Symmetry Detection in General Game Playing

Stephan Schiffel
Department of Computer Science
Dresden University of Technology
stephan.schiffel@inf.tu-dresden.de

Abstract

We develop a method for detecting symmetries in arbitrary games and exploiting these symmetries when using tree search to play the game. Games in the General Game Playing domain are given as a set of logic based rules defining legal moves, their effects and goals of the players. The presented method transforms the rules of a game into a vertex-labeled graph such that automorphisms of the graph correspond with symmetries of the game. The algorithm detects many kinds of symmetries that often occur in games, e.g., rotation and reflection symmetries of boards, interchangeable objects, and symmetric roles. A transposition table is used to efficiently exploit the symmetries in many games.

1 Introduction

Exploiting symmetries of the underlying domain is an important optimization 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 [Puget, 2005], Planning [Fox and Long, 1999] or SAT-solving [Aloul *et al.*, 2002]. 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.

General game playing is concerned with the development of systems that can play well an arbitrary game solely by being given the rules of the game. This raises a number of issues different from traditional research in game playing, where it is assumed that the rules of a game are known to the programmer. Systems able to play arbitrary, unknown games can't be given game-specific knowledge. They rather need to be endowed with high-level cognitive abilities such as general strategic thinking and abstract reasoning. To exploit symmetries in a general game playing domain, the system must be able to automatically detect symmetries based on the rules of the game. [Banerjee *et al.*, 2006] already exploit symmetries in general games in order to improve learning efficiency. However, they use only rotation and reflection symmetries of boards and do not treat the problem of symmetry exploitation in general.

We present an approach to transform the rules of a game into a vertex-labeled graph such that automorphisms of the graph correspond with symmetries of the game and prove that the approach is sound. The algorithm detects many kinds of symmetries that often occur in games, e.g., rotation and reflection symmetries of boards, interchangeable objects, and symmetric roles. Additionally, we present an extension that can be used for many search algorithms to exploit the discovered symmetries.

2 Games and Symmetries

Games in the general game playing domain are usually modeled as finite state machines. A state of the state machine is a state of the game and actions of the players correspond to transitions of the state machine. In this paper we use the definitions of [Schiffel and Thielscher, 2009] and model a game as a multiagent environment.

Definition (Game (multiagent environment)). *Let Σ be a countable set of ground (i.e., variable-free) symbolic expressions. A (discrete, synchronous, deterministic) game Γ is a structure (R, s_0, t, l, u, g), where*

- $R \subseteq \Sigma$ *finite (the agents, or roles);*
- $s_0 \subseteq \Sigma$ *finite (the initial state);*
- $t \subseteq 2^\Sigma$ *finite (the terminal states);*
- $l \subseteq R \times \Sigma \times 2^\Sigma$ *finite (the action preconditions, or legality relation);*
- $u : (R \mapsto \Sigma) \times 2^\Sigma \mapsto 2^\Sigma$ *finite (the transition function, or update function);*
- $g \subseteq R \times \mathbb{N} \times 2^\Sigma$ *finite (the utility, or goal relation).*

Here, 2^Σ denotes the set of all finite subsets of Σ, and for any $r \in R$ and $s \in 2^\Sigma$, $l(r, a, s)$ holds for finitely many $a \in \Sigma$.

For the sake of simplicity no distinction is made between symbols for roles, objects, state components, actions, etc. States of the game are finite sets of ground symbols. The legality relation $l(r, a, s)$ defines a to be a legal action for role r in state s. The update function u takes an action for each role and (synchronously) applies the joint actions to a current state, resulting in the updated state.

Several kinds of symmetries may be present in such a game, e.g., symmetries of states, moves, roles, and sequences of moves. Intuitively, symmetries of a game can be understood as mappings between objects such that the structure of the game is preserved. E.g., two states of a game are symmetric if the same actions (or symmetric ones) are legal in both states, either both or none of them is a terminal state, for each role both states have the same goal value and executing symmetric joint actions in both states yields symmetric successor states.

Definition (Symmetry)**.** *A mapping $\sigma : \Sigma \mapsto \Sigma$ is a symmetry of a game $\Gamma = (R, s_0, t, l, u, g)$ iff the following conditions hold*

- $r \in R \equiv \sigma(r) \in R$
- $(\forall r, a, s)\ \ l(r, a, s) \equiv l(\sigma(r), \sigma(a), \sigma^s(s))$
- $(\forall A, s)\ \ u(\sigma^a(A), \sigma^s(s)) = \sigma^s(u(A, s))$
- $s \in t \equiv \sigma^s(s) \in t$
- $(\forall r, n, s)\ \ g(\sigma(r), n, \sigma^s(s)) \equiv g(r, n, s)$

Here, $\sigma^s(s) \overset{def}{=} \{\sigma(x) | x \in s\}$ *and* $\sigma^a(A) \overset{def}{=} \{(\sigma(r), \sigma(a)) | (r, a) \in A\}$*. We will omit the superscripts in the rest of the paper.*

A symmetry of a game expresses role, state, and action symmetries at the same time. That means there can be a symmetry σ of a game with $\sigma(s_1) = s_2$ and $\sigma(r_1) = r_2$ which means that a state s_1 is symmetric to a state s_2, but only if the roles r_1 and r_2 are swapped. This is not what one would expect the term "symmetric states" to mean. Therefore, we give more intuitive definitions for symmetric states and joint actions here.

Definition (Symmetric States)**.** *Let $\Gamma = (R, s_0, t, l, u, g)$ be a game with symbolic expressions Σ. A symmetry σ of Γ is a state symmetry of Γ iff $(\forall r \in R)\sigma(r) = r$.*

Two states $s_1, s_2 \subseteq \Sigma$ are called symmetric if there is a state symmetry σ of Γ with $\sigma(s_1) = s_2$

That means a state symmetry maps each role to itself and two states are symmetric if there is a symmetry mapping one state to the other without affecting the roles.

Definition (Symmetric Actions)**.** *Let $\Gamma = (R, s_0, t, l, u, g)$ be a game with symbolic expressions Σ. Two joint actions $A_1, A_2 \in 2^{(R \mapsto \Sigma)}$ are called symmetric in a state $s \subseteq \Sigma$ iff there is a state symmetry σ of Γ with $\sigma(s) = s$ and $\sigma(A_1) = A_2$*

Since the result of an actions depend on the state they are applied in, it only makes sense to talk about symmetric actions wrt. one particular state. From the definition of symmetry it follows that the states resulting from the execution of two symmetric joint actions are symmetric.

Notice that, the symmetries of a game are independent of the initial state. As a consequence, all games that only differ in the initial state have the same set of symmetries. We call such games instances of each other.

Definition (Game Instances)**.** *Let $\Gamma = (R, s_0, t, l, u, g)$ be a game with symbolic expressions Σ. A game $\Gamma' = (R, s'_0, t, l, u, g)$ is called an instance of Γ with initial state s'_0 if $s'_0 \subseteq \Sigma$.*

So far, the games that are considered in General Game Playing are finite. Thus, it is in principle possible to encode games directly as lists and tables. However, because of the size of such a representation the description is given in a modular fashion. The language that is used is the Game Description Language[Love *et al.*, 2008] (GDL). The GDL is an extension of Datalog with functions, equality, some syntactical restrictions to preserve finiteness, and some predefined keywords.

The following is a partial encoding of a Tic-tac-toe game. We use Prolog syntax where words starting with uppercase letters stand for variables and the remaining words are constants.

```
 1 role(xplayer). role(oplayer).
 2
 3 init(cell(a,1,blank)). init(cell(a,2,blank)).
 4 init(cell(a,3,blank)). init(cell(b,1,blank)).
 5 init(cell(b,2,blank)). init(cell(b,3,blank)).
 6 init(cell(c,1,blank)). init(cell(c,2,blank)).
 7 init(cell(c,3,blank)).
 8 init(control(xplayer)).
 9
10 legal(P, mark(X, Y)) :-
11   true(control(P)), true(cell(X, Y, blank)).
12 legal(P, noop) :-
13   role(P), not true(control(P)).
14
15 next(cell(M, N, x)) :-
16   does(xplayer, mark(M, N)).
17 next(cell(M, N, o)) :-
18   does(oplayer, mark(M, N)).
19 next(cell(M, N, C)) :-
20   true(cell(M, N, C)),
21   does(P, mark(X, Y)),
22   (X \= M ; Y \= N).
23
24 goal(xplayer, 100) :- line(x).
25 goal(xplayer, 50) :- not line(x), not line(o).
26 goal(xplayer, 0) :- line(o).
27 goal(oplayer, 100) :- line(o).
28 goal(oplayer, 50) :- not line(x), not line(o).
29 goal(oplayer, 0) :- line(x).
30
31 terminal :- line(x) ; line(o) ; not open.
32
33 line(P) :- true(cell(X, 1, P)),
34   true(cell(X, 2, P)), true(cell(X, 3, P)).
35 line(P) :- true(cell(a, Y, P)),
36   true(cell(b, Y, P)), true(cell(c, Y, P)).
37 line(P) :- true(cell(a, 1, P)),
38   true(cell(b, 2, P)), true(cell(c, 3, P)).
39 line(P) :- true(cell(a, 3, P)),
40   true(cell(b, 2, P)), true(cell(c, 1, P)).
41
42 open :- true(cell(X, Y, blank)).
```

The first line declares the roles of the game. Next the initial state of the game is defined. The unary predicate `init` defines the properties that are true in the initial state. Lines 10-13 define the legal moves of the game, where `mark(X,Y)` is a legal move for role `P` if `control(P)` is true in the current state (i.e., it's `P`'s turn) and the cell `X,Y` is blank (`cell(X,Y,blank)`). If it is not `P`'s turn then `noop` is the only legal move for `P`. The rules for predicate `next` define the properties that hold in the successor state, e.g., `cell(M,N,x)` holds if `xplayer` marked the cell `M,N`. Lines 24 to 29 define the rewards of the players and the condition for terminal states. The rules for both contain auxiliary predicates `line(P)` and `open` which encode the concept of a line-of-three and the existence of a blank cell, respectively. The remaining rules are those for `line(P)` and `open`.

Besides the keywords, which are printed in bold face, all predicates, functions, and constants of the game description are game specific and do not carry a special meaning. That means, replacing any of them by some other word consistently in the whole game description does not change the game.

A game description in GDL is a set of GDL rules. In the following we will interpret this as a set of clauses. The game for a game description is the multiagent environment that is its semantics. We repeat Definition 6 of [Schiffel and Thielscher, 2009] here for the sake of completeness.

Definition (Game for a game description)**.** *Let D be a valid GDL game description, whose signature determines the set of ground terms Σ. The game for D is the game (R, s_0, t, l, u, g), where*

- $R = \{r \in \Sigma | D \models role(r)\}$
- $s_0 = \{f \in \Sigma | D \models init(f)\}$
- $t = \{s \in 2^\Sigma | D \cup s^{true} \models terminal\}$
- $l = \{(r, a, s) \in R \times \Sigma \times 2^\Sigma | D \cup s^{true} \models legal(r, a)\}$
- $u(A, s) = \{f \in \Sigma | D \cup A^{does} \cup s^{true} \models next(f)\}$
- $g = \{(r, n, s) | r \in R, n \in \mathbb{N}, s \in 2^\Sigma, D \cup s^{true} \models goal(r, n)\}$

Here $s^{true} \overset{def}{=} \{true(f) | f \in s\}$ axiomatizing s as the current state and $A^{does} \overset{def}{=} \{does(r, a) | r \in R, a = A(r)\}$ axiomatizes A as a joint action.

3 Rule Graphs

It is not a new idea to use graph automorphisms to compute symmetries of a problem. This approach has been successfully applied to CSPs[Puget, 2005] and SAT solving[Aloul *et al.*, 2002], among others. However, a key for using this method is to have a graph representation of the problem such that the graph has the same symmetries.

Unrelated to symmetries in games, [Kuhlmann and Stone, 2007] describes a mapping of GDL game descriptions to so called "rule graphs" such that two rule graphs are isomorphic if and only if the game descriptions are scramble equivalent. A scrambling of a game description is defined as a one-to-one mapping between constants and a one-to-one mapping between variables of the game. That means, scramble-equivalent game descriptions are identical up to renaming of non-keyword constants and variables. Basically, rule graphs contain vertices for all predicates, functions, constants, and variables in the game and connections between them that match the structure of the rules. The nodes of rule graphs are colored such that isomorphisms can only map constants to other constants, variables to variables, etc.

We argue that these graphs can be used to compute symmetries of games. If there is an automorphisms of such a rule graph, that means an isomorphism of the graph to itself, then there is a scrambling of the game description that does not change the rules of the game. Since constants of the game description refer to objects in the game, a mapping between constants that does not change the rules describes configurations of objects that are interchangeable in the game. Figure 1 shows the partial game description of a simple blocks world domain. The two blocks `a` and `b` have the same properties initially and do not occur in any other rule. Therefore, they are interchangeable. It can be seen that substituting `a` for `b` and `b` for `a` simultaneously in the game description only changes the order of the rules, which is unimportant since the game description is considered to be a set of rules.

```
role(player).
init(clear(a)). init(clear(b)).
legal(player, stack(X, Y)) :-
  true(clear(X)), true(clear(Y)).
next(on(X, Y)) :- does(player, stack(X, Y)).
goal(player, 100) :- true(on(X, Y)).
...
```

Figure 1: In this simple blocks world domain, the goal is to stack two arbitrary blocks on each other.

Since roles of a game and coordinates of a board game are typically encoded by constants, too, automorphisms of the rule graph also reflect reflection symmetries of boards (swapping of coordinates) and symmetric roles. E.g., in the Tic-tac-toe example above, swapping of `a` and `c` or `1` and `3` corresponds to horizontal or vertical reflection of the board, respectively. However, rotation symmetry of the board can not be expressed by a mapping between constants of the game, if the typical representation of a board, `cell(X,Y,_)`, is used. A rotation of the Tic-tac-toe board by 90 degrees would correspond to swapping of the row and column argument of the `cell`-function and the mapping between the coordinates $\{1 \mapsto \mathtt{a}, 2 \mapsto \mathtt{b}, 3 \mapsto \mathtt{c}, \mathtt{a} \mapsto 3, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 1\}$. The rule graphs from [Kuhlmann and Stone, 2007] do not allow this kind of mapping.

Therefore, we propose enhanced rule graphs, which differ from the rule graphs from [Kuhlmann and Stone, 2007] mainly by replacing the ordering edges between arguments with argument index vertices.

Definition (Enhanced Rule Graph)**.** *Let D be a valid GDL game description. The enhanced rule graph of D is the smallest vertex labeled graph $G = (V, E, l)$ with the following properties:*

- *If $v = \mathtt{h:-b_1, \ldots, b_n}$ is a rule in D then $v \in V$, $(v, h), (v, b_1), \ldots, (v, b_n), (h, b_1), \ldots, (h, b_n) \in E$ and $l(r) = \mathtt{rule}$.*
- *If a negation $v = \mathtt{not\ a}$ occurs in a rule in D then $v \in V$, $(v, a) \in E$ and $l(n) =$ not.*
- *If an atom $v = \mathtt{p(t_1, \ldots, t_n)}$ occurs in a*

rule in D and p *is* init, true, next, does, legal, role, goal, *or* terminal *then $v \in V$, $(v,t_1),\dots,(v,t_n),(t_1,t_2),\dots,(t_{n-1},t_n) \in E$ and $l(v) = $* p.

- *If an atom $v = (\mathtt{t_1} \neq \mathtt{t_2})$ occurs in a rule in D then $v \in V$, $(v,t_1),(v,t_2) \in E$ and $l(v) = \neq$.*
- *If an atom $v = \mathtt{p}(\mathtt{t_1},\dots,\mathtt{t_n})$ occurs in a rule in D and p is not a GDL keyword then $v \in V$, $(v,t_1),\dots,(v,t_n),(p,v),(p^1,t_1),\dots,(p^n,t_n) \in E$ and $l(v) = $* predicate.
- *If a function $v = \mathtt{f}(\mathtt{t_1},\dots,\mathtt{t_n})$ occurs in a rule in D then $v \in V$, $(v,t_1),\dots,(v,t_n)$, (f,v), $(f^1,t_1),\dots,(f^n,t_n) \in E$ and $l(v) = $* function.
- *If a variable v occurs in a rule in D then $v \in V$, $l(v) = $* variable.
- *For every n-ary non-keyword relation symbol or function symbol p in D, $p/n, p^1,\dots,p^n \in V$, $(p,p^1),\dots,(p,p^n) \in E$, $l(p) = \mathtt{symbol_{const}}$, and $l(p^i) = $* arg *for $i \in [1,n]$.*
- *For every variable symbol v in D, $v^s \in V$, and $l(v) = \mathtt{symbol_{var}}$.*

Constants are treated as null-ary functions.

In the definition we considered only game descriptions without disjunctions and where only atomic formulas are allowed to occur negated. Variables in different clauses should be named differently. Every game description can be easily transformed into an equivalent one which meets these requirements. There is a vertex v in the rule graph for every formula and term in a game description. Additionally there are vertices for every relation symbol and function symbol (vertices p and f). The vertex of a relation symbol is connected to every vertex for an atom of this relation symbol and the vertex of a function symbol is connected to every vertex for function term with this function symbol. Furthermore for every argument position i of relation symbol p (or function symbol f) there is a vertex p^i (or f^i) which is connected to every term that occurs in the i-th argument of p (or f) somewhere in the game description. Note that every occurrence of an atom or term is treated as a different atom or term. That means if the same term occurs twice in the rules there are two vertices, one for each occurrence.

In figure 2 you can see the (enhanced[1]) rule graph for the following rule of Tic-tac-toe:

```
next(cell(M, N, x)) :-
  does(xplayer, mark(M, N)).
```

4 Theoretic Results

If we use the rule graph of the complete game description to compute symmetries, we only get symmetries that are present in the initial state of the game. However, there may be so called "dynamic symmetries", i.e., symmetries that occur only in some states of the game but are not present in the initial state. To also find these symmetries, the initial state of the game must not be part of the rule graph.

[1]In the remainder of the paper we write "rule graph" instead of "enhanced rule graph". All results apply to enhanced rule graphs.

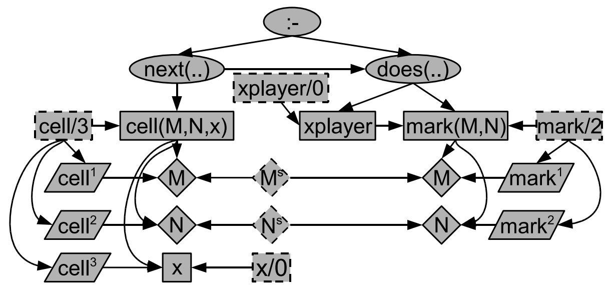

Figure 2: Rule graph for `next(cell(M, N, x)) :- does(xplayer, mark(M, N))`. Different labels are depicted by different shapes, relation vertices by ellipses, function vertices by rectangles, variable vertices by diamonds. The respective symbol vertices have dashed instead of solid lines. Argument index vertices are parallelograms.

Definition (State-independent game description)**.** *Let D be a game description. The state-independent game description $D' = D \setminus \{init(F) \in D\}$. That means D' contains all the rules of D except for the initial state description. The rule graph G of D' is called the state-independent rule graph of D.*

It is clear, that game descriptions that differ only in the initial state have the same state-independent rule graph. Therefore, a state-independent rule graph includes symmetries of all instances of a game.

Definition (Game description for a state)**.** *Let D be a game description, Γ be a game for D, and D' be the state-independent game description of D. The game description of a state s of Γ is defined as $D(s) \overset{def}{=} D' \cup s^{true}$. The rule graph of $D(s)$ is called the rule graph for state s.*

Intuitively, a rule graph for a state s only contains the symmetries of a game that map s to itself.

In order to reflect the reordering of arguments we extend the definition of a scrambling of a game description from [Kuhlmann and Stone, 2007].

Definition (Scrambling of a game description)**.** *A scrambling of a game description D is a one-to-one function over function, relation and variable symbols and argument indices of function symbols and non-keyword relation symbols in D.*

Our first theorem describes the connection between automorphisms of rule graphs and scramblings of game description.

Theorem 1 (Scramblings and Automorphisms)**.** *Let D be a game description, $G = (V,E,l)$ its rule graph and H the set of automorphisms of G. Let*

$$\begin{aligned} h_1 \sim h_2 \overset{def}{=} \\ & (\forall v \in V)\ l(v) \in \{\mathtt{symbol_{const}}, \mathtt{symbol_{var}}, \mathtt{arg}\} \\ & \qquad \supset h_1(v) = h_2(v) \end{aligned}$$

That means two automorphisms h_1, h_2 of H are considered equivalent exactly if they agree on the mapping of all symbol vertices and argument index vertices. There is a one-to-one mapping between the quotient set $H/\sim$ and scramblings that map D to D.

Proof Sketch. *The rule graph construction algorithm adds exactly one symbol label vertex to the graph for each symbol in D, and exactly one argument index vertex for each argument index of all function symbols and non-keyword relation symbols in D. That means, there are one-to-one mappings between symbols of D and symbol vertices V_s, and between argument indices and argument index vertices V_a.*

The remaining proof consists of two parts. First, it is shown that for each automorphism h of G there is indeed a scrambling of D, which corresponds to h via the mappings defined by the rule graph construction. The proof follows the structure of the proof in [Kuhlmann and Stone, 2007] but is adapted to deal with argument reorderings.

For the inverse direction it is clear that a scrambling of D determines one-to-one mappings over V_s and V_a. Based on the edges which are in the rule graph it can be shown that a set of equivalent automorphisms of G (equivalent wrt. $\sim$) is completely determined by these mappings.

One implication of the theorem is that we can compute all scramblings mapping a game description to itself by computing all automorphisms of its rule graph.

Theorem 2 (Symmetries and Automorphisms). *Let D be a game description, $\Gamma = (R, s_0, t, l, u, g)$ the game for D, D' the state independent game description for D, G' the state independent rule graph of D, h be an automorphism of G', and σ' be a scrambling of D' corresponding to h. Intuitively, σ' can be understood as a bijective mapping between arbitrary terms of D'. Now, let σ be a bijective extension of σ' to all terms of D. That means σ agrees with σ' on the mapping of all terms of D' and may map the remaining terms of D arbitrarily without violating bijectivity.*

Then σ is a symmetry of Γ corresponding to h.

Proof Sketch. *The proof uses the construction of the game Γ from the game description D to show that σ satisfies the defining properties of a symmetry.*

E.g., one has to prove that $s \in t \equiv \sigma(s) \in t$, where t is the set of terminal states of Γ. By the construction of a game from a game description $\sigma(s) \in t$ is equivalent to

$$D \cup \{true(f) | f \in \sigma(s)\} \models terminal$$

This is equivalent to

$$D \cup \sigma(\{true(f) | f \in s\}) \models terminal$$

because true is a keyword and keywords are mapped to themselves by σ. Now since $\sigma(D) = D$ and terminal is a keyword, this is equivalent to

$$D \cup s^{true} \models terminal,$$

which is equivalent to $s \in t$ by definition.

The remaining properties of a symmetry can be shown in a similar fashion.

Observe that, the game description D can contain function symbols or constants that are not included in the state-independent game description D'. If so, these symbols are only part of the initial state description and do not occur anywhere else in the rules of the game. Therefore, they refer to objects of the game that are interchangeable and can be arbitrarily mapped to each other by the symmetry.

In order to use the approach for computing only state symmetries of a game or only symmetries of a particular state of the game only minor changes have to be made to the rule graph. To compute only state symmetries it is necessary to restrict the automorphisms of the rule graph in such a way that constants denoting roles can only be mapped to themself. This can be achieved by assigning each node in the rule graph that belongs to a `role`-fact a different label. Symmetries of a particular state s, like symmetric actions in a state, can be computed by using the rule graph for state s to compute the symmetries.

5 Exploiting Symmetries

There are several ways to exploit symmetries in game tree search. One option is to prune symmetric joint actions in node expansion. It is clear that it is sufficient to use only one joint action of each set of symmetric joint actions in a state for node expansion because symmetric joint actions lead to symmetric states and yield the same value in game tree search.

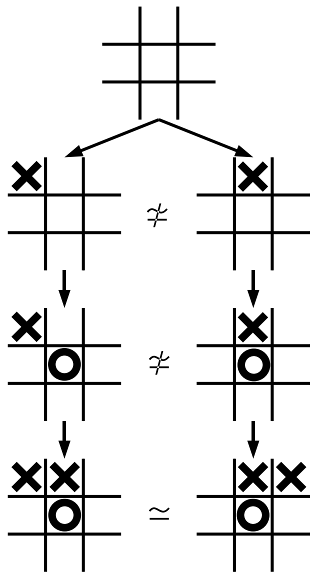

Figure 3: Two non-symmetric action sequences in Tic-tac-toe that lead to symmetric states.

However, this does not use the full potential of the available information. In particular, it does not completely avoid the expansion of both states of a pair of symmetric states. E.g., there may be two non-symmetric sequences of joint actions leading to symmetric states. The expansion of the second state is not avoided since the action sequences are not sym-

Game	depth	runtime (in s)	# states	# symmetries
8puzzle	15	81	389286	1
asteroidsparallel	7	375	1460896	127
asteroids	15	356	2575774	7
chinesecheckers1	11	24	63761	1
chinesecheckers2	7	66	287483	1
connect4	6	147	94372	1
knightmove	7	81	168849	7
knightthrough	4	1071	866625	1
peg	7	265	109154	7
pentago	5	312	661145	7
tictactoeparallel	4	80	117918	127
tictactoe	9	1.2	5478	7
tictictoe	9	8.3	12829	7

Figure 4: This table shows the depth-limit that we used for each game, the average runtime of the normal search, the number of states expanded by the normal search and the number of symmetries found in the game.

metric. An example is shown in figure 3. A common reason for this are transpositions. E.g., in our formulation of Tic-tac-toe the order in which the actions are executed is unimportant for the resulting state. Therefore, every transposition of a symmetric action sequence also leads to a symmetric state.

We propose to use a transposition table to detect those symmetric states before expanding a node. That means before we evaluate or expand a state in the game tree we check whether this state or any state that is symmetric to this one has an entry in the transposition table. If so, we just use the value stored in the transposition table without expanding the state. It is clear, that the algorithm does not use any additional memory compared to normal search. On the contrary, the transposition table may get smaller because symmetric states are not stored. However, the time for node expansion is increased by the time that it takes to compute the symmetric states and check whether some symmetric state is in the transposition table.

Therefore, it is essential to be able to compute hash values of states and symmetric states very efficiently. We use zobrist hashing [Zobrist, 1970] where each ground fluent is mapped to a randomly generated hash value and the hash value of a state is the bit-wise exclusive disjunction of the hash values of its fluents. For efficiently computing symmetric states all ground fluents are numbered consecutively and the symmetry mappings are tabulated for the fluents. In our implementation the time to compute all symmetric states for some a state depends on the game and ranges from $\frac{1}{50}$ to 3 times the time for expanding a state for the 13 games we tried. The time depends on the complexity of the `legal` and `next` rules, the size of the state and the number of symmetries in the game.

We conducted experiments on a selection of games where we measured the time it took to do a depth-limited search in every state on a path through the game. We compared normal search with a transposition table but without checking for symmetric states ("normal search"), the approach where only symmetric moves were pruned ("prune symmetric moves") and the approach where we check all symmetric states before expanding a state ("check symmetric states"). Figure 4 shows runtimes of the "normal search" with and without heuristic evaluation and the depth limits we used for the games. In figure 5 the time savings for search with symmetry pruning compared to "normal search" are shown.

It can be seen that for the majority of games exploiting the symmetries improves the performance. Also, in most cases the additional effort of transposition table look-up for all symmetric states pays off compared to pruning only symmetric moves. This is not too surprising because for pruning symmetric moves in a state we have to compute the symmetries that map the state to itself. In many cases this is only slightly faster then computing all symmetric states.

For some games the overhead of checking for symmetric states is higher than the gain, most notably asteroidsparallel, which is just two instances of asteroids played in parallel. The bad result has several reasons. One problem is that because of the rather large number of symmetries, computing all symmetric states is quite expensive. Another reason is that because of the large branching factor the depth-limit is relatively small, but many symmetric states can only be reached after longer action sequences. The same is true for knightmove, where it is hardly possible to reach symmetric states with seven moves. Additionally, because of the very simple rules of the game, computing state expansion is fast compared to computing symmetric states. For tictactoe and tictictoe the results are near optimal. Because every symmetric state is indeed reachable from the initial state and the complete game tree was searched about $\frac{7}{8} = 87.5\%$ of the states were not explored.

It should be noted that the experiments were run with blind search, i.e., without a heuristic evaluation of non-terminal leaf nodes. If heuristic search is used the saved time is increased by the saved heuristic evaluation time, which may be considerable, depending on the complexity of the heuristic function. In our game player that means that even in games like 8puzzle and pentago exploiting symmetries pays off.

In order to avoid big negative impact like in asteroidsparallel or knightmove we keep track of the number n_{saved} of saved state expansions due to symmetry checking by counting the state expansions in each subtree during search and storing this number for each state in the transposition table. Whenever a symmetric state is found in the transposition table we can add the stored number to the number of

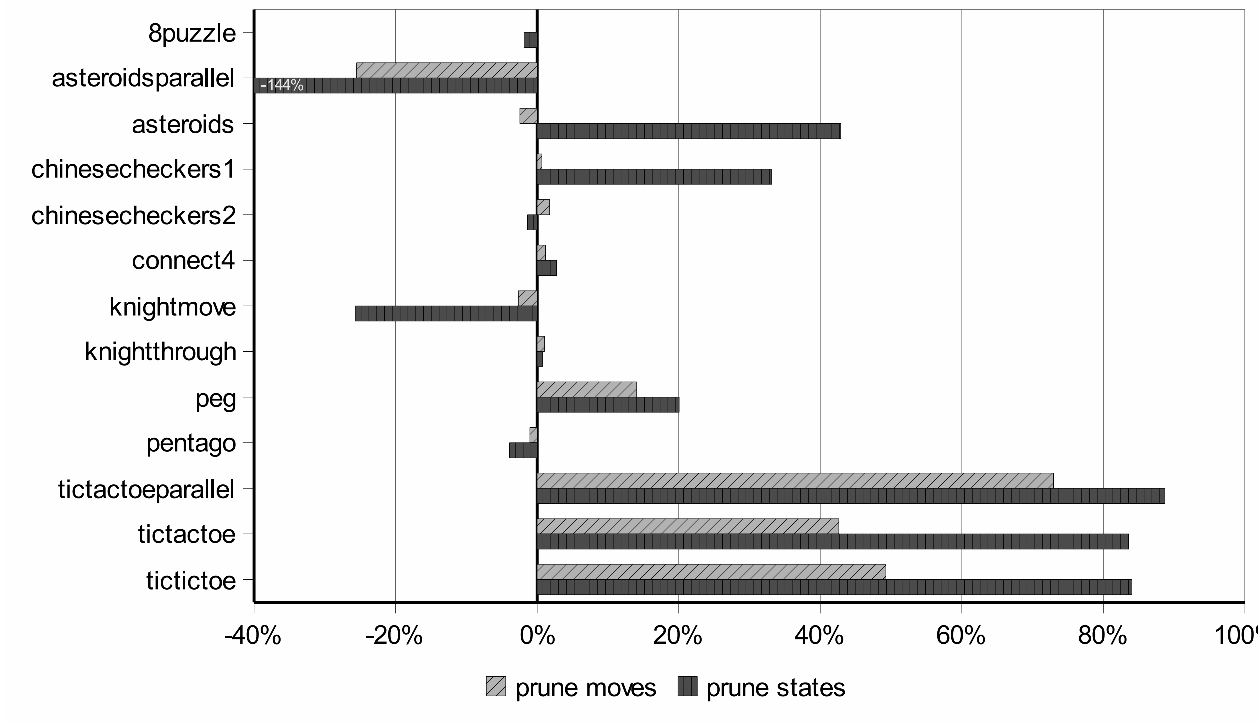

Figure 5: The chart shows the time savings of using search with pruning symmetric moves and pruning symmetric states compared to normal search, i.e., without using any symmetry information.

n_{saved}. Based on n_{saved} we compute an estimate of the saved time $t_{saved} = n_{saved} * t_{exp} - n_{total} * t_{sym}$, where t_{exp} is the average time for expanding a state, n_{total} is the total number of expanded states and t_{sym} is the average time for computing and checking all symmetric states for a state. If $t_{saved} < -t_{limit}$ we switch to normal search thereby limiting the negative impact to t_{limit}.

6 Discussion

The presented method can be used to detect and exploit many symmetries that often occur in games, e.g.,

- object symmetries (functionally equivalent objects),
- configuration symmetries (symmetries between collections of objects and their relations to each other), and
- action symmetries (actions leading to symmetric states).

This includes the typical symmetries of board games, like rotation, and reflection, as well as symmetric roles.

None of the tested games contained object symmetries. This type of symmetries leads to a number of symmetries exponential in the number of functionally equivalent objects and should therefore be handled more efficiently then with our approach. The method described in [Fox and Long, 2002] for planning can be easily adapted to the general game playing domain. Plan permutation symmetries, that are exploited in e.g., [Long and Fox, 2003], are not to be confused with our symmetric action sequences. Symmetric plan permutations are permutations of a plan that lead to the same state, whereas symmetric action sequences are sequences of element-wise symmetric joint actions. Plan permutation symmetries are typically exploited in a game playing program by a transposition table without any symmetry detection. To our knowledge no previous approach exist that can exploit symmetries in general games.

Because the symmetry detection is based on the game description instead of the game graph itself, it can only detect symmetries that are apparent in the game description. Since syntactically different rules can have the same semantics, symmetry detection based on different game descriptions for the same game may lead to different results. Even adding a tautological rule to a game description may cause some symmetries to not be detected. E.g., adding the following rule to the blocks world example from figure 1, would mean that substituting `a` for `b` and `b` for `a` results in different game description. Therefore the symmetry between `a` and `b` wouldn't be detected.

```
next(clear(a)) :- a\=a.
```

Consequently, our approach may benefit from removing superfluous rules and transforming the game description to some normal form.

Another limitation of the approach is that does not allow to map arbitrary terms to each other. E.g., the approach can not detect the symmetry in a variant of the blocks world domain, where we rename `a` to `f(1)`, because an automorphism only maps single vertices to each other but `f(1)` is not represented by a single vertex in the rule graph. It is in principle possible to overcome this limitation by propositionalizing a game description. The resulting rule graphs would be very similar to propositional automata and could in addition be used to improve reasoning speed[Schkufza *et al.*, 2008]. However, this is only feasible for small games because the ground representation of the game rules can be exponentially larger than the original one. Not only does propositionalizing of large game descriptions take valuable time, but computing automorphisms of the resulting large rule graphs is also more expensive. Therefore, we are working on partially grounding the game rules in order to limit the size of the description but still benefit from the advantages of propositional representations when possible.

7 Summary

We presented a method to compute symmetries of a game whose rules a given in the Game Description Language (GDL). Symmetries are computed by transforming the rules of the game into a vertex-colored graph and computing automorphisms of this graph. Depending on the game description our method is able to detect many of the typical symmetries that occur in games and planning problems. Additionally, we presented an approach that is able to exploit the detected symmetries efficiently in many games.

References

[Aloul *et al.*, 2002] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Design Automation Conference*. University of Michigan, June 2002.

[Banerjee *et al.*, 2006] Bikramjit Banerjee, Gregory Kuhlmann, and Peter Stone. Value function transfer for general game playing. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[Fox and Long, 1999] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 956–961, 1999.

[Fox and Long, 2002] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Proceedings of AIPS'02*, pages 83–91, 2002.

[Kuhlmann and Stone, 2007] Gregory Kuhlmann and Peter Stone. Graph-based domain mapping for transfer learning in general games. In *Proceedings of The Eighteenth European Conference on Machine Learning*, September 2007.

[Long and Fox, 2003] D. Long and M. Fox. Symmetries in planning problems. In *Proceedings of SymCon'03 (CP Workshop)*, 2003.

[Love *et al.*, 2008] D. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. *General Game Playing: Game Description Language Specification.* Stanford Logic Group, Stanford University, 353 Serra Mall, Stanford, CA 94305, March 2008.

[Puget, 2005] Jean-Francois Puget. Automatic detection of variable and value symmetries. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 2005.

[Schiffel and Thielscher, 2009] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In *International Conference on Agents and Artificial Intelligence (ICAART)*. Springer, 2009.

[Schkufza *et al.*, 2008] Eric Schkufza, Nathaniel Love, and Michael R. Genesereth. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Australasian Conference on Artificial Intelligence*, volume 5360, pages 56–66. Springer, 2008.

[Zobrist, 1970] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin, April 1970.