search tree itself, which has the advantage that the analysis is typically quite fast.

Exploiting symmetries of the underlying domain is an important optimisation technique for all kinds of search algorithms. Typically, symmetries increase the search space and thus the cost for finding a solution to the problem exponentially. There is a lot of research on symmetry breaking in domains like CSP [12], Planning [6] or SAT-solving [1]. However, the methods developed in these domains are either limited in the types of symmetries that are handled or are hard to adapt to the general game playing domain because of significant differences in the structure of the problem.

Formally, games can be understood as state machines [17] with the states of the state machine corresponding to the states or positions of the game and transitions in the state machine corresponding to the joint actions of the players. Using this notion of a game, a symmetry is a mapping between states and actions of a game such that the structure of the state machine stays the same. Although theoretically possible, finding and representing the symmetries directly in the state machine is not feasible because the direct representation of the state machine is too large even for simple games. Our approach finds symmetries of the game solely based on the rules of the game.

To find symmetries, we transform the game rules into a so-called rule graph and use standard tools to compute the automorphisms of this graph. These automorphisms correspond to mappings of the symbols (constants, functions, and relations) of the game rules and thus to symmetries of states, actions and roles of the game that are described with these symbols. For example the rule graph in Figure 1 has an automorphism that maps the vertices shown in red to those shown in green and vice versa. This automorphism corresponds to interchanging the x- and y-coordinates of the Tic-Tac-Toe board and thus represents one of the symmetries of the Tic-Tac-Toe game. We showed that the approach is indeed a sound method for detecting

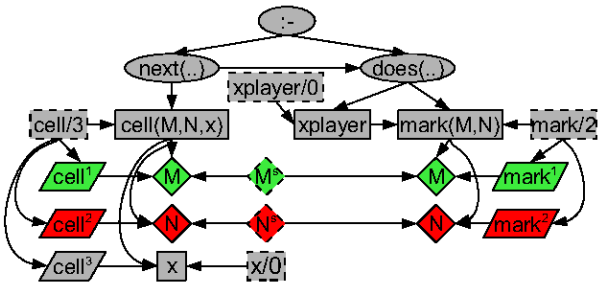

Fig. 1 The rule graph for the rule `next(cell(M,N,x)) :- does(xplayer,mark(M,N))` of Tic-Tac-Toe. Different labels are depicted by different shapes. There is an automorphisms for this graph interchanging the vertices shown in red and green.

symmetries in general games. However, the method is incomplete because it is always possible to write rules with the same semantics but a different structure.

Once we know the symmetries of a game, we can exploit them in different ways depending on the algorithms that we use for playing the game. In [14] we have shown the effectiveness of an extended use of transposition tables to reduce the search space for depth-limited search. In a similar way, symmetries can certainly be exploited for Monte-Carlo simulation based players. We also use the symmetries to reduce the effort for proving properties of games as described in Section 4.

6 Factoring

The general value of factoring has been widely recognised in AI Planning, where it is used to help solve large, complex problems arising in practical settings using a divide-and-conquer strategy [2,3,9]. We address the following two issues: Given its mere rules, how can a previously unknown game be automatically decomposed into independent subgames? And how can a successful decomposition be exploited for a significant improvement of game tree search during play?

6.1 Subgame Detection

The basic idea is to build a *dependency graph* for a given GDL description of a game, consisting of the actions and fluents as vertices and edges between them if a fluent is a precondition or an effect of an action. The *connected components* of this graph then correspond to independent subgames of that particular game.

The main difficulty is to find an easy way to detect preconditions and effects of actions based on the game rules. In [22] we provide formal definitions, which we can informally summarise as follows:

- A move m is called a **noop move** if it is the only legal move of a player when not in control.
- A rule $\texttt{next}(f) :- B$ is a called a **frame axiom** for f if B implies $true(f)$.
- Fluent f is a **potential positive effect** of move m if m is not a noop move and there is a next rule with head $\texttt{next}(f)$ that is compatible with the execution of the action and is not a frame-axiom. Thus, a fluent is a potential positive effect of a move if it could be true in the successor state when the action is executed.
- Fluent f is a **potential negative effect** of move m if m is not a noop move and there is no frame axiom for f that always applies when m is executed. Thus, a fluent is a potential negative effect if there is no rule stating that it will stay true whenever the action is executed.
- Fluent f is a **potential precondition** of move m if f occurs in the body of a game rule with head $\texttt{legal}(p, m)$, or head $\texttt{next}(f')$ where f' is a potential positive or negative effect of m. Therefore, the potential preconditions of a move include all fluents occurring in a `legal` rule for that move and also the fluents that are preconditions of its (conditional) effects.

The `control`-fluent that is used to encode turn-taking in multi-player games typically occurs in the `legal` rules of all actions. We identify and subsequently ignore the `control`-fluent (independent on the actual name) as precondition during the subgame detection in turn-taking games. Otherwise, it would connect all actions in the dependency graph, effectively rendering subgame detection for turn-taking games impossible.

The definitions are weak in the sense that some of the potential effects or preconditions might not be actual effects or preconditions in a game, thus making the subgame detection less effective. However, the advantage of these definitions is that they admit an efficient implementation using a sound but potentially incomplete inconsistency check for GDL formulas.

To illustrate the definitions, consider the well-known game Nim. Nim is a two-player game where the players alternate in removing an arbitrary number (>= 1) of objects from exactly one of the heaps. The game ends when all heaps are empty. The player to take the last object loses the game. The following rules describe an instance of Nim with four heaps (`a,b,c,d`):

```
1  role(player1).
2  role(player2).
3
4  init(heap(a,1)). init(heap(b,2)).
5  init(heap(c,3)). init(heap(d,5)).
6  init(control(player1)).
7
8  legal(W,reduce(X,N)) :- true(control(W)),
9      true(heap(X,M)), smaller(N,M).
10 legal(W,noop) :-
11  true(control(W2)), distinct(W,W2).
12
13 next(heap(X,N)) :- does(W,reduce(X,N)).
14
15 next(heap(X,N)) :- true(heap(X,N)),
16    does(W,reduce(Y,M)), distinct(X,Y).
17 ...
```

In our example, the heaps are represented by the fluent `heap` and the four heaps have initially 1, 2, 3, and 5 objects, respectively (lines 4 and 5). The fluent `control` encodes whose turn it is. The player in control can only choose the action to reduce one of the

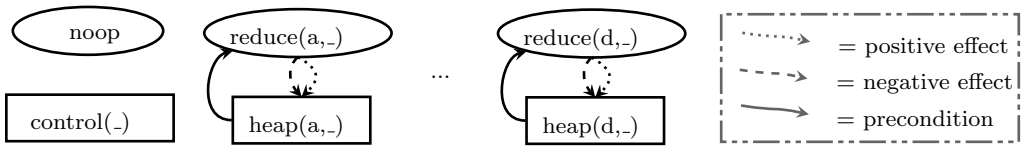

Fig. 2 Dependency graph for the game Nim.

heaps (`X`) to a number of objects `N` (lines 8 and 9). The other player can only do a `noop` move (lines 10 and 11). The next rule in line 13 encodes the positive effect that heap `X` contains `N` objects after execution of the action `reduce(X,N)`. The next rule in lines 15 and 16 is a frame axiom stating that `heap(X,N)` does not change if objects were taken from some other heap. Note that the negative effect of `reduce(X,N)`, namely that heap `X` no longer contains the previous number of objects, is not encoded directly in the rules. It can only be derived indirectly by the absence of a frame axiom for heap `X` that applies when action `reduce(X,N)` is executed.

Applying the definitions for potential effects and preconditions to the rules of Nim, we obtain the dependency graph in Figure 2 with six subgames: one for each heap consisting of the respective `heap`-fluent and `reduce`-action, one consisting of the `control`-fluent, and one for the `noop`-action.

6.2 Solving Decomposable Games

In [8] and [22] we also describe algorithms to solve decomposable games by solving the subgames and composing solutions for the subgames. The main idea is to interleave *subgame search* with *global game search*.

Subgame search scans the game tree of a subgame and returns a subtree where paths that are irrelevant to the solution of the global game are removed. How many of the paths can be ignored depends on the structure of the goal of the game. For example, games with additive goals, i.e., where the goal value of the complete game can be understood as the sum of goal values of all subgames, admit an evaluation of the subgame paths. In that case a path of the same length as another one but with a smaller evaluation can be ignored. In general, paths can be compared based on so-called *local concepts*, i.e., subformulas of the goal and terminal condition that refer only to fluents of one subgame. Paths that lead to the same evaluation of all local concepts of the subgame can be considered equivalent regarding the global game.

The partial game trees of the subgames are then combined by the global game search. Global game search selects moves based only on the subgame-trees returned by the subgame search instead of searching the game tree of the complete game. This leads to an exponentially smaller branching factor unless subgame search returns the complete game tree for every subgame.

Further optimisations of the search algorithm are possible for special classes of games. For example, an algorithm that is able to solve impartial games in linear time is also presented in [22].

References

1. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in the presence of symmetry. In: Design Automation Conference. University of Michigan (2002)
2. Amir, E., Engelhardt, B.: Factored planning. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 929–935 (2003)
3. Brafman, R., Domshlak, C.: Factored planning: How, when and when not. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 809–814 (2006)
4. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1134–1139. AAAI Press, Vancouver (2007)
5. d'Avila Garcez, A.B.K.B., Gabbay, D.: Neural Symbolic Learning Systems. Springer (2002)
6. Fox, M., Long, D.: The detection and exploitation of symmetry in planning problems. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 956–961 (1999)
7. Gelfond, M.: Answer sets. In: F. van Harmelen, V. Lifschitz, B. Porter (eds.) Handbook of Knowledge Representation, pp. 285–316. Elsevier (2008)
8. Günther, M., Schiffel, S., Thielscher, M.: Factoring general games. In: Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA), pp. 27–34 (2009)
9. Kelareva, E., Buffet, O., Huang, J., Thiébaux, S.: Factored planning using decomposition trees. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1942–1947 (2007)
10. Kuhlmann, G., Dresner, K., Stone, P.: Automatic Heuristic Construction in a Complete General Game Player. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence, pp. 1457–62. AAAI Press, Boston (2006)
11. Michulke, D., Thielscher, M.: Neural networks for state evaluation in general game playing. In: ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 95–110. Springer (2009)
12. Puget, J.F.: Automatic detection of variable and value symmetries. In: P. van Beek (ed.) CP, *Lecture Notes*

in Computer Science, vol. 3709, pp. 475–489. Springer (2005)

13. Ruan, J., van der Hoek, W., Wooldridge, M.: Verification of games in the game description language. Journal of Logic and Computation **19**(6), 1127–1156 (2009)
14. Schiffel, S.: Symmetry detection in general game playing. In: Proceedings of AAAI'10, pp. 980–985 (2010)
15. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the National Conference on Artificial Intelligence, pp. 1191–1196. AAAI Press, Vancouver (2007)
16. Schiffel, S., Thielscher, M.: Automated theorem proving for general game playing. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 911–916 (2009)
17. Schiffel, S., Thielscher, M.: A multiagent semantics for the game description language. In: International Conference on Agents and Artificial Intelligence (ICAART). Springer (2009)
18. Sutton, R.S.: Learning to predict by the methods of temporal differences. In: Machine Learning, pp. 9–44. Kluwer Academic Publishers (1988)
19. Tesauro, G.: Temporal difference learning and td-gammon. Communications of the ACM **38**(3), 58–68 (1995)
20. Thielscher, M.: Answer set programming for single-player games in general game playing. In: P. Hill, D. Warren (eds.) Proceedings of the International Conference on Logic Programming (ICLP), *LNCS*, vol. 5649, pp. 327–341. Springer, Pasadena (2009)
21. Thielscher, M., Voigt, S.: A temporal proof system for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1000–1005 (2010)
22. Zhao, D., Schiffel, S., Thielscher, M.: Decomposition of multi-player games. In: Proceedings of the Australasian Joint Conference on Artificial Intelligence, pp. 475–484. Melbourne (2009)

Sebastian Haufe is a PhD Student and Research Associate at Dresden University of Technology, where he received his diploma in Computer Science in 2008. His current research interests are General Game Playing, Logic Programming and Theorem Proving.

Daniel Michulke is a PhD Student at Dresden University of Technology. He received his diploma in Applied Computer Science at Chemnitz University of Technology in 2007. His research interests lie in the field of applied AI, specifically General Game Playing, Hybrid Systems of Logic and Learning, Schema Matching and Agents.

Stephan Schiffel is a PhD Student and Research Associate at Dresden University of Technology where he received his diploma in computer science in 2005. He is the co-author of Fluxplayer, the winner of the AAAI General Game Playing competition 2006. His research interests are General Game Playing, Knowledge Representation and Logic Programming.

Michael Thielscher is an ARC Future Fellow and a Professor at The University of New South Wales in Sydney since 2010. He received his PhD in 1994 and his Habilitation in 1997 from Darmstadt University. He then joined Dresden University of Technology as an Associate Professor before he moved to Australia. His current research is in Systems with General Intelligence, Knowledge Representation, Agents, Cognitive Robotics, and Constraint Logic Programming.