Factoring General Games

Martin Günther Stephan Schiffel Michael Thielscher
Department of Computer Science
Dresden University of Technology, Germany
{martin.guenther, stephan.schiffel, mit}@inf.tu-dresden.de

Abstract

The goal of General Game Playing is to construct an autonomous agent that can play games it has never encountered before. Only the game rules are given explicitly; interesting and useful properties of a game are implicit and need to be deduced. One particular such property is decomposability into separate subgames that have little interaction with each other. We present a method to decompose a given game into its subgames by analyzing the preconditions and effects of actions. Moreover, a search algorithm that exploits this information in single-player games is proposed.

1 Introduction

General Game Playing (GGP) is the challenge to build an autonomous agent that can effectively play games that it has never seen before. Unlike classical game playing programs, which are designed to play a single game like chess or checkers, the properties of these games are not known to the programmer at design time. Instead, they have to be discovered by the agent itself at runtime. This demand for higher flexibility requires the use and integration of various Artificial Intelligence techniques and makes GGP a grand AI challenge.

Held since 2005, the annual General Game Playing competition [Genesereth *et al.*, 2005] fosters research efforts in this area. Participating systems are pitted against each other on a variety of different types of games. One special class of games that appeared in the last two GGP competitions can be described as *composite games*: games that are composed of simpler *subgames*. Examples from last year's competition include "doubletictactoe" (two games of tic-tac-toe, played on separate boards in parallel) and "incredible amazing blocks world", or short "incredible" (a single player game consisting of a "blocks world" problem, mixed with another simple puzzle called "maze"). Although each subgame had a comparatively small search space and therefore would have been easy to solve individually, all of the players showed a bad performance on the composite games because of the exponentially larger search space of the composite game. For example, a combination of n Tic-Tac-Toe games has an initial branching factor of 9^n, which reduces to $n \cdot 9$ if decomposability can be detected and exploited. Current GGP systems lack two capabilities that would be necessary to play these games on a human level. First, they are not able to separate the consequence of a move in one of the subgames from a move in another. Second, even if they were able to extract subgame information from the game description, the standard search algorithms have no way of exploiting this information.

In this paper, we address both of these issues: We present a method to decompose an arbitrary game into subgames, and we develop an efficient search algorithm for such composite games in the single-player case. The methods and algorithms described in this paper have all been successfully integrated into the GGP system "Fluxplayer", which has won the competition in 2006 [Schiffel and Thielscher, 2007].

The remainder of this paper is organized as follows. In the next section, we briefly recapitulate the Game Description Language from [Genesereth *et al.*, 2005]. In Section 3, we describe a method for extracting subgame information from a game description, and in Section 4 we develop a search algorithm for composite, single-player games. We then evaluate the implementation of the algorithm on a composite game and conclude with a discussion and comparison to related work.

2 Game Description Language

The Game Description Language (GDL) [Genesereth *et al.*, 2005; Love *et al.*, 2008] is the language used to communicate the rules of the game to each player. It is a variant of first order logic, enhanced by distinguished symbols for the conceptualization of games. GDL is purely axiomatic, i.e. no algebra or arithmetics is included in the language; if a game requires this, the relevant portions of arithmetics have to be axiomatized in the game description.

The class of games that can be expressed in GDL can be classified as n*-player* ($n \geq 1$), *deterministic*, *perfect information* games with *simultaneous moves*. "Deterministic" excludes all games that contain any element of chance, while "perfect information" prohibits that any part of the game state is hidden from some players, as is common in most card games. "Simultaneous moves" allows to describe games like "roshambo", where all players move at once, while still permitting to describe games with alternating moves, like chess or checkers, by restricting all players except one to a single "no-op" move. Also, GDL games are *finite* in several ways: All reachable states are composed of finitely many fluents;

Figure 1: Some GDL rules of the game "incredible"

```
1  role(robot).
2  init(clear(c)).  init(on(c,a)).
3  init(table(a)).  init(cell(w)).
4  init(gold(y)).   init(step(c1)).
   ...
5  next(on(X,Y))   :- does(robot,stack(X,Y)).
6  next(table(X)) :- does(robot,stack(U,_)),
     true(table(X)), U \= X.
   ...
7  legal(robot,stack(X,Y)):- true(clear(X)),
     true(table(X)), true(clear(Y)), X \= Y.
   ...
8  goal(robot, 75) :- completed(caetower),
     not completed(bdftower), true(gold(w)).
   ...
9  completed(caetower) :- true(on(c,a)),
     true(on(a,e)).
   ...
10 terminal :- true(step(c20)).
11 terminal :- true(gold(w)).
```

there is a finite, fixed number of players; each player has finitely many possible actions in each game state, and the game has to be formulated such that it leads to a terminal state after a finite number of moves. Each terminal state has an associated goal value for each player, which need not be zero-sum.

A game state is defined by a set of atomic properties, the *fluents*, that are represented as ground terms. The leading function symbol of a fluent will be called a *fluent symbol*.

One of these game states is designated as the initial state. The transitions are determined by the combined actions of all players. The game progresses until a terminal state is reached.

Example 1. Figure 1 shows the GDL game description[1] of the game "incredible", which consists of two subgames: the well-known "blocks world" and "maze", another very simple single-player game in which a robot has to find a piece of gold.

The `role` keyword (line 1) declares the argument, `robot`, to be a player in the game (the only one, in this case).

The initial state of the game is described by the keyword `init` (lines 2–4). A blocks world state with six blocks is described, two of which (`a` and `c`) are shown here: `c` is clear (no block on top), `c` is on `a`, and `a` is directly on the table. The fluent symbols `cell` and `gold` belong to the subgame "maze" and describe the positions of the robot and the gold, respectively.

The fluent `step(c1)` is a "step counter", a technicality often encountered in GDL games. Since only games with finite length are permitted, the game designer must ensure that all matches reach a terminal state after a finite number of steps (if this is not already guaranteed, as in tic-tac-toe). The most common way to do this is by adding a fluent (here: `step`) to the game state, increment its value after each action, and terminate the game when a certain maximal value is reached. These step counters can have a negative impact on the performance of many search enhancements, for example transposition tables, and also on subgame decomposition (Section 3). To solve this problem, this paper will include a method to detect such action-independent fluents.

The keyword `next` (lines 5–6) defines the effects of the players' actions. For example, line 5 declares that, after the player `robot` has executed action `stack` on two blocks `X` and `Y`, these blocks are `on` (top of) each other in the resulting state. The reserved keyword `does` can be used to access the actions executed by the players, while `true` refers to all fluents that hold in the current state. GDL also requires the game designer to state the non-effects of actions by specifying frame axioms, as can be seen on line 6: Stacking a block `U` onto some other block leaves all blocks `X` other than `U` on the table if they have been on the table before.

The keyword `legal` (line 7) defines what actions are possible for each player in the current state; the game designer has to ensure that each player always has at least one legal action available. An action is a ground term; its leading function symbol will be called an *action symbol* here.

The `goal` predicate (line 8) assigns a number between 0 (loss) and 100 (win) to each role in a terminal state. It is defined with the help of the *auxiliary predicate* `completed` (line 9). Auxiliary predicates are not part of the language, but are defined in the game description itself.

The game is over when a state is reached where the `terminal` predicate (lines 10–11) holds.

Predicate bodies may call other predicates. For the analysis of such relationships, the concept of a *call graph* is often useful.

Definition 1. *Let F be a GDL formula. The* ***call graph*** *$G = \langle V, E\rangle$ for F is the smallest graph with the following properties:*

1. *if t is an atomic formula occurring in F, then $t \in V$*
2. *if there is a game rule r with head h, and $t \in V$ and h are unifiable, then*
   - *all atomic formulæ $b_1, \ldots, b_n$ occurring in the body of r are elements of E, and*
   - *$\{\langle t, b_1\rangle, \ldots, \langle t, b_n\rangle\} \subseteq E$*

The overall goal of "incredible" is to achieve a specific configuration of blocks as well as to bring home the gold. The best way to play this game is to solve the two subgames independently and then to combine the solutions. However, the fact alone that the game is decomposable is not explicit in the mere game rules.

3 Subgame Detection

In this section, we present a method to determine if a certain game has independent parts that do not affect each other. The basic idea is to build a dependency graph of the action symbols and fluent symbols and then identify the connected components of this graph. These connected components correspond to the subgames of that particular game.

The dependency graph for the game of "incredible" is shown in Figure 2. Each fluent symbol and action symbol

[1] We use prolog notation with variables denoted by uppercase letters.

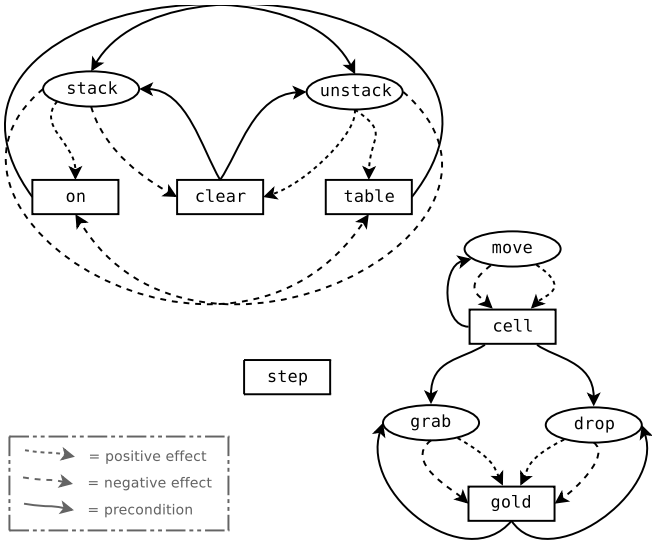


Figure 2: Fluent and action dependency graph for the game "incredible"

of the game is represented by a node in the graph. For each potential positive or negative effect of an action, an edge from action symbol to fluent symbol is added. Also, for each precondition, an edge from fluent symbol to action symbol is added.

Unfortunately, because of the use of frame axioms and the closed-world assumption, a GDL axiomatization does not contain explicit information about the positive and negative effects of an action. We therefore base the construction of the dependency graph on the following definitions:

Definition 2. *Fluent symbol φ is a **potential positive effect** of action symbol α if there is a game rule* $\texttt{next}(\varphi(\vec{x})) \leftarrow B$ *such that B does not imply* $\texttt{true}(\varphi(\vec{x}))$*, and B is compatible with* $\exists p, \vec{y}.\, \texttt{does}(p, \alpha(\vec{y}))$.

*Fluent symbol φ is a **potential negative effect** of action symbol α if there is* no *game rule* $\texttt{next}(\varphi(\vec{x})) \leftarrow B$ *such that* $\forall p, \vec{y}.\, (\texttt{true}(\varphi(\vec{x})) \wedge \texttt{does}(p, \alpha(\vec{y})) \supset B)$.

Definition 3. *Fluent symbol φ is a **potential precondition** for action symbol α if φ occurs in the call graph of the body of a game rule with*

- *head* $\texttt{legal}(p, \alpha(\vec{x}))$*, or*
- *head* $\texttt{next}(\varphi'(\vec{y}))$*, where φ' is a potential positive or negative effect of α.*

In the definitions above, compatibility means logical consistency under the condition that each player can do only one action at a time (and assuming both uniqueness-of-names for action symbols and domain closure for role names). Thus a fluent is a potential positive effect if it is entailed by a non-frame axiom compatible with the action in question, and it is a potential negative effect if there is no frame axiom for this fluent which is compatible with the action.

It can be easily verified that every actual (positive or negative) effect must be a potential effect according to this definition. The converse, however, is not true; e. g., the body of a rule $\texttt{next}(F) \leftarrow B$ may only be compatible with some $\texttt{does}(P, A)$ in states where A is not a legal move. This implies that our dependency graph may contain more arcs than necessary. Determining the exact positive and negative effects in a game would in general require to traverse the entire state space.

It can also easily be seen that this approach fails in the presence of action-independent fluent symbols, such as `step`: By definition, these fluent symbols appear in the positive and negative effects of all actions. Hence, all actions would end up in the same connected component, preventing any decomposition.

To avoid this, we detect an action-independent fluent symbol according to the following definition and put it into a separate subgame.

Definition 4. *A fluent symbol φ is called **action-independent** if*

- *the call graph of any axiom with head* $\texttt{next}(\varphi(\vec{x}))$ *contains no actions or fluents except φ, and*
- *φ does not appear in the call graph of the body of any other* `next` *or* `legal` *axioms.*

The result of finding the connected subgraphs in the dependency graph is a set of subgames $\Phi = \{\sigma_1, \ldots, \sigma_n\}$, where each subgame σ_i is a pair of sets representing the fluents and actions of one connected component.

Example 2. As an example, the result of running the subgame detection algorithm on the game "incredible" is the set $\Phi = \{\sigma_{maze}, \sigma_{blocks}, \sigma_{step}\}$ with

$$\sigma_{maze} = (\{\texttt{cell}, \texttt{gold}\}, \{\texttt{move}, \texttt{grab}, \texttt{drop}\})$$
$$\sigma_{blocks} = (\{\texttt{on}, \texttt{clear}, \texttt{table}\}, \{\texttt{stack}, \texttt{unstack}\})$$
$$\sigma_{step} = (\{\texttt{step}\}, \emptyset)$$

This subgame information can be exploited by specialized search algorithms. In the following section, we will present such an algorithm for single-player games.

4 Concept Decomposition Search

4.1 Motivation

A naïve search algorithm that exploits subgame information would be the following: Search each subgame as if it were a separate game (this is possible because the fluents and actions of different subgames cannot influence each other), yielding a local plan (a sequence of actions that leads to a desired goal state). Then, execute the resulting local plans one after another.

However, this simple approach does not generally find an optimal solution due to the following two reasons: First, the `goal` predicate is only defined on complete states. Therefore, the local searches cannot determine the goal value of a partial state using only fluents from one subgame. Second, the local plans cannot simply be concatenated, since the `terminal` predicate may terminate the game before all plans have been executed. Instead, it is necessary to interleave the plans, i. e., to search all permutations of actions that respect the ordering of the actions in their respective local plans. This is illustrated by the following example.

Figure 3: Concept Decomposition Search

```
function ConceptDecompositionSearch(Φ)
    depth ← 1
    best_glob_plan ← ∅
    loc_plans ← ∅
    while (val(best_glob_plan) < 100) do
        for each σ ∈ Φ do
            loc_plans′ ←
                LocalSearch(σ, depth, loc_plans)
            loc_plans[σ] ← loc_plans[σ] ∪ loc_plans′
        end for
        best_glob_plan ← GlobalSearch(loc_plans)
        depth ← depth + 1
    end while
    return best_glob_plan
end function
```

Example 3. Consider a game played on the following two graphs:

$$\texttt{pos1(a)} \xrightarrow{\texttt{move1(b)}} \texttt{pos1(b)} \xrightarrow{\texttt{move1(c)}} \texttt{pos1(c)}$$

and

$$\texttt{pos2(x)} \xrightarrow{\texttt{move2(y)}} \texttt{pos2(y)} \xrightarrow{\texttt{move2(z)}} \texttt{pos2(z)}$$

The initial state is $[\texttt{pos1(a)}, \texttt{pos2(x)}]$, and each `move` action moves to the corresponding node on the graph. The two states $[\texttt{pos1(a)}, \texttt{pos2(z)}]$ and $[\texttt{pos1(c)}, \texttt{pos2(x)}]$ are terminal and have the goal value 0. The only state with goal value 100 is $[\texttt{pos1(c)}, \texttt{pos2(z)}]$.

It is easy to see that the game can be decomposed into two subgames and that each optimal global plan involves interleaving local plans from both subgames, such as:

$$[\texttt{move1(b)}, \texttt{move2(y)}, \texttt{move2(z)}, \texttt{move1(c)}]$$

These considerations motivate the algorithm that will be presented in the remainder of this section.

4.2 Overview of the Algorithm

The search process (Figure 3) is split into two stages, *local search* and *global search*. Local search only considers one subgame at a time, collecting all *local plans* (i. e., sequences of actions that only contain actions from the given subgame) that may be relevant to the global solution. In a second step, global search tries to interleave local plans from different subgames to find the best *global plan*. These two steps are embedded into an iterative deepening framework.

Dividing local and global search, and interleaving local plans during global search, solves the two problems demonstrated by the previous example. It enables local search to collect only those local plans that might have a novel effect on the global `goal` and `terminal` predicates, while deferring evaluation of these predicates to the point when the complete state is known.

4.3 Local Search

LocalSearch (Figure 4) searches the game tree of subgame σ up to the given $depth$, returning all relevant local plans that are not yet contained in $loc\_plans[\sigma]$.

Figure 4: Local Search

```
function LocalSearch(σ, depth, loc_plans)
    loc_plans′ ← ∅
    traverse tree using LimitedDepthSearch(σ, depth)
    for each leaf_node reached via actions plan do
        plan_sig ← CalculatePlanSig(plan)
        if ⟨plan_sig, _⟩ ∉ loc_plans′ ∪ loc_plans[σ] then
            loc_plans′ ← loc_plans′ ∪ ⟨plan_sig, plan⟩
        end if
    end for
    return loc_plans′
end function
```

Figure 5: Global Search

```
function GlobalSearch(loc_plans)
    best_glob_plan ← ∅
    subsets ← ChoosePlans(loc_plans)
    for each plan_set ∈ subsets do
        glob_plan ← CombinePlans(plan_set)
        if val(glob_plan) > val(best_glob_plan) then
            best_glob_plan ← glob_plan
        end if
    end for
    return best_glob_plan
end function
```

LimitedDepthSearch is any standard tree search algorithm; when it reaches a leaf node, it checks if the action sequence $plan$ (i. e. the sequence of actions from the root of the search tree to the leaf node) constitutes a "relevant" new local plan.

This notion of relevance is based on the following claim: Not all local plans have to be included into the global search. Instead, it is possible to compute some characteristics of local plans, here called *plan signature*, with the property that, if two local plans have the same plan signature, it does not matter which one of them is included in the global search.

This $plan\_sig$ is calculated by a function CalculatePlanSig, whose inner workings will be examined in Section 4.6; for now, it suffices to treat it as a black box.

4.4 Global Search

In between local search iterations, global search (Figure 5) tries to find a globally optimal execution order of the local plans. Global search thus is not a state space search, but a search on the space of plans.

The function ChoosePlans simply calculates all subsets of loc_plans that include at most one plan from each subgame. Then, CombinePlans (Section 4.5) tries to find an execution order of the actions in each of those $plan_set$s such that the resulting global plan can be executed (i. e., does not reach a terminal state until all its actions have been executed).

After all possible global plans have been evaluated, the one with the highest goal value is returned.

Figure 6: CombinePlans

global variable $visited : \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{n \text{ times}} \rightarrow \{true, false\}$

function COMBINEPLANS($plan\_set$)
  return CPR($plan\_set, [0, 0, \ldots, 0], initial\_state$)
end function

function CPR($\{plan_1, \ldots, plan_n\}, pos, state$)
  let $pos = [p_1, \ldots, p_n]$
  let $plan_1$ = $[action_1^1, action_2^1, \ldots, action_{m_1}^1]$
  let $plan_2$ = $[action_1^2, action_2^2, \ldots, action_{m_2}^2]$
  $\vdots$
  let $plan_n$ = $[action_1^n, action_2^n, \ldots, action_{m_n}^n]$
  if $visited[pos] = true$ **then**
    return $fail$
  else if $pos = [m_1, m_2, \ldots, m_n]$ **then**
    return $\emptyset$
  end if
  for $i = 1, \ldots, n$ **do**
    if $p_i < m_i$ **then**
      $state' \leftarrow$ EXECUTE($action_{p_i+1}^i, state$)
      $pos' \leftarrow [p_1, \ldots, p_{i-1}, p_i + 1, p_{i+1}, \ldots, p_n]$
      $res \leftarrow$ CPR($\{plan_1, \ldots, plan_n\}, pos', state'$)
      if $res \neq fail$ **then**
        return $[action_{p_i+1}^i \circ res]$
      end if
    end if
  end for
  $visited[pos] \leftarrow true$
  return $fail$
end function

4.5 Combination of Plans

The purpose of COMBINEPLANS is to combine a set of plans $\{plan_1, \ldots, plan_n\}$ such that the resulting global plan does not reach a terminal state prematurely.

Using a dynamic programming technique (Figure 6), all valid sequences of actions can be enumerated in $m_1 * \cdots * m_n$ steps, where the m_i are the lengths of the local plans. The idea is to search the plan space recursively via depth-first search, while maintaining a map of already visited nodes, so that no node is visited twice. Each node of the search space is determined by the set of executed actions, without regarding their order. Since there are only $m_1 * \cdots * m_n$ unique nodes in the search space and each of them is at most visited once, both space and time complexity are polynomial in the number of actions.

4.6 Calculating Plan Signatures

So far, we have avoided the topic what exactly constitutes a "plan signature". The only requirement was that, if two local plans have the same plan signature, both of them must be equivalent with respect to the global search.

One obvious feature that distinguishes one local plan from another is the final state that is reached by executing the plan. More precisely, only those fluents of the final state that also appear in the `goal` and `terminal` predicates need to be considered.

Unfortunately, comparing the final states of two local plans is not enough. As demonstrated by the example in Section 4.1, the `terminal` predicate can make the combination of local plans more complicated or even impossible, depending on the order in which fluents from the `terminal` predicate become true or false in the intermediary states of a local plan. Therefore, if two local plans from the same subgame reach the same final state, but the sequence of terminal fluents in the intermediary states is different, both must be returned to be considered by the global search. Thus, the sequence of terminal fluents must be part of the plan signature.

Including all possible combinations of `terminal` fluents in the plan signature would drastically increase the number of local plans that local search returns, thereby eliminating the benefit of factoring the game into subgames. Therefore, we perform a deeper analysis of the `goal` and `terminal` predicates to identify specific combinations of fluents that need to be considered. The idea of this approach is to split the `goal` and `terminal` predicates into subpredicates that are local to a single subgame. These subpredicates often represent a concept like "checkmate" or "line" that is used to describe the terminal and goal states abstractly. Hereafter, these subpredicates will be called *local goal* (resp. *terminal*) *concepts*.

Definition 5. *A **local goal (resp. terminal) concept**, or short "local concept", is a ground predicate call that occurs in the call graph of the* `goal` *(resp.* `terminal`*) predicate. The call graph of a local goal (resp. terminal) concept must only contain fluents from exactly one subgame.*

To find these concepts, call graphs of the `goal` and `terminal` predicates are built. These graphs are then traversed from the root nodes (`goal` or `terminal`) downward until a predicate is found whose children all belong to the same subgame. The algorithm requires that the part of the call graph up to the local concepts is ground (contains no variables). Also, recursion is only allowed inside the local concepts (i. e., a local concept may not appear in the body of itself or another local concept, or in the body of a predicate that appears in the body of a local concept and so on).

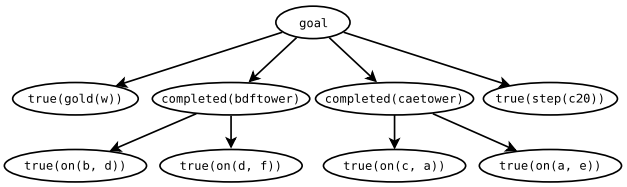
Example 4. The goal and terminal call graphs of "incredible" are depicted in Figure 7. Application of the algorithm leads to the following local goal and terminal concepts:

- `true(gold(w))` (goal and terminal concept of σ_{maze})
- `completed(bdftower)` (goal concept of σ_{blocks})
- `completed(caetower)` (goal concept of σ_{blocks})
- `true(step(c20))` (goal and terminal concept of σ_{step})

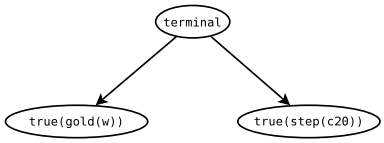
Definition 6. *A **concept evaluation** for a given state s from subgame σ is a tuple of boolean values $\langle b_1, b_2, \ldots, b_n \rangle$. Assuming a fixed order among the local concepts, b_i $(1 \leq i \leq n)$ corresponds to the evaluation of the i^{th} local (goal or terminal) concept of σ in state s.*

Example 4 (cont.). In the initial state of "incredible", the concept evaluation of σ_{maze} is $\langle false \rangle$, that of σ_{blocks} is $\langle false, false \rangle$ and that of σ_{step} is also $\langle false \rangle$.

One last auxiliary definition is the following:



(a) `goal` predicate



(b) `terminal` predicate

Figure 7: Call graphs of the game "incredible"

Table 1: Comparison of Concept Decomposition Search and standard Fluxplayer search algorithm on the game "incredible"

	# computed states	computation time [s]
Decomposition Search	3,212	45.331
Standard Fluxplayer	41,191,436	8510.648

Definition 7. *The **terminal concept evaluation sequence** for a given local plan p from subgame σ is a sequence of concept evaluations for σ, but restricted to terminal concepts, of all states that are traversed by executing p. Repeated elements of this sequence are omitted.*

With this in hand, we can finally give a formal definition of the term "plan signature".

Definition 8. *A **plan signature** of a local plan p is a pair $\langle s, t \rangle$, where s is the concept evaluation of the final state reached by executing p, and t is the terminal concept evaluation sequence of p.*

5 Experiments

The algorithms described above have been implemented and integrated into the General Game Player Fluxplayer. Here we present experimental results for the game of "incredible". The experiments were carried out on a Pentium M 1.7 GHz CPU and 1 GB of RAM.

Table 1 shows the results of running the subgame decomposition algorithm, followed by concept decomposition search. Both the number of expanded states and the CPU time is shown. For comparison, the results of running Fluxplayer's standard non-uniform search [Schiffel and Thielscher, 2007] are also listed.

6 Discussion

We have presented an automatic decomposition method for general games, and a search algorithm for single player general games that builds on top of it. In this section, we will relate our results to previous work and suggest directions for future improvements.

Subgame decomposition could be improved by propositionalizing parts of the fluents and actions. This would allow decomposition into more subgames, improving the efficiency of the search algorithm at the cost of a higher computational complexity of the decomposition algorithm.

The issue of searching single-player general games has already been addressed by research in planning, most notably three algorithms in the area of Factored Planning: *PartPlan* [Amir and Engelhardt, 2003], *LID-GF* [Brafman and Domshlak, 2006] and *dTreePlan* [Kelareva *et al.*, 2007]. Similar to Concept Decomposition Search, *PartPlan* and *LID-GF* find all local plans in the different subdomains, then combine them into a global plan. The approach of *dTreePlan* is different in that it does not search subdomains exhaustively before searching the parent, but instead uses a depth-first search with backtracking over subdomains.

A key difference between the present paper and the Factored Planning approaches is the problem encoding at hand. The *dTreePlan* algorithm uses the more restricted STRIPS formalism. While a large part of the algorithm can be extended to more expressive formalisms, the goal predicate must be a STRIPS-style goal formula, i. e., a conjunction of non-negated fluents. This is usually not the case in GDL games. While *PartPlan* uses the situation calculus, and *LID-GF* the SAS$^+$ formalism, their results can be generalized to PDDL, so they do not have this restriction.

The presence of frame axioms in GDL makes the analysis of the preconditions and effects of an action harder than in either of the planning formalisms. Another difference is the presence of the `terminal` predicate in GDL. This adds another level of possible interactions between subgames, which is handled by the terminal concept evaluation sequence in our algorithm. Also, the Factored Planning algorithms only operate on the fluent level and not on concepts; this is probably due to the fact that derived predicates are very common in GDL games, but not widely used in PDDL.

Since the Factored Planning approaches allow for a greater deal of interaction between subgames, unifying Concept Decomposition Search and Factored Planning would be desirable.

Future work could also be done on other forms of independence, like *parallel independence*: games where each action makes a move in all independent subgames simultaneously.

On the topic of multi-player games, *decomposition search* [Müller, 1999] presents a search method for decomposable two-person games based on combinatorial game theory [Conway, 1976]. It does not, however, contain an automatic method for decomposition of general games. Our decomposition method can deliver a subgame decomposition that can be used by two-player decomposition search, which is an important topic for future work. Also, search methods for *non-combinatorial multi-player games* (decomposable games that have more than two players, feature simultaneous moves or do not have zero-sum awards) are needed.

References

[Amir and Engelhardt, 2003] Eyal Amir and Barbara Engelhardt. Factored planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 929–935, Acapulco, Mexico, August 9–15, 2003. Morgan Kaufmann.

[Brafman and Domshlak, 2006] Ronen I. Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference*, pages 809–814, Boston, Massachusetts, USA, July 16–20, 2006. AAAI Press.

[Conway, 1976] John Horton Conway. *On Numbers and Games*. Number 6 in London Mathematical Society Monographs. Academic Press, London, 1976.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Kelareva *et al.*, 2007] Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1942–1947, Hyderabad, India, January 6–12, 2007.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford University, March 2008.

[Müller, 1999] Martin Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgames. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 578–583, Stockholm, Sweden, July 31 – August 6, 1999. Morgan Kaufmann.

[Schiffel and Thielscher, 2007] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1191–1196, Menlo Park, California, USA, July 22–26, 2007. The AAAI Press.