Großer Beleg

Decomposition of Single Player Games

Accelerating Search by Symmetry Detection in General Game Playing

by

Martin Günther

Born on November 10, 1980 in Berlin-Neukölln

Dresden University of Technology

Department of Computer Science
Artificial Intelligence Institute
Computational Logic Group

Supervised by:

Prof. Dr. rer. nat. habil. Michael Thielscher
Dipl.-Inf. Stephan Schiffel

Submitted on July 16, 2007

Günther, Martin
Decomposition of Single Player Games
Großer Beleg, Department of Computer Science
Dresden University of Technology, July 2007

Aufgabenstellung Großer Beleg

Name, Vorname:	Günther, Martin
Studiengang:	Informatik
Matrikelnummer:	2849363
Thema:	***Automatic Decomposition of Compound Single Player Games***
Zielstellung:	Das *General Game Playing (GGP)* ist ein junger Forschungszweig, dessen Herausforderung darin besteht, universelle Spielprogramme zu entwerfen. Diese *General Game Player* sollen in der Lage sein, ein beliebiges Spiel zu beherrschen, von dem sie nur die Regeln in Form von Axiomen erhalten. Dies erfordert eine automatische Analyse der Axiome, um Besonderheiten der konkreten Spiele erkennen und ausnutzen zu können. Eine solche Besonderheit sind Spiele, die aus vollständig unabhängigen Teilspielen bestehen. Während der letzten GGP Competition kamen zwei solche Spiele vor. Der Aufwand zur Lösung des zusammengesetzten Spiels ist exponentiell größer als der Aufwand für die separate Lösung der einzelnen Teilspiele. Daher soll in dieser Arbeit ein Verfahren entwickelt werden, welches unabhängige Teilspiele erkennt und mit Hilfe dieser Information den Spielbaum effizient durchsucht. Es darf vorausgesetzt werden, dass die Spielbeschreibung im Fluentkalkül vorliegt, insbesondere dürfen die Vorbedingungsaxiome und Effektbeschreibungen bei der Analyse benutzt werden. Außerdem sollen vorerst nur Einzelspielerspiele betrachtet werden. Das entwickelte Verfahren soll in Prolog implementiert und in den General Game Player *Fluxplayer* integriert werden. Schwerpunkte: • Erkennung von unabhängigen Teilspielen in einem beliebigen “General Game” • Entwicklung eines effizienten Suchverfahrens für das zusammengesetzte Spiel • Nachweis der Korrektheit des Verfahrens • Implementierung des Verfahrens
Betreuer und verantwortlicher Hochschullehrer:	Prof. Michael Thielscher
Institut:	Künstliche Intelligenz
Lehrstuhl:	Computational Logic
Beginn am:	15.01.2007
Einzureichen am:	15.07.2007

Verantwortlicher Hochschullehrer

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

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. This makes GGP an ideal testbed for the development of new Artificial Intelligence methods.

Since 2005, the Stanford Logic Group, headed by Michael Genesereth, holds an annual General Game Playing competition to foster research efforts in this area. Sponsored by the American Association for Artificial Intelligence, this competition offers the opportunity to compare different approaches in a competitive setting. In the course of several rounds, the participating general game playing systems are pitted against each other on 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 the 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 the subgames themselves were relatively easy to solve, as they had a comparatively small search space, all of the players showed a bad performance on the composite games because of the exponentially bigger search space of the composite game.

This is due to the fact that the current systems are not able to separate the consequence of a move in one of the subgames from a move in another, and even if they were able to extract subgame information from the game description, the employed standard search algorithms have no way of exploiting this information.

Therefore, the aim of this work is

- to formulate a suitable formalism for expressing subgame information,
- to develop an algorithm that detects subgames inside composite games, and
- to acquire a search algorithm that can make use of the detected symmetries in order to accelerate search.

In order to limit the scope of this project, only single player games are considered. All algorithms are implemented in Prolog and integrated into the General Game Player "Fluxplayer" (Schiffel & Thielscher, 2007).

The remainder of this work is organized as follows:

- Chapter 2 (Preliminaries) will introduce two formalisms that will be needed as a foundation, particularly the Game Description Language and the Fluent Calculus.
- Chapter 3 (Subgame Detection) will contain a formalization of the context as well as an algorithm for extracting subgame information from a game description.
- Chapter 4 (Greedy Decomposition Search) will deal with a simple, albeit non-optimal algorithm that can make use of this subgame information for speeding up search.
- Chapter 5 (Concept Decomposition Search) will treat a more complex search algorithm for composite games that is provably optimal.
- Chapter 6 (Discussion) will conclude with an evaluation of the results.

2 Preliminaries

2.1 Game Description Language (GDL)

The Game Description Language (Genesereth *et al.*, 2005), a subset of the Knowledge Interchange Format, 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 "prefect 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: The state space consists of finitely many states; 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 is modeled in GDL as a game state graph; the nodes in the graph are called *game states*, which in turn consist of atomic properties, so-called *datums*, that are represented as ground terms. 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 2.1. Listing 2.1 on the following page shows the GDL game description of the well-known "blocks world" domain.

The `role` keyword (line 1) declares the argument, `robot`, to be a player in the game. From the fact that there is only one `role` statement in the game description, we can see that this is a single player game. Also note that the infix notation of GDL is used; in the prefix notation, the same statement would read `role(robot)`.

The initial state of the game is described by the keyword `init` (lines 3–8). A blocks world state with three blocks (`a`, `b` and `c`) is described, where `b` and `c` are clear (no block on top of them), `c` is on `a`, and both `a` and `b` are directly on the

Listing 2.1 Blocks world

```
(role robot)

(init (clear b))
(init (clear c))
(init (on c a))
(init (table a))
(init (table b))
(init (step 1))

(<= (next (on ?x ?y))
    (does robot (s ?x ?y)))

(<= (next (on ?x ?y))
    (does robot (s ?u ?v))
    (true (on ?x ?y)))

(<= (next (table ?x))
    (does robot (s ?u ?v))
    (true (table ?x))
    (distinct ?u ?x))

(<= (next (clear ?y))
    (does robot (s ?u ?v))
    (true (clear ?y))
    (distinct ?v ?y))

(<= (next (on ?x ?y))
    (does robot (u ?u ?v))
    (true (on ?x ?y))
    (distinct ?u ?x))

(<= (next (table ?x))
    (does robot (u ?x ?y)))

(<= (next (table ?x))
    (does robot (u ?u ?v))
    (true (table ?x)))

(<= (next (clear ?y))
    (does robot (u ?x ?y)))

(<= (next (clear ?x))
    (does robot (u ?u ?v))
    (true (clear ?x)))

(<= (next (step ?y))
    (true (step ?x))
    (succ ?x ?y))

(succ 1 2)
(succ 2 3)
(succ 3 4)

(<= (legal robot (s ?x ?y))
    (true (clear ?x))
    (true (table ?x))
    (true (clear ?y))
    (distinct ?x ?y))
(<= (legal robot (u ?x ?y))
    (true (clear ?x))
    (true (on ?x ?y)))

(<= (goal robot 100)
    (true (on a b))
    (true (on b c)))
(<= (goal robot 0)
    (not (true (on a b))))
(<= (goal robot 0)
    (not (true (on b c))))

(<= terminal
    (true (step 4)))
(<= terminal
    (true (on a b))
    (true (on b c)))
```

table.

The term `(step 1)` defined on line 8 is a "step counter", a technicality often encountered in GDL games. As was mentioned before, only games with finite length are permitted. Whereas some games, like tic-tac-toe or nim, are guaranteed to terminate after a limited number of steps, most games, including blocks world, have cycles in their game graph that would allow infinite sequences of actions (for example, stacking and unstacking the same two blocks over and over again). Therefore, the game designer must ensure that a state in the game graph can never be reached twice during the same match, and that all matches eventually reach a terminal state. The most common way to do this is by adding a primitive feature (here: `step`) to the game state, increment its value after each action (lines 46–48), and terminate the game when a certain maximal value is reached (lines 71–72). These step counters can have a negative impact on the performance of many search enhancements, e.g. transposition tables, and also on subgame decomposition (chapter 3). Therefore, a separate part of the algorithm will be dedicated to their treatment.

The keyword `next` (10–48) defines the effects of the players' actions. For example, lines 10 and 11 declare that, after the player `robot` has executed action `s` (short for "stack") on two blocks `?x` and `?y`, these blocks are `on` top of each other in the resulting state [1] . The reserved keyword `does` can be used to access the actions executed by the players, while `true` refers to all primitive features that are true 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 lines 17–20: Stacking a block `?u` onto another block `?v` leaves all other blocks `?x` on the table, if they were on the table before.

The rule on lines 46–48 increments the step counter, using the successor function defined on lines 50–52. GDL contains no definitions of arithmetics; therefore the constants `1`, `2`, `3` and `4` have no special meaning and could be replaced by something completely different; they only gain meaning as ordinal numbers through the auxiliary predicate `succ`. This is a very simple example of how the vocabulary can be extended by user-defined functions.

The keyword `legal` (54–61) defines what actions are possible for each player in the current state; the game designer has to ensure that each player has at least one legal action in each reachable state of the game tree.

The `goal` predicate (63–69) assigns a number between 0 (loss) and 100 (win) to each role in a terminal state (this is the only place where numbers, such as 100, have a special meaning). Games need not be zero-sum; each player's goal is to maximize its own payoff. The game is over when a state is reached where the `terminal` predicate (71–75) holds.

[1] The symbol `<=` is the reverse implication operator; all expressions beginning with a question mark are variables. Additionally, GDL provides the keyword `distinct` for syntactical inequality and the logical operators `and`, `or` and `not`.

2.2 Game Representation in the Fluent Calculus

While GDL is used to communicate the game description to the players, the players are free to choose a different internal representation of the game, as long as both representations are equivalent. One such alternative representation is the fluent calculus (Thielscher, 1998) and its Prolog-based implementation FLUX, which is the approach chosen by the General Game Player "Fluxplayer" (Schiffel & Thielscher, 2007).

The subgame detection algorithm (presented in chapter 3) will need to prove potential positive and negative effects of actions; for this task, the fluent calculus representation is much better suited than the original GDL representation, because in fluent calculus, positive and negative effects are given directly. The translation from GDL to fluent calculus/FLUX is still work in progress, but lies outside the scope of this work; therefore, in the following, I will assume that the rules of the game have already been translated into a fluent calculus domain axiomatization.

Since GDL only permits deterministic domains with complete knowledge of states, the special fluent calculus (which, in contrast to the general fluent calculus, has no extensions for incomplete states and knowledge representation) is expressive enough. For the sake of brevity, "special fluent calculus" will from now on often be referred to simply as "fluent calculus".

For a thorough introduction to the fluent calculus, refer to Thielscher (2005). The definitions from this book that are most relevant to this work will briefly be repeated in the following section.

2.3 Fluent Calculus

The special fluent calculus is a formalism for reasoning about actions in first-order logic with sorts and equality. Each "datum" in GDL corresponds to a fluent (sort FLUENT) in fluent calculus, and each "game state" corresponds to a state (of sort STATE, a supersort of FLUENT).

Definition 2.1. *A triple* $\langle \mathcal{F}, \circ, \emptyset \rangle$ *is a* ***fluent calculus state signature*** *if*

- $\mathcal{F}$ *finite, non-empty set of function symbols into sort* FLUENT *(the set of* fluent functions*)*
- $\circ : \text{STATE} \times \text{STATE} \to \text{STATE}$
- $\emptyset : \text{STATE}$

The following macro is an abbreviation for the statement that a fluent f holds in a state z[2]:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')(z = f \circ z')$$

[2]Free variables in formulas are assumed to be universally quantified.

The function $\circ$ must obey the following properties, similar to the union operation for sets:

Definition 2.2. *The **foundational axioms** Σ_{sstate} of special fluent calculus are*

1. *Associativity and commutativity:*

$$(z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3)$$
$$z_1 \circ z_2 = z_2 \circ z_1$$

2. *Empty state axiom:* $\neg Holds(f, \emptyset)$
3. *Irreducibility:* $Holds(f, g) \implies f = g$
4. *Decomposition:* $Holds(f, z_1 \circ z_2) \implies Holds(f, z_1) \vee Holds(f, z_2)$
5. *State Equality:* $(\forall f)(Holds(f, z_1) \equiv Holds(f, z_2)) \implies z_1 = z_2$

Definition 2.3. *A **finite state** τ is a term $f_1 \circ \cdots \circ f_n$ such that each f_i $(1 \leq i \leq n)$ is a fluent $(n \geq 0)$. If $n = 0$, then τ is $\emptyset$. A **ground state** is a state without variables.*

Due to the constraints of GDL, all states can in the following be assumed to be ground and finite.

Fluent calculus uses two more sorts: ACTION for representing the actions of the robot, and SIT for situations. Situations represent series of actions, or nodes in the game tree; due to transpositions, several different situations can correspond to the same state.

Definition 2.4. *A tuple $\mathcal{S} \cup \langle \mathcal{A}, S_0, Do, State, Poss \rangle$ is a **fluent calculus signature** if*

- *$\mathcal{S}$ state signature*
- *$\mathcal{A}$ finite, non-empty set of function symbols into sort* ACTION *(the set of* action functions*)*
- S_0 : SIT *(the initial situation)*
- Do : ACTION$\times$SIT $\rightarrow$ SIT *(the successor situation, after performing an action)*
- $State$: SIT $\rightarrow$ STATE *(the mapping of a situation to its corresponding state)*
- $Poss$: ACTION $\times$ STATE *(the precondition predicate, denoting if performing a certain action in a given state is possible)*

For convenience, two more macros are declared:

- $z_1 - z_2$ (removal of all fluents of state z_2 from z_1)
- $z_1 + z_2$ (addition of all fluents of state z_2 to z_1)

Example 2.2. To illustrate how a game can be expressed in fluent calculus, a running example will be provided throughout this section. The game is, again, blocks world, which helps to compare the fluent calculus formulation to the GDL representation from listing 2.1.

The initial state S_0 is characterized by the following formula:

$$State(S_0) = Clear(B) \circ Clear(C) \circ On(C, A) \circ Table(A) \circ Table(B) \circ Step(1)$$

Definition 2.5. *A **state formula** $\Delta(z)$ is a first-order formula with free state variable z and without any occurrence of states other than in expressions of the form $Holds(f, z)$, and without actions of situations. If $\Delta(z)$ is a state formula and s a situation variable, then $\Delta(z)\{z/State(s)\}$ is a **situation formula**, written $\Delta(s)$.*

Definition 2.6. *Consider a fluent calculus signature with action functions $\mathcal{A}$, and let $A \in \mathcal{A}$. A **precondition axiom** for A is a formula*

$$Poss(A(\vec{x}), z) \equiv \Pi(z)$$

where $\Pi(z)$ is a state formula with free variables among $\vec{x}, z$.

Example 2.2 (cont.). In our blocks world example, the `legal` clauses from GDL can be translated pretty straightforwardly into the following precondition axioms.

$$\begin{aligned} Poss(S(x, y), z) \equiv & Holds(Clear(x), z) \wedge Holds(Table(x), z) \\ & \wedge Holds(Clear(y), z) \wedge x \neq y \wedge \neg Terminal(z) \\ Poss(U(x, y), z) \equiv & Holds(Clear(x), z) \wedge Holds(On(x, y), z) \\ & \wedge \neg Terminal(z) \end{aligned}$$

The predicate $Terminal(z)$ can be ignored for now, it will be defined in definition 2.10.

Definition 2.7. *Consider a fluent calculus signature with action functions $\mathcal{A}$, and let $A \in \mathcal{A}$. A **state update axiom** for A is a formula*

$$\begin{aligned} & Poss(A(\vec{x}), s) \implies \\ & \quad (\exists \vec{y}_1)(\Delta_1(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_1^- + \vartheta_1^+) \\ & \quad \vee \cdots \vee \\ & \quad (\exists \vec{y}_n)(\Delta_n(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_n^- + \vartheta_n^+) \end{aligned}$$

where

- $n \geq 1$;
- *for* $i = 1, \ldots, n$,
  - $\Delta_i(s)$ *is a situation formula with free variables among* $\vec{x}, \vec{y}_i, s$;
  - $\vartheta_i^+, \vartheta_i^-$ *are finite states with variables among* $\vec{x}, \vec{y}_i$.

The terms ϑ_i^+ *and* ϑ_i^- *are called, respectively,* ***positive*** *and* ***negative*** *effects of* $A(\vec{x})$ *under* ***condition*** $\Delta_i(s)$.

Example 2.2 (cont.). The following two state update axioms describe the effects of the blocks world actions. As has been mentioned before, the (potential) effects of actions are given directly – compare this to the GDL `next` axioms, where not all action effects are obvious at first glance. State update axioms are the hardest part of the translation between GDL and fluent calculus; while the (potential) positive effects of an action can usually also be easily proven in GDL, the negative effects are specified by a *missing* frame axiom, or a restricted frame axiom that does not sustain a fluent under all conditions.

$$\begin{aligned}
Poss(S(x,y),s) \implies & Holds(Step(u),s) \wedge Succ(u,v) \\
& \wedge State(Do(S(x,y),s)) = State(s) \\
& \quad - Table(x) \circ Clear(y) \circ Step(u) \\
& \quad + On(x,y) \circ Step(v) \\
Poss(U(x,y),s) \implies & Holds(Step(u),s) \wedge Succ(u,v) \\
& \wedge State(Do(U(x,y),s)) = State(s) \\
& \quad - On(x,y) \circ Step(u) \\
& \quad + Table(x) \circ Clear(y) \circ Step(v)
\end{aligned}$$

The predicate *Succ* that is used here is part of the so-called auxiliary axioms that will be specified later.

Definition 2.8. *Consider a fluent calculus signature with action functions* $\mathcal{A}$. *A* ***domain axiomatization*** *in special fluent calculus is a finite set of axioms* $\Sigma = \Sigma_{dc} \cup \Sigma_{poss} \cup \Sigma_{sua} \cup \Sigma_{aux} \cup \Sigma_{init} \cup \Sigma_{sstate}$ *where*

- Σ_{dc} *set of domain constraints;*
- Σ_{poss} *set of precondition axioms, one for each* $A \in \mathcal{A}$;
- Σ_{sua} *set of state update axioms, one for each* $A \in \mathcal{A}$;
- Σ_{aux} *set of auxiliary axioms, with no occurrence of states except for fluents, no occurrence of situations, and no occurrence of Poss;*

- $\Sigma_{init} = \{State(S_0) = \tau_0\}$ *where* τ_0 *is a ground state;*
- Σ_{sstate} *foundational axioms of special fluent calculus.*

As Σ_{dc} has no counterpart in GDL, it will not be used in this work, and the definition of domain constraints will be omitted.

Example 2.2 (cont.). The sets Σ_{init}, Σ_{poss} and Σ_{sua} for blocks world have already been specified; the following axiom is the only auxiliary axiom that is needed for blocks world:

$$Succ(1,2) \wedge Succ(2,3) \wedge Succ(3,4)$$

Definition 2.9. *A domain axiomatization* Σ *is* ***deterministic*** *iff it is consistent and for any ground state* τ *and ground action* a *there exists a ground state* τ' *such that*

$$\Sigma \models State(s) = \tau \wedge Poss(a,s) \implies State(Do(a,s)) = \tau'$$

Since GDL only allows the formulation of deterministic games, it is permissible to assume that all domain axiomatizations considered here are deterministic.

In order to accommodate the GDL concepts of terminal states and goal values, we will need two additional types of axioms that are not part of standard fluent calculus; these can be defined in a way similar to the precondition and state update axioms.

Definition 2.10. *A* ***termination axiom*** *is a formula*

$$Terminal(z) \equiv \Omega(z)$$

where $\Omega(z)$ *is a state formula with free variable* z*.*

Example 2.2 (cont.). The single termination axiom for blocks world is:

$$\begin{aligned} Terminal(z) \equiv & Holds(Step(4), z) \\ & \vee Holds(On(A,B), z) \wedge Holds(On(B,C), z) \end{aligned}$$

Definition 2.11. *A* ***goal value axiom*** *is a formula*

$$\begin{aligned} & Terminal(z) \implies \\ & (\exists \vec{y}_1)(\Gamma_1(z) \wedge Goal(z) = g_1) \\ & \vee \cdots \vee \\ & (\exists \vec{y}_n)(\Gamma_n(z) \wedge Goal(z) = g_n) \end{aligned}$$

where

- $Goal$ *is a function into the interval* $[0, 100]$

- $n \geq 1$;
- *for* $i = 1, \ldots, n$,
  - $\Gamma_i(z)$ *is a state formula with free variables among* $\vec{y}_i, z$;
  - $g_i \in \mathbb{N}, 0 \leq g_i \leq 100$.

As usual, two macros

$$Terminal(s) \stackrel{\text{def}}{=} Terminal(State(z))$$

and

$$Goal(s) \stackrel{\text{def}}{=} Goal(State(z))$$

are defined.

Example 2.2 (cont.). Like the termination axiom, the goal value axiom for blocks world can be translated very easily:

$$\begin{aligned} Terminal(z) \implies & Holds(On(A,B),z) \wedge Holds(On(B,C),z) \\ & \wedge Goal(z) = 100 \\ & \vee \neg Holds(On(A,B),z) \wedge Goal(z) = 0 \\ & \vee \neg Holds(On(B,C),z) \wedge Goal(z) = 0 \end{aligned}$$

Definition 2.12. *A* ***game axiomatization*** *is a finite set of axioms* $\Sigma = \Sigma_{da} \cup \Sigma_{terminal} \cup \Sigma_{goal}$ *where*

- Σ_{da} *domain axiomatization in special fluent calculus;*
- Σ_{term} *set containing only the single termination axiom;*
- Σ_{goal} *set containing only the single goal value axiom.*

In order for a set Σ to be a valid game axiomatization, the following properties must hold:

- Each terminal state has a goal value:

$$\Sigma \models Terminal(z) \implies (\exists g) Goal(z) = g$$

- In a terminal state, no actions are possible:

$$\Sigma \models Terminal(z) \implies \neg Poss(a, z)$$

The two formalisms introduced in this chapter, the game description language and the fluent calculus, will help to formulate an algorithm for detecting subgames in the next chapter.

3 Subgame Detection

3.1 Games, Subgames and Independency

In this chapter, a method to recognize symmetries in games will be presented. The goal of this *subgame detection algorithm* is to determine if a certain game has independent parts that do not affect each other. If this is the case, the game is called a *composite game* and the parts are called *subgames*. This information can then be used to search the game more efficiently, as we will see in chapters 4 and 5.

Definition 3.1. *A **game** is a pair $\langle \mathcal{F}, \mathcal{A} \rangle$, where $\mathcal{F}$ is a set of fluent functions and $\mathcal{A}$ a set of action functions.*

Intuitively, a definition of the term "game" should also include the rules of the game. However, the rules (which are given in the form of a fluent calculus game axiomatization, cf. definition 2.12) will be considered fixed, so they do not have to be explicitly included here.

Definition 3.2. *A game $\sigma = \langle \mathcal{F}_\sigma, \mathcal{A}_\sigma \rangle$ is a **subgame** of another game $\hat{\sigma} = \langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$, written $\sigma \lhd \hat{\sigma}$, iff $\mathcal{F}_\sigma \subseteq \mathcal{F}_{\hat{\sigma}}$ and $\mathcal{A}_\sigma \subseteq \mathcal{A}_{\hat{\sigma}}$.*

Next, we need to define what it should mean for two subgames to be independent. Loosely speaking, two subgames are independent if the execution of an action $A(\vec{x})$ from one subgame has no influence on the fluents $F(\vec{y})$ of the other subgame. Moreover, this "intervention" should not change what actions are possible in the second subgame afterwards, nor what the effects of any sequence of actions are. This is depicted in the following diagram, followed by a formal definition of independency.

$$\begin{array}{ccccccc} s & \xrightarrow{\hat{A}_1(\vec{z}_1)} & \bullet & \xrightarrow{\hat{A}_2(\vec{z}_2)} & \cdots & \xrightarrow{\hat{A}_n(\vec{z}_n)} & s'' \\ {\scriptstyle A(\vec{x})}\downarrow & \text{all } F(\vec{y}) \text{ identical} & & & & \text{all } F'(\vec{y}') \text{ identical} & \\ s' & \xrightarrow{\hat{A}_1(\vec{z}_1)} & \bullet & \xrightarrow{\hat{A}_2(\vec{z}_2)} & \cdots & \xrightarrow{\hat{A}_n(\vec{z}_n)} & s''' \end{array}$$

Definition 3.3. *Consider two games $\sigma_1 = \langle \mathcal{F}_{\sigma_1}, \mathcal{A}_{\sigma_1} \rangle$ and $\sigma_2 = \langle \mathcal{F}_{\sigma_2}, \mathcal{A}_{\sigma_2} \rangle$ that are subgames of $\hat{\sigma} = \langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$. The game σ_1 is called **non-dependent** on the game σ_2 iff the following holds:*
If, for any action function $A \in \mathcal{A}_{\sigma_2}$, ground terms $\vec{x}$ and ground situations s, s'

$$Poss(A(\vec{x}), s) \wedge Do(A(\vec{x}), s) = s'$$

then, for all action functions $\{\hat{A}_1, \ldots, \hat{A}_n\} \subseteq \mathcal{A}_{\hat{\sigma}}$*, ground terms* $\vec{y}, \vec{y}', \vec{z}_1, \ldots, \vec{z}_n$*, ground situations* s'', s''' *and fluent functions* $F, F' \in \mathcal{F}_{\sigma_1}$*,*

1. $Holds(F(\vec{y}), s) \equiv Holds(F(\vec{y}), s')$
2. $Poss([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s) \wedge Do([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s) = s''$
   $\wedge\, Poss([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s') \wedge Do([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s') = s'''$
   $\implies Holds(F'(\vec{y}'), s'') \equiv Holds(F'(\vec{y}'), s''')$

and, for all action functions $\{A_1, \ldots, A_m\} \subseteq \mathcal{A}_{\sigma_1}$ *and ground terms* $\vec{w}_1, \ldots, \vec{w}_m$*,*

3. $Poss([A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)], s) \implies$
   $Terminal(s') \vee Poss([A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)], s')$

If additionally σ_2 *is non-dependent on* σ_1*, the two games are called* ***independent*** *of each other.*

Note that non-dependency and independency are not equivalent – in the case $\mathcal{A}_{\sigma_2} = \emptyset$, σ_1 is automatically non-dependent on σ_2, although the converse is not always true.

On another side remark, only considering sequences of actions with finite length n and m in the above definition is admissible, because in GDL, all games have only a finite number of steps (refer to section 2.1).

Independent subgames can, in a way, be thought of as parallel games, each having its own game tree, where any move made in one of the independent games does not affect the position in the game trees of the other ones – except for possibly reaching a terminal state in the composite game. However, there is a second type of independency:

Definition 3.4. *Consider a game* $\langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$*. The fluent function* F *is called* ***action-independent*** *iff, for all action functions* $A_1, A_2 \in \mathcal{A}_{\hat{\sigma}}$*, ground situations* s, s'_1, s'_2*, and ground terms* $\vec{x}, \vec{y}, \vec{z}$*,*

$$Poss(A_1(\vec{x}), s) \wedge Do(A_1(\vec{x}), s) = s'_1 \wedge Poss(A_2(\vec{y}), s) \wedge Do(A_2(\vec{y}), s) = s'_2$$
$$\implies Holds(F(\vec{z}), s'_1) \equiv Holds(F(\vec{z}), s'_2)$$

A subgame $(\mathcal{F}_{ais}, \emptyset)$*, where each element of* $\mathcal{F}_{ais}$ *is an action-independent fluent function, is called an* ***action-independent subgame****.*

In other words, all actions of $\hat{\sigma}$ have the same effect on the fluent function F; thus, F is independent of the choice of action. This typically occurs with fluent functions such as `step` in "incredible" that are used as a step counter to end the game after a certain number of actions.

Definition 3.5. *A set* $\Phi = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ *with* $\sigma_1 = \langle \mathcal{F}_{\sigma_1}, \mathcal{A}_{\sigma_1} \rangle$*,* $\sigma_2 = \langle \mathcal{F}_{\sigma_2}, \mathcal{A}_{\sigma_2} \rangle$*,* $\ldots$*,* $\sigma_n = \langle \mathcal{F}_{\sigma_n}, \mathcal{A}_{\sigma_n} \rangle$ *is called a* ***subgame decomposition*** *of a game* $\hat{\sigma} = \langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$ *iff*

1. *all* σ_i $(1 \leq i \leq n)$ *are either*
   a) *action-independent subgames of* $\hat{\sigma}$ *or*
   b) *non-action-independent subgames of* $\hat{\sigma}$ *that are pairwise independent,*
2. *all* σ_i $(1 \leq i \leq n)$ *are non-empty (i.e.* $\mathcal{F}_{\sigma_i} \neq \emptyset \vee \mathcal{A}_{\sigma_i} \neq \emptyset$*),*
3. $\bigcup_{1 \leq i \leq n} \mathcal{F}_{\sigma_i} = \mathcal{F}_{\hat{\sigma}}$ *and* $\bigcup_{1 \leq i \leq n} \mathcal{A}_{\sigma_i} = \mathcal{A}_{\hat{\sigma}}$*, and*
4. $\mathcal{F}_{\sigma_i} \cap \mathcal{F}_{\sigma_j} = \emptyset$ *and* $\mathcal{A}_{\sigma_i} \cap \mathcal{A}_{\sigma_j} = \emptyset$ $(1 \leq \{i, j\} \leq n, i \neq j)$.

A subgame decomposition Φ *of* $\hat{\sigma}$ *is called* ***maximal*** *iff there is no other subgame decomposition* Φ' *of* $\hat{\sigma}$ *with* $|\Phi'| > |\Phi|$.

At first glance, it may seem strange that this definition does not explicitly demand that the non-action-independent subgames (item 1b) are not affected by any fluents from the action-independent subgames (item 1a). But the definition of independency (def. 3.3) ensures that no such "indirect" effects can occur, so the requirement that the non-action-independent subgames be pairwise independent is sufficient.

3.2 Subgame Detection Algorithm

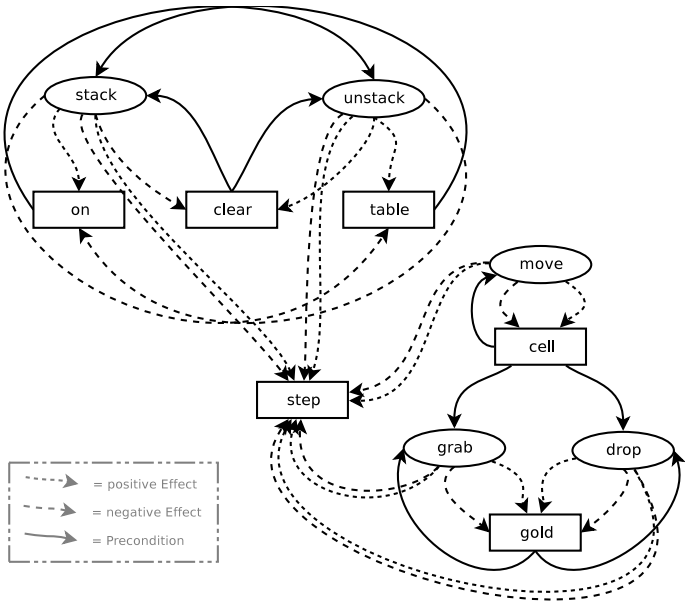
The idea of the algorithm is to build a dependency graph of the game's action functions and fluent functions and then identify this graph's connected components. These connected components correspond to the game's subgames. In building this graph, the algorithm assumes a dependency between an action function A and a fluent function F, if F occurs in a state update or precondition axiom of A.

We will shortly see that while this leads to a correct solution (in the sense that the algorithm always produces a subgame decomposition), and the algorithm has very good computational behavior, the resulting subgame decomposition may not be maximal. The reason for this is that not all potential effects detected by the algorithm really come into play. Further discussion of this topic is delayed until section 3.4, after the algorithm has been presented.

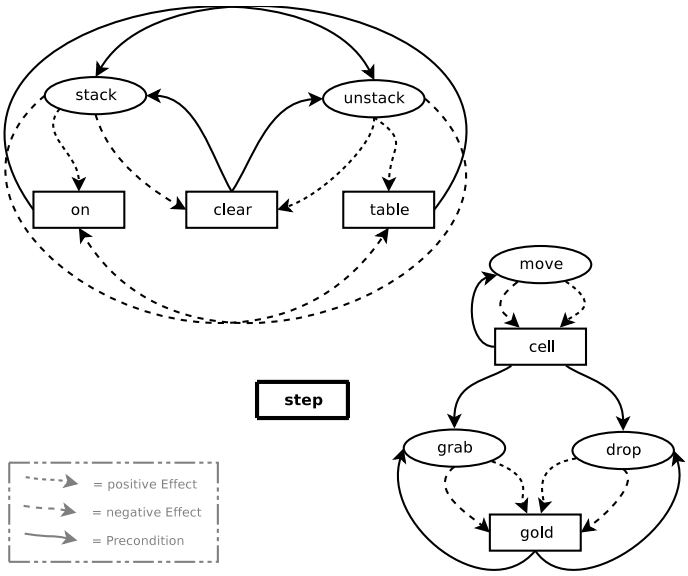
It can easily be seen that this approach, in its naïve form, fails in the presence of action-independent fluent functions: By definition, these fluent functions appear in the positive and negative effects of all actions. Hence, all actions would end up in the same connected component, preventing any decomposition. This is illustrated (for the example game of "incredible" [1]) in figure 3.1a.

On the other hand, if an action-independent fluent function does not occur in the preconditions of any action, the (identical) effects of all actions on the fluent could be ignored and the game could be decomposed, as is shown in figure 3.1b.

[1] Throughout the remainder of this work, I will use the game of "incredible" as an example. For reference, the source code is included in appendix A. "Incredible" consists of the two simple subgames "maze" and "blocks world".



(a) without action-independency information



(b) with fluent function `step` marked as action-independent

Figure 3.1: Feature and action dependency graphs for the game "incredible". Fluent functions are shown in boxes and action functions in circles; the dotted lines represent positive effects, the dashed lines represent negative effects and the continuous lines represent preconditions. Please note that although the lines are shown with arrowheads in the figure, the edges of the graph used by the algorithm are undirected.

To make use of this circumstance, the algorithm is enhanced by the possibility to use a set $\mathcal{S}$ of action-independent fluent functions. Since the algorithm is unable to prove action-independency of fluent functions itself, this set needs to be supplied to the algorithm by some external procedure.

The full algorithm is thus as follows:

Algorithm 3.1 Subgame Detection Algorithm

Algorithm. *Let $\hat{\sigma} = \langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$ be the game to be decomposed, and let $\mathcal{S} \subseteq \mathcal{F}_{\hat{\sigma}}$ be the set of action-independent fluents, if that information is available. Construct the undirected graph $G = (V, E)$ as follows:*

1. *Add each fluent function $F \in \mathcal{F}_{\hat{\sigma}}$ and action function $A \in \mathcal{A}_{\hat{\sigma}}$ to the set of vertices V.*
2. *Repeat the following step for all precondition axioms:*
   a) *Let $Poss(A(\vec{x}), z) \equiv \Pi(z)$ be the current precondition axiom.*
   b) *For each fluent function F that occurs in $\Pi(z)$, add the edge $\{F, A\}$ to E.*
   c) *If $F \in \mathcal{S}$, abort and return $\Phi = \{\hat{\sigma}\}$.*
3. *Repeat the following steps for all positive effects, negative effects and all conditions of all state update axioms:*
   a) *Let ϑ_i^+ and ϑ_i^- be the positive and negative effects of action $A(\vec{x})$ under condition $\Delta_i(s)$.*
   b) *For each fluent function F that occurs in $\Delta_i(s)$, add the edge $\{F, A\}$ to E.*
   c) *If $F \in \mathcal{S}$, abort and return $\Phi = \{\hat{\sigma}\}$.*
   d) *For each fluent function F that is not contained in the set of action-independent fluents (if available) and occurs in either ϑ_i^+ or ϑ_i^-, add the edge $\{A, F\}$ to E.*
4. *Find the connected components of this graph.*
5. *For each connected component (V_{cc}, E_{cc}), let $\mathcal{F}$ and $\mathcal{A}$ be the set of fluent functions resp. action functions in V_{cc}. Add $\langle \mathcal{F}, \mathcal{A} \rangle$ to the set Φ. This set Φ is the required subgame decomposition.*

Example 3.1. As an example, the result of running the subgame detection algorithm on the game $\sigma_{incredible} = (\{\texttt{cell}, \texttt{gold}, \texttt{on}, \texttt{clear}, \texttt{table}, \texttt{step}\}, \{\texttt{move}, \texttt{grab}, \texttt{drop}, \texttt{stack}, \texttt{unstack}\})$ with action-independent fluent set $\mathcal{S} = \{\texttt{step}\}$ is the subgame

decomposition $\Phi = \{\sigma_{maze}, \sigma_{blocks}, \sigma_{ais}\}$ with

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

3.3 Proof of Correctness

Before the proper proof of correctness, let us prove the following corollary:

Corollary 3.1. *Let Φ be the result of the application of the subgame detection algorithm to a game $\hat{\sigma} = \langle \mathcal{F}_{\hat{\sigma}}, \mathcal{A}_{\hat{\sigma}} \rangle$ with action-independent fluent functions $\mathcal{S}$. Then the following are equivalent:*

1. *No $F \in \mathcal{S}$ occurs in any $\Delta_i(s)$ of a state update axiom or right-hand side $\Pi(z)$ of a precondition axiom.*
2. *$(\forall F \in \mathcal{F}_{\hat{\sigma}}) F \in \mathcal{S} \implies (\{F\}, \emptyset) \in \Phi$*

Proof. We will show each direction in turn.

"⇒" Let F be an arbitrary element of $\mathcal{S}$. The steps 3b and 2b of the algorithm do not add any edge to the graph, because F does not occur in any $\Delta_i(s)$ or $\Pi(z)$. Step 3d does not add an edge, because $F \in \mathcal{S}$. Therefore, there are no edges including F in the graph, which means that the graph contains the connected component $\{F\}$, and therefore $(\{F\}, \emptyset) \in \Phi$.

"⇐" The fact that all $F \in \mathcal{S}$ are in their own game, and therefore in their own connected component, implies that no edges were added to the graph during the execution of the algorithm. Thus, no F can occur in any $\Delta_i(s)$ or $\Pi(z)$. □

Corollary 3.2. *If there is an $F \in \mathcal{S}$ that occurs in any $\Delta_i(s)$ of a state update axiom or right-hand side $\Pi(z)$ of a precondition axiom, then $\Phi = \{\hat{\sigma}\}$.*

Proof. If an $F \in \mathcal{S}$ occurs in any $\Delta_i(s)$ or $\Pi(z)$, the algorithm aborts in step 2c resp. 3c with result $\Phi = \{\hat{\sigma}\}$. □

In the following lemma, correctness should mean that the algorithm always returns a subgame decomposition. Maximality of that decomposition is explicitly not required.

Lemma 3.1. *Correctness: The result Φ of the subgame detection algorithm is always a subgame decomposition of the game $\hat{\sigma}$.*

Proof. We need to show that Φ meets each of the four criteria of definition 3.5 (see page 13).

1. *All $\sigma_i (1 \leq i \leq n)$ are either*
   a) action-independent subgames of $\hat{\sigma}$ or
   b) non-action-independent subgames of $\hat{\sigma}$ that are pairwise independent

   The fact that all σ_i are indeed subgames of $\hat{\sigma}$ follows from criterion 3, which will be proved below. Also, in the case where at most one σ_i is not action-independent, the statement is trivially true. Thus assume there are two subgames $\sigma_1 = \langle \mathcal{F}_{\sigma_1}, \mathcal{A}_{\sigma_1} \rangle$ and $\sigma_2 = \langle \mathcal{F}_{\sigma_2}, \mathcal{A}_{\sigma_2} \rangle$, such that $\sigma_1, \sigma_2 \in \Phi$, $\sigma_1 \neq \sigma_2$ and σ_1, σ_2 not action-independent.

   It remains to show that w.l.o.g. σ_1 is non-dependent on σ_2. By definition 3.3, for all actions $A \in \mathcal{A}_{\sigma_2}$, ground terms $\vec{x}$ and ground situations s, s' such that $Poss(A(\vec{x}), s) \wedge Do(A(\vec{x}), s) = s'$, all of the following must hold for all action functions $\{\hat{A}_1, \ldots, \hat{A}_n\} \subseteq \mathcal{A}_{\hat{\sigma}}$, $\{A_1, \ldots, A_m\} \subseteq \mathcal{A}_{\sigma_1}$, ground terms $\vec{w}_1, \ldots, \vec{w}_m$, $\vec{y}$, $\vec{y}'$, $\vec{z}_1, \ldots, \vec{z}_n$, ground situations s'', s''' and fluent functions $F, F' \in \mathcal{F}_{\sigma_1}$:

   a) $Holds(F(\vec{y}), s) \equiv Holds(F(\vec{y}), s')$

      Since each fluent function $F_{ais} \in \mathcal{S}$ occurs in an action-independent subgame $(\{F_{ais}\}, \emptyset)$ (by corollaries 3.1 and 3.2), and no fluent function occurs in more than one subgame (by criterion 4, also proved below), no action-independent fluent function is contained in σ_1 and σ_2 ($\mathcal{F}_{\sigma_1} \cap \mathcal{S} = \emptyset$ and $\mathcal{F}_{\sigma_2} \cap \mathcal{S} = \emptyset$). Hence, $F \notin \mathcal{S}$.

      Moreover, as $\sigma_1 \neq \sigma_2$ and $F \in \mathcal{F}_{\sigma_1}$ and $A \in \mathcal{A}_{\sigma_2}$, the fluent function F and the action function A are in two different connected components. Hence, F does not occur in any positive or negative effect of A; otherwise, since $F \notin \mathcal{S}$, step 3d would have added an edge between them, and they would have ended up in the same connected component. Therefore, all $F(\vec{y})$ remain unchanged by the execution of any $A(\vec{x})$.

   b) $Poss([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s) \wedge Do([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s) = s''$
   $\wedge\, Poss([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s') \wedge Do([\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)], s') = s'''$
   $\implies Holds(F'(\vec{y}'), s'') \equiv Holds(F'(\vec{y}'), s''')$

      Let L_1 be the sublist of $[\hat{A}_1(\vec{z}_1), \ldots, \hat{A}_n(\vec{z}_n)]$ that only consists of actions of the subgame σ_1, and let L_2 be the sublist with all actions that are not in σ_1. None of the fluents with fluent functions from $\mathcal{F}_{\sigma_1}$ have changed between s and s' (see 1a), and the $\Delta_i(s)$'s, ϑ_i^+'s and ϑ_i^-'s of the actions in L_1 only contain fluent functions from $\mathcal{F}_{\sigma_1}$ (and only in the case of the positive and negative effects from $\mathcal{S}$). Also, the domain axiomatization is deterministic. Therefore, the execution of the actions in L_1 has the same effect on all fluents with fluent functions from $\mathcal{F}_{\sigma_1}$. There can also be

no interference by actions from L_2, because their $\Delta_i(s)$'s, ϑ_i^+'s and ϑ_i^-'s contain no fluents with fluent functions from $\mathcal{F}_{\sigma_1}$.

c) $Poss([A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)], s) \implies$
$Terminal(s') \vee Poss([A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)], s')$

If $Terminal(s')$ holds, we are done; otherwise, due to the facts that all fluents with fluent functions from $\mathcal{F}_{\sigma_1}$ are equal in s and s' and that only these are evaluated by the precondition and state update axioms of the actions in $[A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)]$ (as was stated before), the sequence $[A_1(\vec{w}_1), \ldots, A_m(\vec{w}_m)]$ is executable in s'.

2. *All* $\sigma_i, 1 \leq i \leq n$ *are non-empty (i.e.* $\mathcal{F}_{\sigma_i} \neq \emptyset \vee \mathcal{A}_{\sigma_i} \neq \emptyset$*)*

   As there are no empty connected components, this is trivial.

3. $\bigcup_{1 \leq i \leq n} \mathcal{F}_{\sigma_i} = \mathcal{F}_{\hat{\sigma}}$ *and* $\bigcup_{1 \leq i \leq n} \mathcal{A}_{\sigma_i} = \mathcal{A}_{\hat{\sigma}}$

   Each fluent and action function gets added to the graph, which is partitioned into connected components. Each of these are converted into an element of Φ, so each fluent and action function is retained.

4. $\mathcal{F}_{\sigma_i} \cap \mathcal{F}_{\sigma_j} = \emptyset$ *and* $\mathcal{A}_{\sigma_i} \cap \mathcal{A}_{\sigma_j} = \emptyset$ $(1 \leq \{i, j\} \leq n, i \neq j)$

   Obviously, each node of the graph (which corresponds to either an action function or a fluent function) is assigned a unique connected component. □

3.4 Properties of the Algorithm

3.4.1 Complexity

The algorithm works in time linear in the number of precondition and state update axioms plus the time it takes to find the connected components. This is commonly done as an application of depth-first search. The time complexity is the time complexity of depth-first search: $O(|V| + |E|)$ (see for example Cormen *et al.*, 2001).

Here, $|V|$ and $|E|$ are bounded by the number of fluent functions $|\mathcal{F}_{\hat{\sigma}}|$ and action functions $|\mathcal{A}_{\hat{\sigma}}|$ used in the precondition and state update axioms. More specifically, $|V| \leq |\mathcal{F}_{\hat{\sigma}}| + |\mathcal{A}_{\hat{\sigma}}|$ and $|E| \leq |\mathcal{F}_{\hat{\sigma}}| * |\mathcal{A}_{\hat{\sigma}}|$.

Space complexity is $O(h)$, where h is the length of the longest simple path in the graph.

3.4.2 Solution Quality

As has been mentioned before, the algorithm is correct, but not optimal in that there are cases in which the returned solution is not maximal. This is due to the

fact that, when analyzing the state update axioms, the algorithm can only detect *potential* dependencies between fluent and action functions. To illustrate this, take the following precondition axiom with fluent function foo and action function $foobar$:

$$Poss(foobar(x), z) \equiv (Holds(foo(23), z) \vee \neg Holds(foo(23), z)) \wedge x > 0$$

The algorithm recognizes a potential dependency between fluent function foo and action function $foobar$, because foo appears on the right-hand side of the precondition axiom of $foobar$. But clearly, the fluent $foo(23)$ is not really a precondition for executing $foobar(x)$, therefore foo and $foobar$ could still be in independent subgames.

4 Greedy Decomposition Search

In the preceding chapter, we saw how a game can be decomposed into relatively independent subgames. Still, standard search algorithms cannot make use of this information. This chapter and the next will cover two different search algorithms that use subgame information to improve performance.

The first of these algorithms, which is the topic of this chapter, is called the *greedy decomposition search*. The idea of this algorithm is relatively simple: Each subgame is searched separately, assuming no move is made in any of the other subgames, using any standard iterative deepening search algorithm. After each iteration step, the highest-valued actions of all subgames are compared and the action with the highest expected value is *greedily* chosen for execution (hence the algorithm's name).

This idea is depicted in pseudocode notation in algorithm 4.1 on the next page. The parameter Φ is a subgame decomposition, as introduced in definition 3.5 on page 13. The function LimitedDepthSearch is any standard implementation of a depth-first or breadth-first search algorithm, with the only modification that only the actions with action functions from the subgame σ are included in the search. The second parameter is an integer *depth*, which limits the search depth. The function returns a number *val* ($0 \leq val \leq 100$), which denotes the expected value of the best move found by the search, and the best move *move* itself.

As it is common in iterative deepening search, the algorithm can be aborted at any time, returning the current $move_{max}$.

4.1 Properties of the Algorithm

4.1.1 Complexity

The runtime of the algorithm is determined by the runtime of the inner search algorithm, LimitedDepthSearch. The current implementation uses depth-first search, which has a time complexity of $O(b_1^{d_1} + \cdots + b_n^{d_n})$, where $b_1, \ldots, b_n$ and $d_1, \ldots, d_n$ denote the breadth and solution depth of the search tree of the subgames $\sigma_i \in \Phi$ ($1 \leq i \leq n$). This is exponentially faster than searching the composite game $\hat{\sigma}$ directly, which would lead to a complexity of $O((b_1 + \cdots + b_n)^{(d_1+\cdots+d_n)})$.

The space complexity of this algorithm equals that of the inner search algorithm, which is (in the case of depth-first search) $O(\max(d_1, \ldots, d_n))$.

Algorithm 4.1 Greedy Decomposition Search

```
1: function GreedyDecompositionSearch(Φ)
2:     depth ← 1
3:     val_max ← (−1)
4:     move_max ← ∅
5:     while (val_max < 100) and (∃ unfinished subgames) do
6:         for each σ ∈ Φ do
7:             ⟨val, move⟩ ← LimitedDepthSearch(σ, depth)
8:             if val > val_max then
9:                 val_max ← val
10:                move_max ← move
11:            end if
12:        end for
13:        depth ← depth + 1
14:    end while
15:    return move_max
16: end function
```

4.1.2 Solution Quality

Like most greedy algorithms, the greedy decomposition search is not optimal in the general case. The reason for this is that there can be an indirect interaction between subgames via the *Goal* function or the *Terminal* predicate. More precisely, if a certain combination of fluents from different subgames must hold in a state z in order for that state to be assigned a high goal value, and if achieving only certain subsets of these fluents does not lead to a higher goal value, then the algorithm may not be able to achieve this goal.

Example 4.1. Imagine a game that consists of two subgames, $\sigma_1 = \langle \mathcal{F}_{\sigma_1}, \mathcal{A}_{\sigma_1} \rangle$ and $\sigma_2 = \langle \mathcal{F}_{\sigma_2}, \mathcal{A}_{\sigma_2} \rangle$, and $F_1 \in \mathcal{F}_1$, $F_2 \in \mathcal{F}_2$. Let the *Goal* function be defined such that a state z is assigned 100 points only if for some $\vec{x}$ and $\vec{y}$, $F_1(\vec{x}) \wedge F_2(\vec{y})$ holds in z, and 0 points otherwise. Starting from a state where neither $F_1(\vec{x})$ nor $F_2(\vec{y})$ hold, the algorithm will try to maximize the outcome of each subgame separately, while assuming that the other subgame remains unchanged; since achieving one of the fluents without achieving the other does not increase the goal value, the algorithm will not find a solution.

Likewise, sometimes the *Terminal* predicate makes it impossible for the algorithm to find a good solution. This is the case when the game tree is a "maze" of bad terminal states, and only a combination of moves from different subgames can find a path toward a good terminal state. This problem will be explored more formally, along with an attempt to solve it, in the next chapter.

Conditions for optimality

However, there is a class of games which can be solved optimally with the greedy decomposition search algorithm. The example game "incredible" is such a case. Table 4.1 depicts the goal values of "incredible" with respect to the truth values of the sub-formulas that occur in the GDL definition of the `goal` predicate. Obviously, the goals are additive: `(true (gold w))` is worth 45 points, `(completed caetower)` is worth 30 points and `(completed bdftower)` is worth 25 points.

`(true (gold w))`	`(completed caetower)`	`(completed bdftower)`	`goal`
false	*false*	*false*	0
false	*false*	*true*	25
false	*true*	*false*	30
false	*true*	*true*	70
true	*false*	*false*	45
true	*false*	*true*	70
true	*true*	*false*	75
true	*true*	*true*	100

Table 4.1: Goal values for "incredible"

In games with such additive (or rather "strictly ordered") goal values and a "sufficiently nice" terminal predicate, the algorithm returns an optimal result. Incidentally, most of the composite games that have appeared in the AAAI competition fall into this class.

The terms "strictly ordered" and "sufficiently nice" are still very imprecise; also, it is not clear what role the sub-formulas `(true (gold w))`, `(completed caetower)` and `(completed bdftower)` play and where they come from, which calls for a deeper analysis of the `goal` and `terminal` predicates. This will be conducted in the next chapter, in an attempt to develop an optimal search algorithm for composite games.

5 Concept Decomposition Search

5.1 Algorithm

The main shortcoming of greedy decomposition search is that it can only find an optimal solution for a very restricted class of games. Hence, the goal of this chapter is to deliver an optimal and correct algorithm for a much broader class of composite games. However, the cost is a very high space and time complexity, which will be discussed in section 5.3.1.

As we have seen in the preceding chapter, the reason for the non-optimality of greedy decomposition search is that each subgame is searched separately, ignoring any possible interaction via the goal and terminal predicates. To account for this interaction, *concept decomposition search* combines the results of the separate subgame searches in a more sophisticated manner.

5.1.1 Overview of the Algorithm

The search process 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 combine these local plans to find a *global plan* that leads to a globally optimal terminal state.

As algorithm 5.1 on the next page shows, these two steps are embedded into an iterative deepening framework, similar to that of algorithm 4.1 on page 22, with *depth* as the iteration counter and the variables $move_{max}$ and val_{max} for holding the currently best move and its value.

The local search is implemented by the function LocalSearch (section 5.1.2), which searches the game tree of subgame σ up to the given *depth*, returning all relevant local plans that are not yet contained in *local_plans*. (The variable *local_plans* is an array indexed by the subgames, containing all local plans that have been found so far.)

GlobalSearch (section 5.1.3) then takes these new local plans, combining them with all "old" local plans from other subgames, and returning the value and first action of the best global plan that was found in $\langle val, move \rangle$. Then, the search continues.

If time runs out before all subgames have been exhaustively searched, the iterative deepening framework returns the best action found so far, stored in $move_{max}$.

Algorithm 5.1 Concept Decomposition Search

```
1: function ConceptDecompositionSearch(Φ)
2:     depth ← 1
3:     val_max ← (−1)
4:     move_max ← ∅
5:     local_plans ← ∅
6:     while (val_max < 100) and (∃ unfinished subgames) do
7:         for each σ ∈ Φ do
8:             new_local_plans ← LocalSearch(σ, depth, local_plans)
9:             local_plans[σ] ← local_plans[σ] ∪ new_local_plans
10:            ⟨val, move⟩ ← GlobalSearch(σ, local_plans, new_local_plans)
11:            if val > val_max then
12:                val_max ← val
13:                move_max ← move
14:            end if
15:        end for
16:        depth ← depth + 1
17:    end while
18:    return move_max
19: end function
```

A word on notation: Of course, all of these search algorithms –and also all local and global plans– only make sense when seen relative to an initial state, which is the current state of the game. For the sake of clarity, this initial state is omitted in all algorithms and implicitly assumed to be passed as an additional parameter to every function.

Also, due to time constraints, the search may be aborted and continued after executing the currently best action. In order to re-use as much information from previous searches as possible, the implementation includes an elaborate update mechanism. However, this is not central to this treatment and will therefore also be ommitted.

Further, in the course of this chapter, the GDL and fluent calculus representations of a game will be used interchangably, depending which of them is better suited to represent the topic. In the context of General Game Playing, both representations are assumed to be equivalent.

5.1.2 Local Search

The local search (algorithm 5.2 on the next page) traverses the game tree of a single subgame σ, limiting the search to a given *depth*. This traversal can be done in a manner similar to that of LimitedDepthSearch from section 4. Only the

Algorithm 5.2 Local Search

```
1: function LocalSearch(σ, depth, local_plans)
2:     new_local_plans ← ∅
3:     traverse game tree using LimitedDepthSearch(σ, depth)
4:     for each leaf_node that is reached via action sequence plan do
5:         plan_signature ← CalculatePlanSignature(plan)
6:         if ⟨plan_signature, _⟩ ∉ new_local_plans ∪ local_plans[σ] then
7:             new_local_plans ← new_local_plans ∪ ⟨plan_signature, plan⟩
8:         end if
9:     end for
10:    return new_local_plans
11: end function
```

action performed when a leaf node is reached needs to be adjusted: In the original LimitedDepthSearch, the leaf node is evaluated, either via the `goal` predicate (if the node is terminal), or heuristically otherwise. Here, the algorithm instead 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: In composite games, 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 will be proven in section 5.2.

Example 5.1. As an example from "incredible", the two plans

$$[U(F,E), U(E,D), U(C,A), S(A,E), S(C,A)]$$

and

$$[U(C,A), U(F,E), U(E,D), S(A,E), S(C,A)],$$

executed in the initial state

$$State(S_0) = Cell(w) \circ Gold(y) \circ Clear(B) \circ Clear(C) \circ Clear(f) \circ On(C,A) \circ On(E,D) \circ On(F,E) \circ Table(A) \circ Table(B) \circ Table(D) Step(1)$$

lead to situations S_1 and S_2 with

$$Holds(On(C,A) \circ On(A,E), State(S_1))$$

and

$$Holds(On(C,A) \circ On(A,E), State(S_2)),$$

and therefore should have the same plan signature.

This *plan_signature* is calculated by a function CalculatePlanSignature, whose inner workings will be examined in section 5.1.5; for now, it suffices to treat it as a black box.

Algorithm 5.3 Global Search

```
1: function GlobalSearch(σ, local_plans, new_local_plans)
2:     val_max ← (−1)
3:     move_max ← ∅
4:     for each plan ∈ new_local_plans do
5:         subsets ← ChoosePlans(σ, local_plans, plan)
6:         for each plan_set ∈ subsets do
7:             ⟨val, move⟩ ← CombinePlans(plan_set)
8:             if val > val_max then
9:                 val_max ← val
10:                move_max ← move
11:            end if
12:        end for
13:    end for
14:    return ⟨val_max, move_max⟩
15: end function
```

5.1.3 Global Search

In between local search iterations, global search (algorithm 5.3) 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. It repeatedly takes one of the *new_local_plans* found in the last run of LocalSearch and tries to combine it with the previously found local plans from other subgames in various ways to produce an optimal combined plan.

In doing so, the function ChoosePlans simply calculates all subsets of *local_plans* that include the currently considered *plan* from subgame σ and at most one plan from each of the other subgames. Then, CombinePlans (section 5.1.4) 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). This issue will be discussed more elaborately in the corresponding section.

In the process, the value and first action of the currently best global plan is stored in $\langle val_{max}, move_{max} \rangle$ and returned in the end.

5.1.4 Combination of Plans

As was mentioned in the preceding section, 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.

Since the plans come from separate subgames and thus cannot have any influence on each others fluents, it may at first seem that it is sufficient to search all per-

mutations of the plans as a whole. Unfortunately, this is not the case; 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 in an example.

Example 5.2. Consider the game from listing 5.1 on the next page. It is a game played on the following two, very simple, 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 situation is `[(pos1 a), (pos2 x)]`, and each `move` action moves to the corresponding node on the graph. The two situations `[(pos1 a), (pos2 z)]` and `[(pos1 c), (pos2 x)]` are terminal and have the goal value 0. Only the situation `[(pos1 c), (pos2 z)]` has a goal value of 100.

The game can be decomposed into the two subgames $\sigma_1 = \langle\{\texttt{move1}\}, \{\texttt{pos1}\}\rangle$ and $\sigma_2 = \langle\{\texttt{move2}\}, \{\texttt{pos2}\}\rangle$. Both have only one local plan, [`(move1 b)`, `(move1 c)`] resp. [`(move2 y)`, `(move2 z)`], so there are only the following two global plans that are permutations of the whole local plans:

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

and

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

It is easy to see that both global plans reach one of the "bad" terminal situations before the whole plan has been executed. This demonstrates that CombinePlans needs to interleave the local plans, which leads to the following solution (amongst others):

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

This said, the naïve approach of simply enumerating all permutations would have a factorial time complexity; for n local plans with lengths $m_1, m_2, \ldots, m_n$, the number of permutations is:

$$\frac{(\sum_{i=1}^{n} m_i)!}{m_1! * m_2! * \cdots * m_n!}$$

Luckily, using a dynamic programming technique, the problem can be solved in polynomial space and time. 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[1]. 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. The detailed algorithm can be found in appendix B on page 44.

[1] Each "node" of the search space is determined by the set of executed actions, without regarding their order.

Listing 5.1 GDL description for example 5.2

```
(role player)

(init (pos1 a))
(init (pos2 x))

(<= (legal player (move1 ?n))
    (true (pos1 ?m))
    (conn1 ?m ?n))

(<= (legal player (move2 ?n))
    (true (pos2 ?m))
    (conn2 ?m ?n))

(<= (next (pos1 ?n))
    (does player (move1 ?n)))

(<= (next (pos1 ?n))
    (true (pos1 ?n))
    (does player (move2 ?m)))

(<= (next (pos2 ?n))
    (does player (move2 ?n)))

(<= (next (pos2 ?n))
    (true (pos2 ?n))
    (does player (move1 ?m)))

(<= terminal
    (true (pos1 c))
    (true (pos2 z)))

(<= terminal
    (true (pos1 a))
    (true (pos2 z)))

(<= terminal
    (true (pos1 c))
    (true (pos2 x)))

(<= (goal player 100)
    (true (pos1 c))
    (true (pos2 z)))

(<= (goal player 0)
    (not (true (pos1 c))))

(<= (goal player 0)
    (not (true (pos2 z))))

; graph 1: a --> b --> c
; graph 2: x --> y --> z

(conn1 a b)
(conn1 b c)

(conn2 x y)
(conn2 y z)
```

5.1.5 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. In order to properly define plan signatures, the notion of "local goal and terminal concepts" will first have to be introduced. The definition of the plan signature will follow at the end of the section.

Local Goal and Terminal Concepts

One obvious feature that distinguishes one local plan from another is the final situation that is reached by executing the plan. More precisely, the distinguishing feature is the evaluation of the `goal` and `terminal` predicates. But unfortunately, since local plans only affect fluents from the subgame that the plan belongs to, the resulting situation is only partially determined. This makes it impossible to say in the general case what the goal value of a partial situation is, or whether it is terminal or not. It's even possible that a particular local plan from one subgame leads to a good terminal situation when combined with another particular plan from a different subgame, but to a bad terminal situation in combination with other local plans. Therefore, a local plan has no "value" of its own; instead, only global plans, i.e. whole combinations of local plans can be evaluated.

This was the main reason for introducing the division between local search and global search; it enables local search to collect only those local plans whose final situation might have a novel effect on the global `goal` and `terminal` predicates, while deferring evaluation of these predicates to the point when the complete situation is known.

But how to decide if a situation is relevant to the `goal` and `terminal` predicates? One idea might be to propositionalize a subgame, i.e. fully instantiate all fluents and actions. This would make it possible to check which (combinations of) fluents from a subgame are relevant for the `goal` or `terminal` predicate. Unfortunately, this is only feasible for the smallest of instances. Therefore, we need a more abstract representation.

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 5.1. *A **local goal (resp. terminal) concept**, or short "local concept", is a ground predicate call*[2] *that occurs in the body of the* `goal` *(resp.* `terminal`*)*

[2] In the terminology of Genesereth *et al.* (2005), the local goal (resp. terminal) concepts are "relational sentences".

Algorithm 5.4 FindLocalConcepts

```
1:  function FindLocalConcepts(call_graph)
2:      local_concepts ← ∅
3:      open_nodes ← all direct children of root of call graph (goal/terminal)
4:      while open_nodes ≠ ∅ do
5:          node ← arbitrary element of open_nodes
6:          open_nodes ← open_nodes \ {node}
7:          if node not ground or node has backedge (recursion) then
8:              abort
9:          end if
10:         n ← number of subgames with fluents in the subtree of node
11:         if n = 0 then
12:             discard node
13:         else if n = 1 then local_concepts ← local_concepts ∪ node
14:         else if n > 1 then
15:             open_nodes ← open_nodes ∪ all direct children of node
16:         end if
17:     end while
18:     return local_concepts
19: end function
```

predicate's definition. The expanded definition of a local goal (resp. terminal) concept must contain fluents with fluent functions 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 (see algorithm 5.4). The algorithm requires that the upper part of the call graph, i.e. the part up to the local concepts, is ground and does not contain any recursion[3]. The concepts themselves may be both non-ground and recursive.

Example 5.3. The goal and terminal definition of "incredible" is given in listing 5.2 on the following page. The corresponding call graphs are depicted in figure 5.1 on page 33. Application of algorithm 5.4 leads to the following local goal and terminal concepts:

- `(true (gold w))` (goal and terminal concept, subgame σ_{maze}[4] on page 16)
- `(completed bdftower)` (goal concept, subgame σ_{blocks})

[3] In a ground call graph, recursion would not make sense anyway, as it would lead to an infinite recursion.

[4] also refer to example 3.1

Listing 5.2 `goal` and `terminal` description of the game "incredible"

```
(<= (completed bdftower)
    (true (on d f))
    (true (on b d)))

(<= (completed caetower)
    (true (on a e))
    (true (on c a)))

(<= (goal robot 100)
    (true (gold w))
    (completed bdftower)
    (completed caetower))

(<= (goal robot 75)
    (true (gold w))
    (not (completed bdftower))
    (completed caetower))

(<= (goal robot 70)
    (true (gold w))
    (completed bdftower)
    (not (completed caetower)))

(<= (goal robot 55)
    (completed bdftower)
    (completed caetower)
    (not (true (gold w))))

(<= (goal robot 45)
    (true (gold w))
    (not (completed bdftower))
    (not (completed caetower)))

(<= (goal robot 30)
    (completed caetower)
    (not (completed bdftower))
    (not (true (gold w))))

(<= (goal robot 25)
    (completed bdftower)
    (not (completed caetower))
    (not (true (gold w))))

(<= (goal robot 0)
    (true (step c20))
    (not (completed caetower))
    (not (completed bdftower))
    (not (true (gold w))))

(<= terminal
    (true (step c20)))

(<= terminal
    (true (gold w)))
```

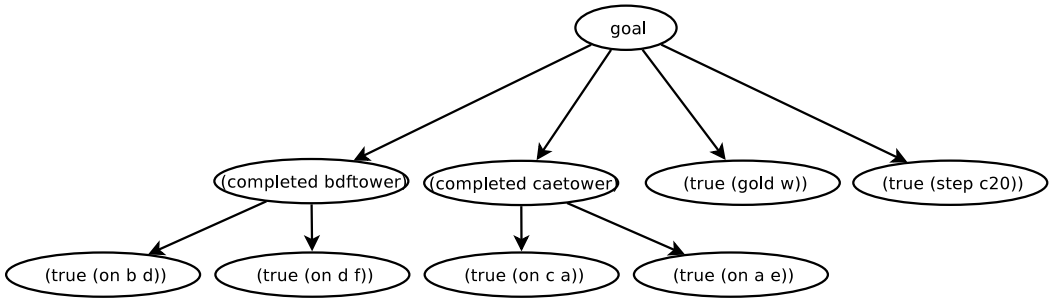
- `(completed caetower)` (goal concept, subgame σ_{blocks})
- `(true (step c20))` (goal and terminal concept, subgame σ_{ais})

Definition 5.2. *A **situation concept evaluation** is a sequence of boolean values* $[b_1, b_2, \ldots, b_n]$. *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 the plan's subgame* σ.

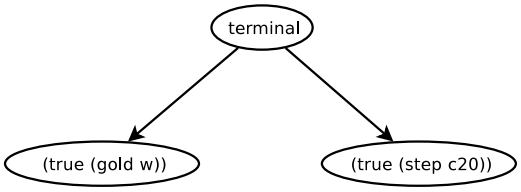
Example 5.3 (cont.). In the initial state of "incredible", the situation concept evaluation of σ_{maze} is $[false]$, that of σ_{blocks} is $[false, false]$ and that of σ_{ais} is also $[false]$.

One last auxiliary definition is the following:

Definition 5.3. *The **terminal concept evaluation sequence** is a sequence of situation concept evaluations, but restricted to the terminal concepts of the plan's subgame* σ, *of all situations that are traversed by executing the local plan.*



(a) `goal` predicate



(b) `terminal` predicate

Figure 5.1: Call graphs of the game "incredible"

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

Definition 5.4. *A **plan signature** of a local plan p is a pair $\langle s, t \rangle$, where*

- *s is the situation concept evaluation of the local plan's final situation; and*
- *t is the plan's terminal concept evaluation sequence.*

The motivation for defining the plan signature in this specific way will be clarified in the following proof of correctness.

5.2 Proof of Correctness

Lemma 5.1. *For any given game, algorithm 5.1 (concept decomposition search) will eventually find a global plan that leads to a terminal state with maximal goal value.*

Proof. The function LocalSearch (algorithm 5.2 on page 26) only excludes local plans with a duplicate plan signature; everything else is passed to the global search. Inside the function GlobalSearch (algorithm 5.3 on page 27), all possible combinations of local plans are generated by the function ChoosePlans and tested; subsequently, CombinePlans searches all possible combinations of actions inside these local plans, until one global plan has been found; thus, if there is a solution, it will eventually be found.

What remains to show is therefore that

1. of all plans with identical plan signature, only one needs to be searched; and

2. if CombinePlans finds one global plan for a given set of local plans, all other possible global plans for these same local plans do not need to be considered.

Proof of (1). Let Φ be a subgame decomposition. Let p_1 and p_2 be two local plans of the same subgame $\sigma \in \Phi$ with identical plan signature $\langle s, t \rangle$. Further, let $q_1, \ldots, q_n$ be local plans from $\Phi \setminus \sigma$, with no q_i, q_j with $i \neq j$ from the same subgame. Let $g_1 = [a_1, \ldots, a_m]$ be a global plan that is the result of a combination of the local plans $\{p_1, q_1, \ldots, q_n\}$, and let s_0 be the initial situation of the search and $s_1 = Do(a_1, s_0), s_2 = Do(a_2, s_1), \ldots, s_m = Do(a_m, s_{m-1})$ be the situations reached by executing the global plan g_1.

Now assume that the global plan is not terminal prematurely, i.e. $\neg Terminal(s_i)$ for all $0 \leq i \leq (m-1)$. Then, a global plan g_2 for the local plans $\{p_2, q_1, \ldots, q_n\}$ that is also not terminal prematurely can be constructed as follows:

Since the terminal concept evaluation sequence t of the plans p_1 and p_2 contains one element for each situation that is reached by one action in p_1 resp. p_2, both plans must have the same length: $|t| = |p_1| = |p_2|$. Therefore, g_2 can be constructed by replacing each action in g_1 that is an element of p_1 by the corresponding element from p_2. Let $[s'_1, \ldots, s'_m]$ be the situations reached by executing the global plan g_2.

Each predicate call inside the definition of the `terminal` predicate, and therefore also of the corresponding *Terminal* axiom, is either

1. a call to an auxiliary predicate that does not include any fluents;

2. a call to a terminal concept of an action-independent subgame;

3. a call to a terminal concept of a subgame $\neq \sigma$ from Φ; or

4. a call to a terminal concept of subgame σ.

We will show that in each of these cases, the evaluation of the call has the same result in all s_i and s'_i $(1 \leq i \leq m)$, and that therefore g_2 is also not terminal prematurely.

1. Since there are no fluents involved, the evaluation of auxiliary predicates is the same in any situation.

2. Since action-independent subgames by definition do not depend on the executed action, but only on the number of actions (which is identical in g_1 and g_2), the evaluation of this type of call is the same in both global plans.

3. The terminal concepts of subgames different from σ do not depend on fluents from σ, and are therefore identical in both global plans.

4. The terminal concepts of σ are identical in both g_1 and g_2, since p_1 and p_2 have the same terminal concept evaluation sequence.

Further, $Goal(s_m) = Goal(s'_m)$ (since p_1 and p_2 have the same situation concept evaluation of the local plans' final situation), and $Terminal(s_m) \equiv Terminal(s'_m)$ (this has already been proven). Hence, both plans are equivalent.

Proof of (2). By definition of independency (def. 3.3), the order of actions from different subgames is not relevant for the fluents that hold in a given situation. Therefore, all orders of executing the actions in the given set of local plans –provided all intermediary states are not terminal– are equivalent. □

5.3 Properties of the Algorithm

5.3.1 Complexity

Time Complexity

Before considering the runtime complexity of concept decomposition search, it is important to point out an important fact: It is not possible to decide locally (inside a subgame's local search) if a state is terminal; hence, the algorithm may search parts of the game tree that are not even reachable and therefore not traversed by a direct search.

Therefore, the time complexity of the algorithm is $O(b_1^{d_1} * \bar{b}_1^{\bar{d}_1} + \cdots + b_n^{d_n} * \bar{b}_n^{\bar{d}_n})$, where $b_1, \ldots, b_n$ and $d_1, \ldots, d_n$ denote the breadth and solution depth of the search tree of the subgames $\sigma_i \in \Phi$ $(1 \leq i \leq n)$, and $\bar{b}_1, \ldots, \bar{b}_n$ and $\bar{d}_1, \ldots, \bar{d}_n$ denote the breadth and depth of the "unreachable" parts of these subgames' search tree[5].

This has to be compared with the complexity of direct search, which is $O((b_1 + \cdots + b_n)^{(d_1 + \cdots + d_n)})$. If the breadth of the reachable and unreachable parts of the search trees are approximately equal (all $\bar{b}_i \approx b_i$), then the time complexity of concept

[5] Strictly speaking, these "unreachable" parts are not guaranteed to terminate. They are, however, limited by the shallowest solution in the reachable parts of the subgame, so the limits $\bar{d}_1, \ldots, \bar{d}_n$ exist.

decomposition search is not significantly worse than that of direct search, as then

$$
\begin{aligned}
& O(b_1^{d_1} * \bar{b}_1^{\bar{d}_1} + \cdots + b_n^{d_n} * \bar{b}_n^{\bar{d}_n}) && \text{(complexity of conc. dec. search)} \\
= \quad & O(b_1^{(d_1+\bar{d}_1)} + \cdots + b_n^{(d_n+\bar{d}_n)}) \\
\leq \quad & O(b_1^{\max(d_1,\ldots,d_n)} + \cdots + b_n^{\max(d_1,\ldots,d_n)}) \\
\leq \quad & O(n * \max(b_1, \ldots, b_n)^{\max(d_1,\ldots,d_n)}) \\
= \quad & O(\max(b_1, \ldots, b_n)^{\max(d_1,\ldots,d_n)}) && \text{since } n \text{ small constant factor} \\
\leq \quad & O((b_1 + \cdots + b_n)^{\max(d_1,\ldots,d_n)}) \\
\leq \quad & O((b_1 + \cdots + b_n)^{(d_1+\cdots+d_n)}) && \text{(complexity of direct search).}
\end{aligned}
$$

If, however, some $\bar{b}_i \gg b_i$, time complexity of concept decomposition search can be arbitrarily worse than that of direct search. This could only be avoided by reducing the class of admissible games such that the terminality of a partial state can be determined locally.

Global search was not included in above calculation, since its complexity is only polynomial and is dominated by the complexity of local search. Still, it should be mentioned that global search may waste some computation time in the cases where a solution for a set of shorter local plans is already known. When computing an answer for a similar set, where one or more of these plans are extended at the end, it makes no use of the information gained in shallower searches; in effect, this means that some upper parts of the search tree can be searched several times. However, this does not matter for overall complexity, because search trees get exponentially bigger with increasing search depth, and because global search has only polynomial runtime complexity. Still, this leaves room for further optimization and could be alleviated by caching some of the results, effectively trading space for time.

Space Complexity

The space complexity of the algorithm is even more problematic; in games where

1. each terminal node of the game tree has a different situation concept evaluation

2. each intermediary node of the game tree has a different terminal concept evaluation,

each path from the root to a leaf node of the search tree constitutes a local plan with a unique plan signature; therefore, the whole search tree has to be kept in memory, resulting in the same space complexity as time complexity, $O(b_1^{d_1} * \bar{b}_1^{\bar{d}_1} + \cdots + b_n^{d_n} * \bar{b}_n^{\bar{d}_n})$.

Note that condition 1 means that there are no predicates used in the goal description that group fluents from the same subgame together and could be detected as concept terms. Thus, concept decomposition search only pays off when there are "real" concept terms in the game description, resulting in the number of distinct

situation concept evaluations being much smaller than the number of states in the search space.

5.3.2 Solution Quality

As was already shown above, when the algorithm is applicable (i.e., the upper part of the `goal`/`terminal` call tree is ground and non-recursive), an optimal solution is found.

6 Discussion

This chapter will cover a discussion and comparison of the presented methods, along with ideas for future work.

6.1 Subgame Detection Algorithm

Subgames are defined here at the level of fluent and action functions. A much finer granularity could be achieved by propositionalizing all fluents and actions. This would in turn allow a decomposition into more subgames, even for games that the current algorithm cannot decompose. The drawback of that approach, however, is a much higher computational complexity.

A possible compromise would be to partially propositionalize fluent and action functions, e.g. by instantiating only a subset of the parameters.

Another possible optimization would be to regard not only completely independent subgames, but also those cases where only one subgame is non-dependent on the other. If the edges in figure 3.1b on page 15 are interpreted as directed arcs, it can be seen that it would be possible to split the subgame "maze" into two subgames, one of which contains the nodes `move` and `cell`, and the other one containing `grab`, `drop` and `gold`. This would in turn allow the implementation of more sophisticated search algorithms that exploit more symmetries in the game, as is done in partial-order planning.

6.2 Greedy Decomposition Search

Greedy decomposition search, while not optimal, still has its merits. Due to the local evaluation of the goal and terminal predicates (which is also the reason for the non-optimality), any standard tree search algorithm can quickly be adapted for search of subgames. It is easy to include optimizations like e.g. transposition tables (caches). The algorithm also has a good space and time complexity.

It could be worthwhile to develop a procedure that checks if the goal concepts are strictly ordered. This would enable to decide if a given game is in the class of games for which greedy decomposition search is optimal.

6.3 Concept Decomposition Search

The goal of the development of concept decomposition search was to deliver an optimal and correct algorithm for composite games which is applicable to the broadest possible class of games, to create a basis for further optimizations and specializations.

Unfortunately, it turned out that in order to guarantee a correct solution in all cases, it is necessary to include so many local plans in the search that the space complexity of the algorithm makes it unpracticable for some games.

To reduce the complexity, more specialized versions of concept decomposition search should be developed that only apply to certain types of composite games. For example, all games that have appeared in the AAAI competition so far had a terminal description that was a simple disjunction of local terminal concepts. In such a case, the value of the terminal predicate can be checked locally, which eliminates the need for a terminal concept evaluation sequence.

Another drawback of the impossibility to check locally if a partial state is terminal –apart from complexity– is that it requires a search on the space of plans, not on the space of states. This complicates integration of search enhancements that were developed for state space search.

So far, concept decomposition search is only applicable to games with ground goal and terminal concepts. Similar to the subgame detection algorithm, concept term decomposition search could profit from a partial propositionalization of certain predicates, in this case the goal and terminal concepts.

6.4 Future Research Topics

The form of independence in games that was presented here is only one out of many. Examples of other forms of independence, the first three of which have already been featured in the AAAI competition, include

1. *sequential independence* – games where some subgames are played before others;

2. *parallel independence* – games where each action makes a move in all independent subgames;

3. *combinatorial games* – two-player turn-taking zero-sum decomposable games; and

4. *non-combinatorial multi-player games* – decomposable games that have more than two players, feature simultaneous moves or do not have zero-sum awards.

All of these classes of games represent another possible future research topic. Especially solving games from class 3 is very well understood in the context of combinatorial game theory (Conway, 1976), but like with classical games, it is usually assumed that the rules of the game are known in advance. To our knowledge there exists no automatic method of decomposition for combinatorial games.

6.5 Conclusion

In a broader context, the decomposition of logical domain descriptions could be useful for agent control. In the real world, there are often many separate tasks that are only loosely coupled. A deeper understanding of its domain description could enable an agent to reason about the world more efficiently by applying a "divide and conquer" approach.

General game playing is not merely about games. The idea behind it is that computers should be able to adapt dynamically to new circumstances. This is a foundational precondition for truly autonomous agents. In this light, general game playing can be seen as another step toward "true" Artificial Intelligence.

Appendix A Source Code of “incredible”

This is the GDL source code of the game “incredible”. It was created by the Stanford Logic Group and is available from http://games.stanford.edu/.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (role robot)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (init (cell w))
  (init (gold y))

  (init (step c1))

  (init (clear b))
  (init (clear c))
  (init (clear f))

  (init (on c a))
  (init (on e d))
  (init (on f e))

  (init (table a))
  (init (table b))
  (init (table d))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (next (cell ?y))
      (does robot move)
      (true (cell ?x))
      (adjacent ?x ?y))

  (<= (next (cell ?x))
      (does robot grab)
      (true (cell ?x)))

  (<= (next (cell ?x))
      (does robot drop)
      (true (cell ?x)))

  (<= (next (cell ?x))
      (true (cell ?x))
      (does robot (s ?u ?v)))

  (<= (next (cell ?x))
      (true (cell ?x))
      (does robot (u ?u ?v)))

  (<= (next (gold ?x))
      (does robot move)
      (true (gold ?x)))

  (<= (next (gold i))
      (does robot grab)
      (true (cell ?x))
      (true (gold ?x)))

  (<= (next (gold i))
      (does robot grab)
      (true (gold i)))

  (<= (next (gold ?y))
      (does robot grab)
      (true (cell ?x))
      (true (gold ?y))
      (distinct ?x ?y))

  (<= (next (gold ?x))
      (does robot drop)
      (true (cell ?x))
      (true (gold i)))

  (<= (next (gold ?x))
      (does robot drop)
      (true (gold ?x))
      (distinct ?x i))

  (<= (next (gold ?x))
      (true (gold ?x))
```

```
79          (does robot (s ?u ?v)))
80
81      (<= (next (gold ?x))
82          (true (gold ?x))
83          (does robot (u ?u ?v)))
84
85      (<= (next (step ?y))
86          (true (step ?x))
87          (succ ?x ?y))
88
89      (<= (next (on ?x ?y))
90          (does robot (s ?x ?y)))
91
92      (<= (next (on ?x ?y))
93          (does robot (s ?u ?v))
94          (true (on ?x ?y)))
95
96      (<= (next (on ?x ?y))
97          (does robot (u ?u ?v))
98          (true (on ?x ?y))
99          (distinct ?u ?x))
100
101     (<= (next (on ?x ?y))
102         (true (on ?x ?y))
103         (does robot move))
104
105     (<= (next (on ?x ?y))
106         (true (on ?x ?y))
107         (does robot grab))
108
109     (<= (next (on ?x ?y))
110         (true (on ?x ?y))
111         (does robot drop))
112
113     (<= (next (table ?x))
114         (does robot (s ?u ?v))
115         (true (table ?x))
116         (distinct ?u ?x))
117
118     (<= (next (table ?x))
119         (does robot (u ?x ?y)))
120
121     (<= (next (table ?x))
122         (does robot (u ?u ?v))
123         (true (table ?x)))
124
125     (<= (next (table ?x))
126         (true (table ?x))
127         (does robot move))
128
129     (<= (next (table ?x))
130         (true (table ?x))
131         (does robot grab))
132
133     (<= (next (table ?x))
134         (true (table ?x))
135         (does robot drop))
136
137     (<= (next (clear ?y))
138         (does robot (s ?u ?v))
139         (true (clear ?y))
140         (distinct ?v ?y))
141
142     (<= (next (clear ?y))
143         (does robot (u ?x ?y)))
144
145     (<= (next (clear ?x))
146         (does robot (u ?u ?v))
147         (true (clear ?x)))
148
149     (<= (next (clear ?x))
150         (true (clear ?x))
151         (does robot move))
152
153     (<= (next (clear ?x))
154         (true (clear ?x))
155         (does robot grab))
156
157     (<= (next (clear ?x))
158         (true (clear ?x))
159         (does robot drop))
160
161     (<= (next (step ?y))
162         (true (step ?x))
163         (succ ?x ?y))
164
165     (adjacent w x)
166     (adjacent x y)
167     (adjacent y z)
168     (adjacent z w)
169
170     (succ c1 c2)
171     (succ c2 c3)
172     (succ c3 c4)
173     (succ c4 c5)
174     (succ c5 c6)
175     (succ c6 c7)
176     (succ c7 c8)
177     (succ c8 c9)
178     (succ c9 c10)
```

```
  (succ c10 c11)
  (succ c11 c12)
  (succ c12 c13)
  (succ c13 c14)
  (succ c14 c15)
  (succ c15 c16)
  (succ c16 c17)
  (succ c17 c18)
  (succ c18 c19)
  (succ c19 c20)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (legal robot move))

  (<= (legal robot grab)
      (true (cell ?x))
      (true (gold ?x)))

  (<= (legal robot drop)
      (true (gold i)))

  (<= (legal robot (s ?x ?y))
      (true (clear ?x))
      (true (table ?x))
      (true (clear ?y))
      (distinct ?x ?y))

  (<= (legal robot (u ?x ?y))
      (true (clear ?x))
      (true (on ?x ?y)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (completed bdftower)
      (true (on d f))
      (true (on b d)))

  (<= (completed caetower)
      (true (on a e))
      (true (on c a)))

  (<= (goal robot 100)
      (true (gold w))
      (completed bdftower)
      (completed caetower))

  (<= (goal robot 75)
      (true (gold w))
      (not (completed bdftower))
      (completed caetower))

  (<= (goal robot 70)
      (true (gold w))
      (completed bdftower)
      (not (completed caetower)))

  (<= (goal robot 55)
      (completed bdftower)
      (completed caetower)
      (not (true (gold w))))

  (<= (goal robot 45)
      (true (gold w))
      (not (completed bdftower))
      (not (completed caetower)))

  (<= (goal robot 30)
      (completed caetower)
      (not (completed bdftower))
      (not (true (gold w))))

  (<= (goal robot 25)
      (completed bdftower)
      (not (completed caetower))
      (not (true (gold w))))

  (<= (goal robot 0)
      (true (step c20))
      (not (completed caetower))
      (not (completed bdftower))
      (not (true (gold w))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= terminal
      (true (step c20)))

  (<= terminal
      (true (gold w)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Appendix B Dynamic Programming Implementation of CombinePlans

Algorithm B.1 on the next page sketches out how to implement the function CombinePlans using a dynamic programming technique, thereby avoiding the factorial complexity of the naïve approach (refer to section 5.1.4 on page 27).

Two short remarks on notation:

- *initial_sit* denotes the start node of the search algorithm; for sake of brevity, this is not listed among the function arguments
- ValueOf returns the goal value of a situation *sit*, if *sit* is terminal; otherwise, a heuristic value is returned

Algorithm B.1 CombinePlans

1: **global variable** $map\_visited : \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{n \text{ times}} \to \{true, false\}$; initially empty
2: **function** CombinePlans($plan\_set$)
3: **return** CombinePlansRec($plan\_set, [0, 0, \ldots, 0], initial\_sit$)
4: **end function**

5: **function** CombinePlansRec($\{plan_1, \ldots, plan_n\}, pos, sit$)
6: let $pos = [p_1, \ldots, p_n]$
7: let $plan_1 = [action_1^1, action_2^1, \ldots, action_{m_1}^1]$
8: let $plan_2 = [action_1^2, action_2^2, \ldots, action_{m_2}^2]$
9: $\vdots$
10: let $plan_n = [action_1^n, action_2^n, \ldots, action_{m_n}^n]$
11: **if** $map\_visited[pos] = true$ **then**
12: **return** $fail$
13: **else if** $pos = [m_1, m_2, \ldots, m_n]$ **then**
14: **return** $\langle$ValueOf(sit)$, \emptyset\rangle$
15: **end if**
16: **for** $i = 1, \ldots, n$ **do**
17: **if** $p_i < m_i$ **then**
18: $new\_sit \leftarrow Do(action_{p_i+1}^i, sit)$
19: $new\_pos \leftarrow [p_1, \ldots, p_{i-1}, p_i + 1, p_{i+1}, \ldots, p_n]$
20: $res \leftarrow$ CombinePlansRec($\{plan_1, \ldots, plan_n\}$, new_pos, new_sit)
21: **if** $res = \langle value, \_\rangle$ **then**
22: **return** $\langle value, action_{p_i+1}^i\rangle$
23: **end if**
24: **end if**
25: **end for**
26: $map\_visited[pos] \leftarrow true$
27: **return** $fail$
28: **end function**

Bibliography

Conway, John Horton. 1976. *On Numbers and Games*. London Mathematical Society Monographs, no. 6. Academic Press.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. 2001. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. Chap. 22.3: Depth-first search, pages 540–549.

Genesereth, Michael R., Love, Nathaniel, & Pell, Barney. 2005. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, **26**(2), 62–72.

Schiffel, Stephan, & Thielscher, Michael. 2007. Fluxplayer: A Successful General Game Player. *In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*. Menlo Park, California: The AAAI Press.

Thielscher, Michael. 1998. Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, **2**(3–4), 179–192.

Thielscher, Michael. 2005. *Reasoning Robots - The Art and Science of Programming Robotic Agents*. Applied logic series, vol. 33. Dordrecht: Springer.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 16. Juli 2007

Martin Günther