Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)

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.

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 & 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 not 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.

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. Furthermore, we present an extension for search algorithms that exploits the symmetries to prune the search space.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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 & 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 (terms), $\mathcal{S}$ a set of states, and $\mathcal{A}$ a set of actions. 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);*
- $s_0 \in \mathcal{S}$ *(the initial state);*
- $t \subseteq \mathcal{S}$ *finite (the terminal states);*
- $l \subseteq R \times \mathcal{A} \times \mathcal{S}$ *finite (the action preconditions);*
- $u : (R \mapsto \mathcal{A}) \times \mathcal{S} \mapsto \mathcal{S}$ *finite (the update function);*
- $g \subseteq R \times \mathbb{N} \times \mathcal{S}$ *finite (the utility, or goal relation).*

For the sake of simplicity no distinction is made between symbols for roles, objects, state components, actions, etc. That means, we define actions as ground terms $\mathcal{A} \stackrel{def}{=} \Sigma$ and states as finite sets of ground terms $\mathcal{S} \stackrel{def}{=} 2^{\Sigma}$. 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, $\sigma^s(s) \stackrel{def}{=} \{\sigma(x)|x \in s\}$ and $\sigma^a(A) \stackrel{def}{=} \{(\sigma(r), \sigma(a))|(r, a) \in A\}$. 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 and Actions). *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 are called symmetric if there is a state symmetry σ with $\sigma(s_1) = s_2$. Two joint actions A_1, A_2 are called symmetric in a state $s \subseteq \Sigma$ iff there is a state symmetry σ of Γ with $\sigma(s) = s$ and $\sigma(A_1) = A_2$.*

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. 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.

Rules of games in General Game Playing are typically described in the Game Description Language(Love *et al.* 2008) (GDL). 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 init(cell(a,1,blank)). init(cell(a,2,blank)).
3 init(cell(a,3,blank)). ...
4 init(cell(c,3,blank)). init(control(xplayer)).
5 legal(P,mark(X,Y)) :-
6   true(control(P)), true(cell(X,Y,blank)).
7 legal(P,noop) :- role(P), not true(control(P)).
8 next(cell(M,N,x)) :- does(xplayer,mark(M,N)).
9 next(cell(M,N,o)) :- does(oplayer,mark(M,N)).
10 next(cell(M,N,C)) :- true(cell(M,N,C)),
11   does(P,mark(X,Y)), (X \= M ; Y \= N).
12 goal(xplayer,100) :- line(x).
13 ...
14 terminal :- line(x) ; line(o) ; not open.
15 line(P) :- true(cell(a,Y,P)),
16   true(cell(b,Y,P)), true(cell(c,Y,P)).
17 ...
18 open :- true(cell(X,Y,blank)).
```

The first line declares the roles of the game. The unary predicate `init` defines the properties that are true in the initial state. Lines 5-7 define the legal moves of the game, e.g., `mark(X,Y)` is a legal move for role `P` if it is `P`'s turn (`control(P)`) and the cell `X,Y` is blank (`cell(X,Y,blank)`). 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 12 to 14 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. 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.

In the following we will interpret a GDL game description as a set of clauses. The game for a game description is the multiagent environment that is its semantics.

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*

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

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

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 & 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 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. E.g., it can easily be seen that consistently interchanging the constants `a` and `c` (or `1` and `3`) in the rules of Tic-tac-toe above yields the same set of rules which means that the "objects" referred to by these constants are interchangeable. Because in this example the objects stand for coordinates of a board, 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 board would correspond to a suitable mapping between the coordinates plus the swapping of the row and column argument of the `cell`-function. The rule graphs from (Kuhlmann & Stone 2007) do not allow this kind of mapping. Therefore, we propose enhanced rule graphs, which differ from the rule graphs from (Kuhlmann & 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:*

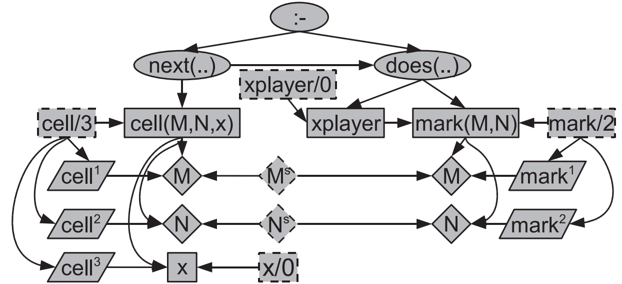
- *For every n-ary non-keyword relation symbol or function symbol p in D, $p/n, p^i \in V$, $(p, p^i) \in E$, $l(p) =$ $\mathtt{symbol_{const}}$, and $l(p^i) = \mathtt{arg}$ for $i \in [1, n]$.*
- *For every variable symbol v in D, $v^s \in V$, and $l(v) =$ $\mathtt{symbol_{var}}$.*

Furthermore, for every part v of D:

- *If $v = \mathtt{h} :- \mathtt{b_1}, \ldots, \mathtt{b_n}$ is a rule then $v \in V$, $(v, b_1), \ldots$, (v, b_n), (v, h), $(h, b_1), \ldots, (h, b_n) \in E$ and $l(r) = \mathtt{rule}$.*
- *If $v = \mathtt{not\ a}$ is a negative literal then $v \in V$, $(v, a) \in E$ and $l(n) =$ not.*
- *If $v = \mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is an atom and $\mathtt{p}$ is a keyword (`true`, `does`, `legal`, ...) then $v \in V$, $(v, t_1), \ldots, (v, t_n)$, $(t_1, t_2), \ldots, (t_{n-1}, t_n) \in E$ and $l(v) = \mathtt{p}$.*
- *If $v = (\mathtt{t_1} \neq \mathtt{t_2})$ then $v \in V$, $(v, t_1), (v, t_2) \in E$ and $l(v) = \neq$.*
- *If $v = \mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is an atom and p is not a GDL keyword then $v \in V$, $l(v) =$ `predicate` and $(v, t_1), \ldots, (v, t_n), (p, v), (p^1, t_1), \ldots, (p^n, t_n) \in E$.*
- *If $v = \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is a function then $v \in V$, $(v, t_1), \ldots, (v, t_n)$, (f, v), $(f^1, t_1), \ldots, (f^n, t_n) \in E$ and $l(v) =$ `function`.*
- *If v is a variable then $v \in V$, $(v^s, v) \in E$ and $l(v) =$ `variable`.*

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 and function symbol (vertices p and f), where constants are treated as nullary functions. 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 the following figure you can see the (enhanced[1]) rule graph for the rule **`next`**`(cell(M,N,x)) :-` **`does`**`(xplayer,mark(M,N))`. Different labels are depicted by different shapes.



Theoretic Results

Our first theorem describes the connection between automorphisms of rule graphs and scramblings of game description. In order to reflect the reordering of arguments we extend the definition of a scrambling of a game description from (Kuhlmann & 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.*

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 $h_1 \sim h_2 \stackrel{def}{=}$*

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

$l(v) \in \{\mathtt{symbol_{const}}, \mathtt{symbol_{var}}, \mathtt{arg}\} \supset h_1(v) = h_2(v)$
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 follows the structure of the proof in (Kuhlmann & Stone 2007) but is adapted to deal with argument reorderings.*

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 Γ be the game for a game description D, h be an automorphism of the rule graph of D, and σ be a scrambling of D corresponding to h. Intuitively, σ can be understood as a bijective mapping between arbitrary terms of D. 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 the definition of $s \in t$. The remaining properties of a symmetry can be shown in a similar fashion.*

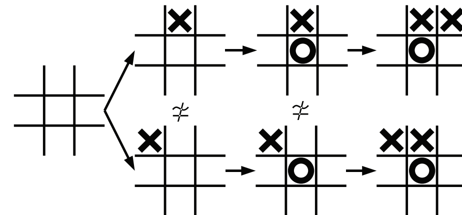
If we use the rule graph of the complete game description D to compute symmetries, we only get symmetries that are present in the initial state of the game. However, there may be so called "dynamic symmetries", i.e., symmetries that occur only in some states of the game but are not present in the initial state. To also find these symmetries, we use the rule graph of $D' = D \setminus \{init(F) \in D\}$, i.e., the rules of D except for the initial state description. Observe that, D can contain function symbols or constants that are not included in 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.

With minor changes, the approach can be used for computing only certain types of symmetries. E.g., only state symmetries are computed by assigning each node belonging to a `role`-fact a different label. Symmetries of a particular state s, can be computed by using the rule graph of $D' \cup s^{true}$.

Exploiting Symmetries

Standard tools, like nauty (`http://cs.anu.edu.au/~bdm/nauty/`), are able to compute the automorphisms of a rule graph and thus the symmetries of a game efficiently (at most a few seconds, even for large games). Which leaves the question of how to exploit the symmetries to improve the game play. Because all current players employ some kind of search to play general games, we present a way to use the symmetries for pruning the search space. Depending on the approach used in the general game player, other uses of the symmetries may exist, e.g., for speeding up analyzation of the game's properties or improving the evaluation function.

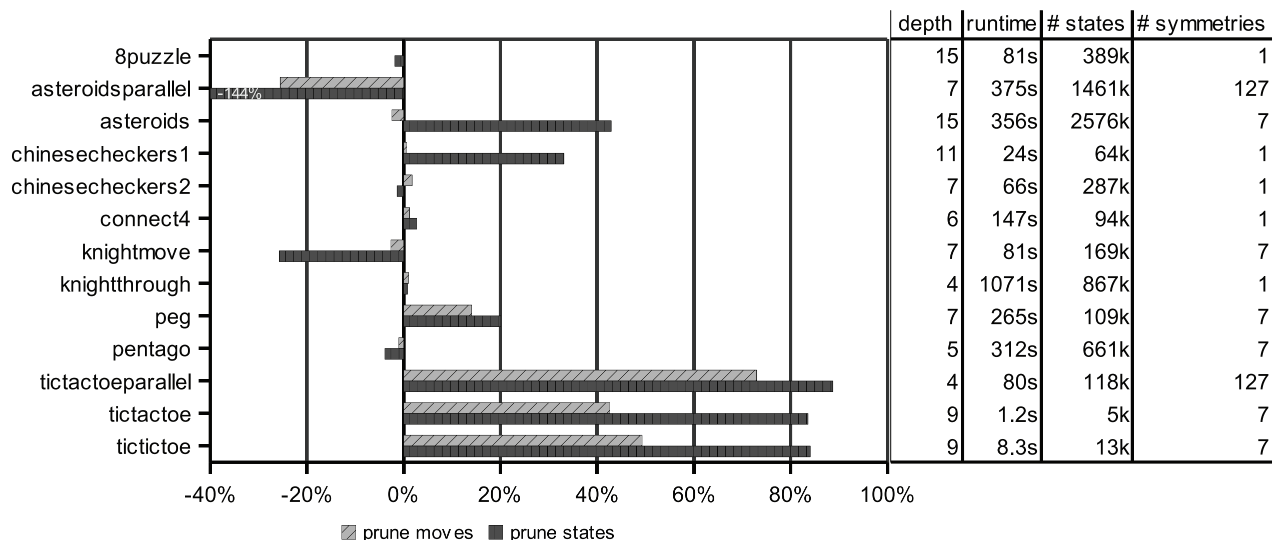
One way of pruning the search space is to prune symmetric joint actions in node expansion. It is clear that it is sufficient to use only one joint action of each set of symmetric joint actions in a state for node expansion because symmetric joint actions lead to symmetric states and yield the same value in game tree search. However, this does not use the full potential of the available information. In particular, 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 symmetric. E.g., the two action sequences of Tic-tac-toe depicted in the following figure are non-symmetric but lead to symmetric states.



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 for computing the symmetric states and checking 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



game	depth	runtime	# states	# symmetries
8puzzle	15	81s	389k	1
asteroidsparallel	7	375s	1461k	127
asteroids	15	356s	2576k	7
chinesecheckers1	11	24s	64k	1
chinesecheckers2	7	66s	287k	1
connect4	6	147s	94k	1
knightmove	7	81s	169k	7
knightthrough	4	1071s	867k	1
peg	7	265s	109k	7
pentago	5	312s	661k	7
tictactoeparallel	4	80s	118k	127
tictactoe	9	1.2s	5k	7
ticticoe	9	8.3s	13k	7

Figure 1: 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. The adjoining table shows the depth-limit that we used for each game, the runtime of the normal search, the number of states expanded by the normal search and the number of symmetries found in the game.

our implementation the time to compute all symmetric states for some 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"). In figure 1 the time savings for search with symmetry pruning compared to "normal search" are shown. The adjoining table shows runtimes of the "normal search" and the depth limits we used for the games.

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. In knightmove the problem is that many symmetric states can only be reached after action sequences that are longer than the depth-limit. Additionally, because of the very simple rules of the game, computing state expansion is fast compared to computing symmetric states. For tictactoe and ticticoe 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. The reason for the large number of symmetries in asteroidsparallel and tictactoeparallel is that these games consist of two independent instances of the same game. These games could be played much more efficiently by decomposing them (Zhao, Schiffel, & Thielscher 2009) and looking for symmetries in each subgame separately.

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 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, we add the stored number to n_{saved}. We estimate 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 all symmetric states for a state. Both, t_{exp} and t_{sym} can be estimated by averaging over times measured for a number or random states of the game. If $t_{saved} < -t_{limit}$, we switch to normal search thereby limiting the negative impact to t_{limit}.

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 than with our approach. The method described in (Fox & Long 2002) for planning can be easily adapted to the general game playing domain. Plan permutation symmetries, that are exploited in e.g., (Long & 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.

A previous approach to symmetry detection in general games is (Banerjee, Kuhlmann, & Stone 2006). The paper informally describes a method to detect certain symmetries in board games that is potentially very expensive, because it requires to enumerate all states of a game. Furthermore, it only works under the assumption that one can detect the board in the game. Our approach is typically much cheaper because it is based on the game rules instead of the states, and more general because it is not limited to board 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. Consequently, symmetry detection based on different game descriptions for the same game may lead to different results. E.g., adding the tautological rule `p(a) :- a\=a.` to the Tic-tac-toe example, would mean that interchanging `a` and `c` results in a different game description. Therefore the symmetry between `a` and `c` wouldn't be detected. 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 it does not allow to map arbitrary terms to each other. E.g., the approach can not detect the symmetry in a variant of Tic-tac-toe, where we rename `a` to `f(a)`, because an automorphism only maps single vertices to each other but `f(a)` is not represented by a single vertex in the rule graph, while `c` is. It is in principle possible to overcome this limitation by propositionalizing a game description and using one vertex per proposition. The resulting rule graphs would be very similar to propositional automata and could in addition be used to improve reasoning speed(Schkufza, Love, & Genesereth 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.

Summary

We presented a sound method to compute symmetries of a game whose rules are given in the Game Description Language. 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, F. A.; Ramani, A.; Markov, I. L.; and Sakallah, K. A. 2002. Solving difficult sat instances in the presence of symmetry. In *Design Automation Conference*. University of Michigan.

Banerjee, B.; Kuhlmann, G.; and Stone, P. 2006. Value function transfer for general game playing. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*.

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

Fox, M., and Long, D. 2002. Extending the exploitation of symmetries in planning. In *Proceedings of AIPS'02*.

Kuhlmann, G., and Stone, P. 2007. Graph-based domain mapping for transfer learning in general games. In *Proceedings of The European Conference on Machine Learning*.

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

Love, D.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. *General Game Playing: Game Description Language Specification*. Stanford Logic Group.

Puget, J.-F. 2005. Automatic detection of variable and value symmetries. In van Beek, P., ed., *CP*, volume 3709 of *Lecture Notes in Computer Science*, 475–489. Springer.

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

Schkufza, E.; Love, N.; and Genesereth, M. R. 2008. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Australasian Conference on Artificial Intelligence*. Springer.

Zhao, D.; Schiffel, S.; and Thielscher, M. 2009. Decomposition of multi-player games. In *Proceedings of the Australasian Joint Conference on Artificial Intelligence*.

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