EMCL Master Thesis

Decomposition of Multi-Player Games

Author:
Dengji Zhao
Advisor:
Dipl.-Inf. Stephan Schiffel
Supervisor:
Prof. Dr. Michael Thielscher

International Center
for Computational Logic
Technical University of Dresden

May 25, 2009

Abstract

Research in General Game Playing aims at building a system which can play arbitrary games. Moreover, the system should not only be able to play any game, but also to play the game fast. Time constraints for choosing moves are a main challenge to the system, especially with very large games. However, some large games can be split into many independent small games (called subgames), and the game can be solved by solving its subgames. This is called decomposition search. We will show in our work that the time complexity of decomposition search for decomposable games is exponentially smaller than that of normal game search.

Acknowledgments

I would like to acknowledge and extend my heartfelt gratitude to Michael Thielscher and Stephan Schiffel who were abundantly helpful and offered undivided support and guidance.

Also to Yao Xu for reading this thesis and giving suggestions, and to Martin Günther for sharing his thesis work.

Most especially to my family and friends.

Dengji Zhao

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 Game Description Language 3

2.2 Properties of Multi-Player Games 4

3 Game Decomposition **7**

3.1 Games and Subgames 7

3.2 Subgame Detection 8

3.2.1 Subgame Detection Algorithm 9

3.2.2 Correctness of Subgame Detection 10

3.2.3 Complexity of Subgame Detection 10

3.3 Fluent and Action Instantiation 11

4 Decomposition Search for Impartial Games **13**

4.1 Impartial Property Checking 13

4.2 The Game Nim and Nimber 19

4.3 Game Search 19

4.3.1 Subgame Search 20

4.3.2 Global Game Search 22

4.4 Complexity Comparison 24

4.5 Experimental Results 26

5 Decomposition Search for General Partial Games **27**

5.1 Alternating Move Games 27

5.1.1 Subgame Search 28

5.1.2 Global Game Search 34

5.2 Simultaneous Move Games 43

5.2.1 Subgame Search 43

5.2.2 Global Game Search 44

5.3 Complexity Comparison 44

5.4 Experimental Results 45

6 Special Cases for Parallel and Serial Games **47**

6.1 Parallel Games 47

6.1.1 Special for Subgame Detection 47

6.1.2 Special for Game Search 49

6.2 Serial Games 50

6.2.1 Special for Subgame Detection 50

6.2.2 Special for Game Search 51

6.3 Complexity Comparison 51

7 Conclusion **53**

Appendix A: Double TicTacToe in GDL **57**

Appendix B: Parallel TicTacToe in GDL **60**

Appendix C: Serial TicTacToe in GDL **63**

List of Algorithms **67**

List of Tables **69**

List of Figures **71**

References **73**

Chapter 1

Introduction

A General Game Playing (GGP for short) system is one that can accept a formal description of a game and use such description to play the game effectively without human intervention. Unlike specialized game players, such as Deep Blue, general game players are able to play arbitrary games and do not rely on algorithms designed in advance for specific games. In GGP, a game is described in the Game Description Language which basically represents the finite state graph, called Finite State Machine, of the game, and the player(s) will search the state graph to find winning strategies (sequences of moves) to play. For most games, the state graph is very large, using normal search algorithms to find a winning strategy in a very limited time period is impossible. Although there are many kinds of optimizations and heuristics used to fasten game search in many ways, most of them focus on search algorithms, i.e. they do not change the game tree. We find that many games can be decomposed into subgames which are loosely coupled, and that the state graph of each subgame is exponentially smaller compared to the whole game state graph, and it can be easily searched by using normal search algorithm with some adaptions if necessary (called subgame search). Combining subgame search results, another search, called global game search, will try to find strategies as optimal as the ones we can get by using normal search on the whole state graph. This kind of research has already been started on single player games [Gue07]. Based on their results, we do further research on multi-player games here.

Chapter 2

Preliminaries

2.1 Game Description Language

According to the definition in [MG05], Game Description Language (GDL) is a formal language for defining discrete games with complete information. GDL supports representation by relying on a conceptualization of game states as databases and by relying on logic to define the notions of legality, state update, etc.

A database schema is a set of objects, a set of tables, and a function that assigns a natural number to each table (its arity the number of objects involved in any instance of the relationship). The Herbrand base corresponding to a database schema is defined as the set of expressions of the form r(a_1,...,a_n), where r is a relationship of arity n and a_1,...,a_n are objects. An instance of a database schema, then, is a subset of the corresponding Herbrand base. Each game has an associated database schema, and each state of the game is an instance of this schema. Different states correspond to different instances, and changes in the game correspond to movement among these database instances.

Unfortunately, such explicit representations are not practical in all cases. Even though the numbers of states and actions are finite, they can be extremely large; and the tables relating to them can be larger still. However, with the database model, it is possible to describe these tables in a more compact form by encoding the notions of legality, goalhood, termination, and update as sentences in logic rather than as explicit tables.

GDL uses an indexical approach to defining games. A GDL game description takes the form of a set of logical sentences that must be true in every state of the game. The distinguished vocabulary words that support this are described below:

role($\langle a \rangle$**)** means that $\langle a \rangle$ is a role (player) in the game.

init($\langle p \rangle$**)** means that the datum $\langle p \rangle$ is true in the initial state.

true($\langle p \rangle$**)** means that the datum $\langle p \rangle$ is true in the current state.

does($\langle r \rangle$**,**$\langle a \rangle$**)** means that player $\langle r \rangle$ performs action $\langle a \rangle$ in the current state.

next($\langle p \rangle$**)** means that the datum $\langle p \rangle$ is true in the next state.

legal($\langle r \rangle$**,**$\langle a \rangle$**)** means it is legal for $\langle r \rangle$ to play $\langle a \rangle$ in the current state.

goal($\langle r \rangle$**,**$\langle v \rangle$**)** means that player $\langle r \rangle$ would receive the goal value $\langle v \rangle$ in the current state, should the game terminate in this state.

terminal means that the current state is a terminal state.

distinct($\langle p \rangle$**,**$\langle q \rangle$**)** means that the datums $\langle p \rangle$ and $\langle q \rangle$ are syntactically unequal.

GDL is an open language in which this vocabulary can be extended; however, the significance of these basic vocabulary items is fixed for all games. Game 2.1 shows one possible GDL representation of game Nim with 4 heaps of size 1, 2, 3 and 5 (more information of this game can be found in chapter 4).

2.2 Properties of Multi-Player Games

There are some special properties of multi-payer games. We will first list these properties which will be used in the rest of this paper.

- Alternating Move Games

  An alternating move game is a game in which every time only one player can move, and the turn changes from one player to another in a predefined order in the game. In GDL, every time every player should do a move no matter if it is his turn. Thus, in an alternating move game described in GDL, a player has to do a special move *noop*, which does not effect game state, when it is not his turn.

- Simultaneous Move Games

  Compared to an alternating move game, a simultaneous move game is a game in which all players have to move in each turn.

Game 2.1 Nim

```
(role player1)
(role player2)

(init (heap a 1))
(init (heap b 2))
(init (heap c 3))
(init (heap d 5))
(init (control player1))

(<= (legal ?p noop)
(true (control ?x))
(role ?p)
(distinct ?x ?p))

(<= (legal ?p (reduce ?x ?n))
(true (control ?p))
(true (heap ?x ?m))
(smaller ?n ?m))

(<= (next (heap ?x ?n))
(does ?p (reduce ?x ?n)))

(<= (next (heap ?x ?n))
(true (heap ?x ?n))
(does ?p (reduce ?y ?m))
(distinct ?x ?y))

(<= (next (control ?p2))
(true (control ?p1))
(next_player ?p1 ?p2))

(<= terminal
(true (heap a 0))
(true (heap b 0))
(true (heap c 0))
(true (heap d 0)))

(<= (goal ?p 100)
(true (control ?p)))

(<= (goal ?p 0)
(true (control ?p1))
(next_player ?p ?p1))

(<= (smaller ?x ?y)
(succ ?x ?y))
(<= (smaller ?x ?y)
(succ ?x ?z)
(smaller ?z ?y))

(next_player player1 player2)
(next_player player2 player1)

(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
```

- Impartial and Partial Games

  An impartial game is an alternating move game in which the legal moves depend only on the position (or state) of the game and not on which player is currently moving, and the payoffs are symmetric, e.g. game Nim a two-player impartial game. All games which are not impartial are partial.

- Parallel and Serial Games

  A parallel game is a game which contains more than one independent subgame (subgame definition will be given later, two subgames are independent iff one move in one subgame does not effect the other, but they might be both related to terminal and goal conditions). A player in a parallel game has to move in all the subgames when it is his turn to move. A serial game is a game which also contains more than one independent subgame, but they are played one after another, i.e the subgames are ordered, one subgame can not be started

until all subgames before it are already finished.

Chapter 3

Game Decomposition

Before we can use decomposition search for a game, we have to find out that the game has subgames. This chapter will first define what are game and subgame and then give the algorithm of subgame detection.

3.1 Games and Subgames

Before going to talk about subgame detection, we need first define what we mean by *game* and *subgame* here.

Definition 3.1. *(Term). A term is*

- *A variable*
- *An object constant*
- *A function constant of arity n applied to n terms, e.g. f(a, X, g(h(c, Y), e))*

Definition 3.2. *(Fluent). A* ***fluent*** *of a game is a* term*, which is not a* variable*.*

Definition 3.3. *(Action). An* ***action*** *of a game is a* term*, which is not a* variable*.*

Definition 3.4. *(Role). A* ***role*** *of a game is a player of the game.*

Definition 3.5. *(Game). A* ***game*** *is a tuple G =* ***(F,A,I,R)*** *where*

- ***F*** *is a set of fluents,*
- ***A*** *is a set of actions,*

- ***I*** *is the initial state of the game, which is a set of ground instances of F,*
- ***R*** *is a set of roles.*

Definition 3.6. *(State). A* ***state*** *S of a game G = (F,A,I,R) is a set of ground instances of F, and S can be reached from initial state by playing G.*

Ground instances of the fluents of a game can be many, but not all of them are reachable states of the game.

Definition 3.7. *(Subgame). A game* $G = (F, A, I, R)$ *is a* ***subgame*** *of* $G' = (F', A', I', R')$ *iff* $F \subseteq F'$, $A \subseteq A'$, $I \subseteq I'$, $R \subseteq R'$, *and F, A, I and R are not empty.*

Game G is a subgame of itself.

Definition 3.8. *(Subgame Independence). Two subgames Gs = (Fs,As,Is,Rs) and* $Gs' = (Fs', As', Is', Rs')$ *of game G are* ***independent*** *each other iff* $Fs \cap Fs' = \oslash$ *and* $As \cap As' = \oslash$.

In the rest of this paper, when we talk about subgames of a game, we mean the subgames are independent each other.

Definition 3.9. *(Action Independent Fluent). A fluent is* ***action independent*** *iff all actions of the game have the same effect on this fluent, i.e. the fluent's change only depends on if the game is moving from one state to another, and not on what the moves are and who is playing.*

Example 3.1. The very simple example for action independent fluent in most single player games would be step counter, and in alternating move multi-player games, there is a control fluent, which does not depend on any action.

Definition 3.10. *(Fluent Independent Action). An action is* ***fluent independent*** *iff the action does not change any fluent's state except for action independent fluents.*

3.2 Subgame Detection

In this section, we will present the subgame detection algorithm which can detect independent subgames for multi-player games. If we can find more than one independent subgame of a game, then we say this game is *decomposable*, and we can use special search methods for subgames to improve the whole game search, mainly, by reducing state space and search time. Depending on the property of the game, later sections will give detailed search methods for different classes of games. In the rest of this section, we first give the subgame detection algorithm, then give a very short proof of the algorithm's correctness, and the complexity of this algorithm will be analyzed in the end.

3.2.1 Subgame Detection Algorithm

The algorithm is similar to single player game subgame detection in [Gue07], the main extension is that, instead of fluent and action functions (only consider their name and arity), we also consider the arguments (can be constant or variable) of them. The main idea is, given all the dependency relations, namely preconditions, positive and negative effects, between fluents and actions, we first build the dependency graph of all these relations, then we try to find all connected components in this dependency graph. Finally, each component will contain fluents and actions of one subgame and all subgames are independent each other. Please note that we do not consider how to compute all the dependency relations between fluents and actions, which are computed additionally. Algorithm 3.1 shows the basic idea of this subgame detection.

Algorithm 3.1: SubgameDetection

Input: DRs: Dependency relations between fluent and action
Output: SGs: Subgames
Data: DG: Dependency graph, CCs: Connected components
begin
  `/* Dependency graph construction */`
  foreach *dependency relation* $DR \in DRs$ **do**
    if *the fluent in DR is not control* **then**
      Add an edge from the fluent to the action of DR in DG;
    end
  end
  `/* Search connected components in dependency graph */`
  foreach *fluent* $F \in DG$ **do**
    Build a connected component with its directly connected actions in CCs;
  end
  for *any two different components* CC_1 *and* $CC_2 \in CCs$ **do**
    if $CC_1 \cap CC_2 \neq \oslash$ **then**
      $CC_3 \leftarrow$ merge CC_1 and CC_2;
      $CCs \leftarrow CCs \backslash \{CC_1, CC_2\}$;
      $CCs \leftarrow CCs \cup CC_3$;
    end
  end
  Repeat previous **for** loop until there is no subgame that can be merged;
  $SGs \leftarrow CCs$;
end

This detection algorithm only considers fluents and actions that have at least one one dependency relation. There might be some *action independent fluents* and *fluent independent actions* which are not included in the dependency relations. For action independent fluents, we will build a special subgame, called *action independent subgame*, containing all action independent fluents without any action. Compared to action independent subgame, collecting all fluent independent actions, we get another subgame called *fluent independent subgame* without fluent. Another exception in algorithm 3.1 is the control fluent, which is a special fluent in alternating move games and a precondition of all actions. It will connect all possible subgames together because of its precondition relation with all actions. However, we know that control fluent is action independent, thus, in order to get optimal subgames, we should ignore any relation with control fluent, if there is one, during dependency graph construction.

3.2.2 Correctness of Subgame Detection

The only thing we need to check is that all subgames the detection algorithm detected are independent and smallest. This is obvious, as if two subgames are not independent, i.e. there is at least one shared fluent (except control fluent) or action, then they must be connected through this fluent or action, which will be detected by this algorithm during finding connected components. Moreover, if one subgame is not smallest, that means it can be decomposed further, this can not happend in this algorithm, because if there is no relation between fluents and actions in two independent subgames, then this subgame detection algorithm will not connect them together. So given dependency relations between fluents and actions, the subgame detection algorithm will give all smallest independent subgames with respect to these given dependency relations.

3.2.3 Complexity of Subgame Detection

The time complexity depends on the number of dependency relations between fluents and actions, and the number of fluents and actions involved in dependency relations. Finding connected components can be implemented by depth-first or breadth-first search, in either case, the time complexity is $O(\mid E \mid + \mid V \mid)$ where E is the number of dependency relations and V is the number of fluents and actions involved in dependency relations. Space complexity is also $O(\mid E \mid + \mid V \mid)$.

3.3 Fluent and Action Instantiation

The main reason we extend the subgame detection algorithm for single player games in [Gue07] is that, for some games, if we only consider fluent and action names, the decomposition will not be optimal and even the game will not be decomposable. For example, in the GDL rules of game Nim which are given in 2.1, there are two fluent functions, namely heap/2 and control/1. The initial state of the game is defined by the following *init* rules:

```
(init (heap a 1))
(init (heap b 2))
(init (heap c 3))
(init (heap d 5))
(init (control player1))
```

The initial state has four heaps, a, b, c and d with size 1, 2, 3 and 5 respectively. All heaps are independent and we can solve each heap separately to solve the whole game. To analyze each heap will be much easier than to analyze all heaps together, especially if the heap size is very big. If we only use fluent names to detect subgames, we will not get smallest subgames (heaps) of it. Because all subgames use the same fluent *heap* just with two different arguments: the first one indicates the heap name and the second one gives the size of the heap. In order to get optimal decomposition, we have to instantiate some arguments of fluents and actions.

We don't want to instantiate all arguments of fluents and actions, because instantiate all arguments will get a lot of fluents and actions and a large number of dependency relations between them as well, which is not efficient. The only arguments we should instantiate are the ones which do not change during the whole game playing. In the above example, the fluent *heap* has two arguments, of course, all the arguments are instantiated in initial state. However, if we only instantiate the first argument of *heap*, we will get an optimal decomposition where each heap is a subgame.

Actually, the first argument of heap will never change during the whole game, but the second one changes depending on what the move is. How can we find out if one argument of a fluent will not change in future states? We can easily find this out by analyzing initial fluents and next rules. Group all initial fluents (only fluents which have at least one argument) by the fluent's name and arity. For each group, we check its fluent's arguments one by one to find out whether or not a argument changes in the future. In each next rule which is defined for this fluent, an variable argument of the fluent in the rule head occurs either in a *does* predicate, i.e. it comes from a move, or in another predicate except for *does*. If it occurs in a move, we need to expand the move's legal rules until we find out that the argument occurs

in the same argument position of the same fluent at least in one disjunctive branch of each legal rule; if it occurs in another predicate which is not *does*, we will check each definition (rule) of this predicate until we find out the argument occurs in the same argument position of the same fluent at least in one disjunctive branch. If in all next rules of a fluent, we can find out that one argument of the fluent finally comes from the same position of the same fluent, then we say this argument does not change during the whole game and we can instantiate it, otherwise we can not instantiate it.

Example 3.2. In the GDL definition of Nim in 2.1, we get the following next rules for fluent *heap*. For example, we want to check the first argument of *heap*. In the first *next* rule, the argument comes from a move *reduce* (the first argument of it). The third rule is the legal definition for move *reduce* and we can see the argument is also from the first argument of *heap*, so the first rule satisfies unchanging condition. The second next rule for *heap* is very obvious, as the argument is directly from the same fluent in the body. So the first argument of *heap* can be instantiated. Use the same way to check the second argument of *heap*, we will get that the argument changes.

```
(<= (next (heap ?x ?n))
    (does ?p (reduce ?x ?n)))

(<= (next (heap ?x ?n))
    (true (heap ?x ?n))
    (does ?p (reduce ?y ?m))
    (distinct ?x ?y))

(<= (legal ?p (reduce ?x ?n))
    (true (control ?p))
    (true (heap ?x ?m))
    (smaller ?n ?m))
```

Chapter 4

Decomposition Search for Impartial Games

4.1 Impartial Property Checking

We have mentioned what is impartial game in chapter 2. A more formal definition of it is the following:

Definition 4.1. *(Impartial Game). A game G is* ***impartial*** *iff G is an alternating move game and in each state S of G, every player (if the player is in control) has the same legal moves and each move, no matter who is playing, has the same effects.*

Definition 4.2. *(Impartial State). A state (or position) of a game is impartial iff legal moves for each player (if he is in control) in the state are the same and the effects of each move are also the same, no matter who is playing the move.*

Theorem 4.1. *Game G is impartial iff its independent subgames are impartial.*

We can easily prove above theorem from two sides in the following way. Assume that a game G can be decomposed into N ($N > 1$) independent subgames: G_1,...,G_n,

1. If G is impartial, then all its independent subgames are impartial.

   Proof. Suppose that there is a subgame G_m is not impartial and in a state S_m of G_m, the legal moves are not the same for all players. As the whole game state is just the combination of all subgame states, all the states of G which contain state S_m are not impartial, thus game G is not impartial. This is a contradiction. □

2. If all independent subgames of game G are impartial, then game G is impartial.

   Proof. Suppose that G is partial, i.e., at least one state S_g of G where not all the players have the same legal moves. We know that all the legal moves of state S_g come from all the subgame states, so there is a least one subgame state, say S_m in subgame G_m, which gives different legal moves to different players. Thus subgame G_m is partial, which is a contradiction of the above assumption. □

In order to check whether or not a game is impartial, it seems that we need to check every state of the game. However we don't need to, as GDL was defined to describe games in a compact format, so we are able to use this language's property in impartial property checking. Before going to the details of impartial property checking, we need few more definitions.

Definition 4.3. *L is a set of rules, P is a set of players, and A is a set of ground instances of actions of a game G, the* ***legality*** *of G is* ***defined equivalently*** *for every player in P iff*

$$\forall_{p_1,p_2 \in P} \forall_{a \in A} (L \cup S \cup \{true(control(p_1))\} \models legal(p_1, a) \Leftrightarrow L \cup S \cup \{true(control(p_2))\} \models legal(p_2, a)),$$

where S is an arbitrary set of ground instances of ***true****(X) where X is any fluent of G except for the control fluent.*

Definition 4.4. *L is a set of rules, P is a set of players, and F is a set of ground instances of fluents (except for the control fluent) of a game G, the* ***effects of all ground actions*** *A of G are* ***defined equivalently*** *for every player in P iff*

$$\forall_{p_1,p_2 \in P} \forall_{f \in F} \forall_{a \in A} (L \cup S \cup \{true(control(p_1)), does(p_1, a)\} \models next(f) \Leftrightarrow L \cup S \cup \{true(control(p_2)), does(p_2, a)\} \models next(f)),$$

where S is an arbitrary set of ground instances of ***true****(X) where X is any fluent (except for the control fluent) of G.*

Corollary 4.1. *A game is impartial iff the legality and the effects of all ground actions are both defined equivalently for each player.*

Proof. If legality of a game is defined the same, then we have, in any state of the game, that every player will get the same legal moves if he is in control. If the

effects of all ground actions are defined equivalently, then we have, in any state of the game, that every move has the same effects no matter which player is playing that move. These exactly fulfill the conditions of impartial game definition. □

Definition 4.5. *(Correspondency). Given two rules (or predicates) R_1 and R_2 of a multi-player game G, R_1 and R_2 are* ***correspondent*** *for players P_1 and P_2 iff simultaneous substituting P_1 for P_2 and P_2 for P_1 in R_1 will get R_2, i.e. $R_1[P_1/P_2, P_2/P_1] = R_2$.*

Lemma 4.1. *Two legal (or next) rules are correspondent for players P_1 and P_2 iff these two rules give the same effect (e.g. the same legal move or the same effect on fluents) for each player when the player is under the same condition, e.g. the player is (or not) in control or doing the same move.*

Example 4.1. Of the following eight rules, rules 1 and 2 are correspondent for *player1* and *player2*, as they give the same legal move for the player when he is in control. Rules 3 and 4 are also correspondent for *player1* and *player2*, because they give the same next fluent when two players are doing the same move. Rules 5 and 6 are also correspondent for *player1* and *player2* as one player get a legal move *noop* when the other is in control. However rules 7 and 8 are not correspondent for any two players.

```
1. (<= (legal player1 move)
       (true (control player1)))
2. (<= (legal player1 move)
       (true (control player2)))
3. (<= (next f)
       (does player1 move))
4. (<= (next f)
       (does player2 move))
5. (<= (legal player1 noop)
       (true (control player2)))
6. (<= (legal player2 noop)
       (true (control player1)))
7. (<= (legal player1 noop)
       (true (control player2)))
8. (<= (legal player2 noop)
       (true (control player3)))
```

Algorithms 4.1 to 4.5 give one possible way to do the above impartial checking, which is not complete, but is soundness. One precondition of the algorithm is that the game going to be checked must be an alternating move game.

Algorithm 4.1: ImpartialCheck

```
Input: GRs: Game Rules
Output: Impartial: True/False
begin
    if LegalImpartialCheck and NextImpartialCheck then
        Return True
    else
        Return False
    end
end
```

Algorithm 4.2: LegalImpartialCheck

```
Input: GRs: Game Rules
Output: LegalImpartial: True/False
begin
    VarLegals ← all legal rules containing variable player argument in their
    heads;
    foreach VarL ∈ VarLegals do
        VarPlayer ← the variable player argument of VarL;
        SubPreds ← all predicates (except control) containing VarPlayer
        from the body of VarL;
        foreach SubPred ∈ SubPreds do
            if PredCheck(SubPred) fail then Return False;
        end
    end
    ConstLegals ← all legal rules containing constant player argument in
    their heads;
    if CorrespondencyCheck(ConstLegals) fail then Return False;
    Return True;
end
```

Algorithm 4.3: NextImpartialCheck

Input: GRs: Game Rules
Output: NextImpartial: True/False
begin
  $VarNexts \leftarrow$ all next rules containing does/2 predicate with variable player argument;
  foreach $VarN \in VarNexts$ **do**
    $VarPlayer \leftarrow$ the variable player argument of VarN;
    $SubPreds \leftarrow$ all predicates (except true(control(X))) containing VarPlayer from the body of VarN;
    foreach $SubPred \in SubPreds$ **do**
      if *PredCheck(SubPred) fail* **then** Return False;
    end
  end
  $ConstNexts \leftarrow$ all next rules containing does/2 predicate with constant player argument;
  if *CorrespondencyCheck(ConstNexts) fail* **then** Return False;
  Return True;
end

Algorithm 4.4: CorrespondencyCheck

Input: RuleList: a list of rules
Output: Correspondent: True/False
begin
  $RuleGroups \leftarrow$ group RuleList by player;
  if *the lengths of all the groups in RuleGroups are not equal* **then**
    Return False
  end
  $Group_1 \in RuleGroups$;
  $RestGroups \leftarrow RuleGroups \backslash \{Group_1\}$;
  foreach $Rule_1 \in Group_1$ **do**
    foreach $Group \in RestGroups$ **do**
      if *there is no* $Rule_2 \in Group$ *correspondent to* $Rule_1$ **then**
        Return False
      end
    end
  end
  Return True;
end

Algorithm 4.5: PredCheck

```
Input: Pred: Predicate
Output: PredImpartial: True/False
begin
    AllRules ← all rules with head unified with Pred;
    ConstRules ← all rules, not containing variable player argument, from
    AllRules;
    VarRules ← all rules, still containing variable player argument, from
    AllRules;
    if CorrespondencyCheck(ConstRules) fail then Return False;
    foreach Rule ∈ VarRules do
        SubPreds ← all predicates (except control) containing variable player
        argument in the body of Rule;
        foreach SubPred ∈ SubPreds, which is not checked so far do
            if PredCheck(SubPred) fail then Return False;
        end
    end
    Return True;
end
```

4.2 The Game Nim and Nimber

"Nim is a two-player mathematical game of strategy in which players take turns removing objects from distinct heaps. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap. Nim is a typical impartial game and is usually played as a *misère game* in which the player who takes the last object loses. Nim can also be played as a *normal play game* in which the player who takes the last object wins. Nim has been mathematically solved for any number of heaps and objects. The key of the theory of the game is the binary digital sum, neglecting all carries, of the heap sizes, which is also known as exclusive or (xor) or nim-sum in combinatorial game theory. The nim-sum of x and y is written x $\oplus$ y to distinguish it from the ordinary sum, x + y.

"Normal play Nim is fundamental to the Sprague-Grundy theorem[1], which essentially says that in normal play every finite impartial game is equivalent to a Nim heap that yields the same outcome when played in parallel with this impartial game."[2] The same theorem can be found in [Con76].

Definition 4.6. *A nimber is a special game denoted by *n for some integer n and $n \geq 0$. We define *0 = {}, and *(n+1) = *n $\cup$ {*n}.*

Nimber *n corresponds to the size of a heap in game Nim that the name comes from. Two nimbers G and H can be added to make a new game G+H in which a player can choose either to move in G or in H. This actually gives the idea of nimber addition.

Definition 4.7. *Given two nimbers G and H, nim addition* $G \oplus H = \{G \oplus h | h \in H\} \cup \{g \oplus H | g \in G\}$.

It is easy to prove that nim addition is commutative and associative. Two games G and G′ are equivalent if for every game H, the game G$\oplus$H is in the same outcome class as G′$\oplus$H. We write G $\approx$ G′. More information of Nim can be found in [Con76].

4.3 Game Search

After subgame decomposition of a impartial game, we get many impartial subgames. We know that all the subgames are equal to some Nim heaps. If we could find out

[1] `http://en.wikipedia.org/w/index.php?title=Sprague-Grundy_theorem&oldid=281887578`, 5 April 2009

[2] `http://en.wikipedia.org/w/index.php?title=Nim&oldid=280608254`, 30 March 2009

all the sizes of heaps (or subgames), then we can very easily compute the best next move by nim addition. So the idea of subgame search is to find the nimber (size) of each subgame, and the global game search will compute the best next move with all subgame nimbers according to different winning conditions.

4.3.1 Subgame Search

As subgames of impartial game are impartial, we don't need to consider all players during subgame search. We will use depth-first search to search each subgame with only one player to compute the nimber of subgame states. We can make sure the nimber we get from subgame search is correct if the search is finished with all leaf nodes are terminal, i.e. no legal moves. Unfortunately, we can not guarantee that we can always completely finish the search with limited time. Thus we need heuristics to evaluate unexpanded leaf nodes, call it unfinished leaf state. One heuristic we used in the following subgame search is mobility, i.e. the number legal moves in unfinished leaf state. It is reasonable to use the number of legal moves as nimber of unfinished leaf state, because in most cases the number of legal moves of a state is equal to the nimber value of the state. One counter example in figure 4.1, it is a two player impartial game (called Hackenbush) with three lines directly or indirectly connect to the bottom dashed line (called ground). In each turn, the player in control cut any line and, afterwards, all lines which do not connect to the ground will also be removed. In this example, the first player has three choices, either moving to *2 by cutting one of the two lines in the top, or moving to *0 by cutting the line directly connecting to the ground. So this impartial game is equal to a nim heap of size 1, but there are three legal moves for the first player.

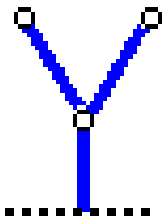

Figure 4.1: Hackenbush

Algorithm 4.6 gives the basic idea of impartial subgame search. Given a subgame state, the subgame name, the player which will be consider for the search, current search depth (starts with 0) and maximum search depth, the algorithm will return the nimber value of the input state, a list of legal move-value pairs of the input

state and a boolean value indicating whether or not all leaf states are terminal. It terminates when all leaf states are terminal or the maximum depth is reached.

Algorithm 4.6: SubgameSearchImpartial

```
Input: State: subgame state, Player, CurrentDepth: current search depth,
       MaxDepth: max search depth
Output: StateValue: evaluation value(Nimber) of State,
        SuccessorValueMoveList: a list of value-move pairs of the state,
        AllTerminal: True/False
begin
    if Cached value of State is good enough then
        Return cached value for this state
    else
        LegalMoves ← all legal moves for Player in State;
        if LegalMoves == ∅ then
            StateValue ← 0; Terminal ←True;
        else if CurrentDepth ≥ MaxDepth then
            StateValue ← length of LegalMoves; Terminal ← False;
        else
            SuccessorValueMoveList ← ∅;
            AllTerminal ←True;
            foreach M ∈ LegalMoves do
                Successor ← update(State,M);
                NewDepth ← CurrentDepth + 1;
                (SuccessorValue,AllTerminal1)←SubgameSearchImpartial(Successor,
                NewDepth,MaxDepth);
                add (M,SuccessorValue) to SuccessorValueMoveList;
                if AllTerminal1 == False then AllTerminal ← False
            end
            Compute StateValue from SuccessorValueMoveList;
        end
        Cache StateValue for State;
    end
end
```

4.3.2 Global Game Search

We are able to compute the nimber of each subgame in previous section. In this section, we will see how to use subgame nimbers to compute global best move.

As we mentioned in the introduction of Nim, there are two winning conditions for impartial games. Thus we have the following two strategies to play a impartial game (related work on this topic see [Con76]):

Normal Play Game (i.e. the player to make the last move wins)

The winning strategy is to finish every move with Nim-sum zero of all heaps (subgames), which is always possible if the Nim-sum is not zero before the move. If the Nim-sum is zero, i.e. the player who moves next will lose provided that the other player does not make a mistake, then we can randomly choose a legal move (or choose a move which makes the game very hard to analyze for the next player).

Misère Game (i.e. the player to make the last move loses)

The winning strategy is to finish every move with a Nim-sum of one instead of zero for normal play games. We first calculate the Nim-sum of all nimbers (subgames), and nim-add nimber 1 to this Nim-sum to get another nimber, say X. The winning strategy is to play the subgame, whose size decreases after adding X, by reducing that heap to the Nim-sum of its size and X. If we can not find a such subgame, we can simply choose a random legal move to play.

Algorithms 4.7 and 4.8 show the idea of the above global game search. It first invokes subgame search to compute all subgame nimbers. Then, depending on the winning condition of the game, it calculates the best move for the input player with above mentioned strategies. The main algorithm is iterative deepening.

Algorithm 4.7: GlobalGameSearchImpartial

```
Input: State: global game state, Player, Depth: current search depth,
       MaxDepth: max search depth
Output: BestMove
begin
    AllSubgames ←all action dependent subgames;
    SubgameValueList ← ∅;
    AllTerminal ←True;
    foreach Subgame∈AllSubgames do
        SubState ←subgame state for Subgame in State;
        (Value1,VMList,AllTerminal1)←SubgameSearchImpartial(SubState,
        Player,0,Depth);
        Add (Value1,VMList) to SubgameValueList;
        if AllTerminal1 == False then AllTerminal ← False
    end
    CurBestMove← CalculateBestMove(SubgameValueList);
    if AllTerminal == True then
        BestMove ← CurBestMove;
    else
        Store CurBestMove;
        NewDepth ← Depth + 1;
        BestMove←GlobalGameSearchImpartial(State,Player,NewDepth,MaxDepth);
    end
end
```

Algorithm 4.8: CalculateBestMove

```
Input: ValueList: a list of subgame state value
Output: BestMove
begin
    if normal play game then
        NimSum ← nim-add all values in ValueList;
    else
        NimSum ← nim-add all values in ValueList and 1;
    end
    if NimSum > 0 then
        Find a value V ∈ ValueList such that V⊕NimSum < V;
        Choose a move in subgame with value V where the move leads to a
        subgame state with value V⊕NimSum;
    else
        /* the player will lose if the opponent plays very well  */
        Randomly choose a move from any subgame;
    end
end
```

4.4 Complexity Comparison

The time complexity of impartial subgame search is the time complexity of normal DFS, i.e. $O(\mid V \mid + \mid E \mid)$ where V and E are the number of states and edges of subgame respectively; the space complexity is the largest path of the game tree. In general, for a heap of size n, memoryless subgame search will search 2^n states (including revisited) with $2^n - 1$ edges. How can we get this result? For a heap of size n (subgame state indicated by [n]), it has n moves going to states [n-1], [n-2], ..., [0]. Recursively state [n-1] has n-1 moves to states [n-2], ..., [0]. Finally we get the whole number of states (including revisited) S:

$$\begin{aligned} S &= [n] + [n-1] + 2 * [n-2] + 2^2 * [n-3] + ... + 2^{n-2} * [1] + 2^{n-1} * [0] \\ &= 1 + 2^0 + 2^1 + ... + 2^{n-1} \\ &= 2^n \end{aligned}$$

and the number of edges is the number of states minus initial state which is $2^n - 1$. The average branching factor is about 2. So in the worst case, the time complexity of subgame game search is $O(2^n)$. But actually we do cache the value of searched states, even though we can not guarantee that each cached value will never be deleted. Assumed that the cached values will not be removed during the search,

then the best time complexity is $O((N+1)+\frac{n*(n+1)}{2})$ (i.e. $O(n^2)$) where (N+1) is the number of states and $\frac{n*(n+1)}{2}$ is the number of edges. The time complexity of subgame search is sensitive to the order of expanding moves in each state, e.g. for a state [n], there are n moves leading to states [n-1],[n-2],...,[0], if subgame search always chooses the smallest successor state to search first, where state [m]>[n] iff m>n, then all subgame searches will finish at depth 2; but if subgame search always chooses the biggest successor state search first (i.e. [n] in this example), then the subgame search will finish at depth d, where

- if n is odd, $d=\frac{n+1}{2}+1$;
- if n is even, $d=\frac{n}{2}+1$.

So the time complexity for this special case would be $O(2^d)$, which is still much smaller than $O(2^n)$.

Global game search is iterative deepening DFS (IDDFS), the time complexity for global game search is the same as the one for subgame search of the biggest subgame, say its size is m, which is $O(2^m)$ in worst case and $O(m^2)$ in best case. The memory used for cache is linear to the sum of sizes of heaps.

If we use normal iterative deepening DFS to search the whole game tree without decomposition. Suppose an impartial game has n heaps (subgames) of sizes n_1, n_2, ..., n_n. Then the number of states of the whole game state graph will be S = n_1*n_2*...*n_n. If we also want to cache their values during the search, it will need much more memory than decomposition search. In the worst case, the time complexity of normal search is $O(b^N)$, N=n_1 + n_2, ..., + n_n and b is the average branching factor. The best case will get at depth n+1 when, for each state, the smallest move expanded first and the biggest goes last. Thus the best time complexity will be $O(b^{n+1})$ n is the number of heaps and b is the average branching factor. The average branching factor is hard to compute here, as it depends on how many heaps and what are the sizes of them. But in general, the average branching factor of the whole game would be not less than the average branching factor for a single heap. As for single heap of size n, there are n moves going to n different states, but for global game state with n as a normal sum of sizes of all heaps will have greater or equal than n moves. For example, a game has two heaps with size 2 for each, so the sum of their sizes is 4 and there are 4 different legal moves going to four different states: [2,1],[2,0],[1,2] and [0,2] with 3, 2, 3 and 2 legal moves respectively. But for a single heap of size 4, it has also 4 legal moves going to four different states: [3],[2],[1] and [0] but with 3, 2, 1 and 0 legal moves respectively.

To sum up, in most cases, the complexity of decomposition search is exponentially smaller than standard search without decomposition. Another advantage of decomposition search is that we can use nimbers of subgames to easily find out a player's best move for each state.

4.5 Experimental Results

We have implemented and tested impartial decomposition search for game Nim with 4 heaps (the GDL definition of it can be found in chapter 2). The following table shows the time cost (in second) of normal search and decomposition search to find the first optimal best move with different heap sizes. The first three games are normal play games and the last one is misère game.

Time Cost(s)	Normal Play			Misère
	Heaps Size			
	1,5,4,2	2,2,10,10	11,12,15,25	12,12,20,20
Normal Search	0.4	3.5	6607	10797
Decomposition Search	0.01	0.01	0.07	0.06

Table 4.1: Search Results Comparison for Nim with 4 Heaps

Chapter 5

Decomposition Search for General Partial Games

In previous impartial decomposition game search, we can independently search and evaluate each subgame, then use subgame evaluations to compute global game strategy. In general partial games (except for parallel or serial games which will be discussed later), we can, of course, search subgames independently, but the problem is the evaluation of subgame states, and what information the subgame search should bring back in order to let global game search find optimal global strategy? In this chapter we propose two game search methods. One is that subgame search brings back the best goal evaluation (actually it is goal concept term evaluation) for each player, and global game search will use these subgame evaluations to simply choose the most urgent subgame to move next. The other is that subgame search brings back turn-move sequences (will be defined later) of each subgame, and global game search will use these sequences to do some additional search to find the global best move. We will see that the first method is very fast, as subgame state evaluation and global game search are both very simple, but it might not be able to find optimal global strategy, whereas the second one takes more time to do both local and global game search, but it is able to find optimal global strategy as we get by using normal search (e.g. minimax) on the whole game.

5.1 Alternating Move Games

In alternating move games, every time only one player can move (In GGP, other players who is not in control also move, but the moves doesn't effect game state). If an alternating move game is decomposable, i.e. it has at least two independent

subgames except action-independent subgame, only one subgame, in each turn, can be played by the player who has control.

5.1.1 Subgame Search

In general search of alternating move games, in each state we know whose turn it is, so we can use pruning techniques, like minimax with alpha-beta pruning. But subgames of alternating move games are not alternative anymore, i.e. the turn in subgame state is nondeterministic. As we don't know who is playing next in subgame search, we will consider every player in expanding of each subgame state, i.e. we need to search all legal moves of each player if he is in control in the state. The main challenge is not the search method but the evaluation of subgame states. The same challenge occurs in single player game decomposition [Gue07], where they introduced the local *goal* (or *terminal*) concept of *goal* (or *terminal*) predicates to help solving the evaluation problem. We will redefine this concept definition with a little change as we do not use fluent function here. The new definition is the following:

Definition 5.1. *A **local goal (resp. terminal) concept** (local concept for short) is a ground predicate call that occurs in the body of the goal (resp. terminal) predicate's definition. The expanded definition of a local goal (resp. terminal) concept must only contain instances of fluents from exactly one subgame. Additional all concepts from one goal or terminal rule must be biggest, i.e. no concept can be dominated by another.*

Using local concept, we will see how these two subgame search methods which we mentioned in the beginning of this chapter work. The main search algorithm is the same in these two methods: for each state, it expands all legal moves of all players. The only difference is the state evaluation.

Method I

We can evaluate subgame state in terms of local concept: one local concept is either satisfied or not in a subgame state. If a local terminal concept term is satisfied in a subgame state, then the whole game (and the subgame) might be terminal depending on what is the relation between this local terminal concept term and its corresponding terminal predicate. If this terminal concept term is the only concept term of a terminal predicate, then the whole game will be terminal and so is the subgame; otherwise, the whole game termination depends also on other local terminal concept terms in other subgame(s), so in this subgame we need to do

further expanding. Thus the subgame state is terminal iff the whole game terminal is satisfied or there is no legal move for all players.

For each terminal state of a subgame, we can evaluate the state by local goal concepts of the subgame. Normally, there are many goal rules defined for different player with different goal value. If we evaluate all local concept terms of all goal rules, we might get many useless local goal concepts which make evaluation comparison very hard. Thus we will only consider goal rules defined for maximum goal value (e.g. 100) of each player, which is reasonable, because every player wants to win the game. Maximum goal value rules can be defined differently from player to player, so we need to record a set of maximum local goal concepts for each player. Please note that the maximum local goal concepts for one player in a subgame can be more than one, this happens, in most cases, when the goal definitions are nonmonotonic, i.e. one player has at least two winning conditions and they are not consistent, i.e. they can not happen at the same time. Here we assume goal definitions are monotonic, i.e. for each player in each subgame, there is at most one maximum local goal concept. Moreover, in order to make the evaluations of local concepts are comparable, we need to know whether a local concept is positive or negative (called it sign of a local concept) in its rule, which can be found out during local concept construction.

After we have all local maximum goal concepts, we can evaluate subgame terminal states by these concepts of each player. If a maximum local goal concept is satisfied (consider its sign) in a subgame state, we can either give a maximum goal value or some other values (e.g. True or False) indicating the satisfaction for the player. Of course, evaluation result format depends on how we want to use them later. In our first method, in each terminal state, we assign goal value (either maximum if the term is satisfied or zero when it is not) to each player who the local goal concept belongs to. Thus the value in a terminal state will be a list of values (one for each player), e.g. [V_{p1},V_{p2}] for a terminal state of a two-player subgame.

For nonterminal state, for each player who has legal move in the state, we choose one biggest value of all one step successors reached by his moves, e.g. $[[V^1_{p1}, V^1_{p2}], [V^2_{p1}, V^2_{p2}]]$ with two state values (one best for each player) in a nonterminal state of a two-player game. Then from all the best values of all players, we select again the biggest value of each player to build up the state value of current state, e.g. $[V^1_{p1}, V^2_{p2}]$ from the above example. So the value of each state tells us that each player has chance to get the value specified for him in the state value and also tells us who (might be not himself) has to move next in order to get the value for him. For the state which is not expanded, we can either use the same evaluation as for terminal states or use some heuristic function to get the value of the state. Algorithm 5.1 summarizes the above idea.

Algorithm 5.1: SubgameSearchIForAlternatingMove

```
Input: State: subgame state, CurrentDepth: current search depth,
       MaxDepth: max search depth
Output: StateValue: a list of best value for each player, BestMoveValueList:
        a list of bestmove-value pair for each player, AllTerminal:
        True/False
begin
    BestMoveValueList ← ⊘;
    AllTerminal ← True;
    Players ← all players;
    foreach P∈Players do
        MyMoveValueList ← ⊘;
        LegalMoves ← all legal moves of P in State;
        foreach M∈LegalMoves do
            NextState ← update(State,M);
            NextDepth ← CurrentDepth+1;
            if NextDepth ≥ MaxDepth then
                NextStateValue ← Heuristic(NextState);
                AllTerminal ← False;
            else
                (NextStateValue,AllTerminal1) ← SubgameSearchIForAlter-
                natingMove(NextState,NextDepth,MaxDepth);
                if AllTerminal1 == False then AllTerminal == False;
            end
            Add NextStateValue to MyMoveValueList;
        end
        if MyMoveValueList ≠ ⊘ then
            MyBestValue ← choose the best for P in MyMoveValueList;
            Add (P,MyBestValue) to BestMoveValueList;
        end
    end
    if BestMoveValueList ≠ ⊘ then
        StateValue ← the best value in BestMoveValueList for each player;
    else
        /* terminal state, no legal move                              */
        StateValue ← goal concept evaluation for each player;
    end
end
```

Method II

In subgame search method I, it does a lot of state value comparisons. When two values from two different states have the same evaluation value for one player, i.e. both states can reach a state which has this concept evaluation for the player, then which one should the player chooses? We can not see any difference just from a number, though each value has a special path behind it and the path can do make difference in global game strategy; even some paths are not feasible when we combine them together with paths from other subgames. In order to have a more useful state value, we can record the number of moves leading to the state giving the value to the player. Moreover, we can split the number into two parts, one is the number of moves my player does and the other is the one other player(s) does. The sum of these two numbers is the total number of moves reaching the state which gives the evaluation value. But this step information does not really help too much, as we still don't know which one is better.

One possible solution is that we remember all turn-move sequences (see the following definition) which are sequences of moves with turn information for each state (i.e. who does this move). Each turn-move sequence has a corresponding local concept evaluation. The evaluation value format used in turn-move sequences is not exactly the same as the one used in method I. We can use any two different values to represent whether a local concept is satisfied or not after doing the moves of the turn-move sequence in the subgame. Let's first give the formal definition of turn-move sequence.

Definition 5.2. *(Turn-Move Sequence). A turn-move sequence is a tuple Seq =* ***(Ts,Ms,Es)*** *where*

- *Ts is a list of player names, indicated by* $T_1 \circ T_2 \circ ... \circ T_n$,
- *Ms is a list of moves, indicated by* $M_1 \circ M_2 \circ ... \circ M_n$,
- *Es is a set of evaluations of local concepts,*

where $n \geq 0$. *If Ts and Ms are both empty* ($n = 0$)*, then we call this turn-move sequence is an* ***empty turn-move sequence****. In the rest of this paper, we sometimes use sequences instead of turn-move sequences.*

Definition 5.3. *Two turn-move sequences* $Seq_1 = (Ts_1, Ms_1, Es_1)$ *and* $Seq_2 = (Ts_2, Ms_2, Es_2)$ *are* ***equal*** *iff* $Ts_1 = Ts_2$, $Ms_1 = Ms_2$ *and* $Es_1 = Es_2$.

Definition 5.4. *Two turn-move sequences* $Seq_1 = (Ts_1, Ms_1, Es_1)$ *and* $Seq_2 = (Ts_2, Ms_2, Es_2)$ *are* ***turn equal****, iff* $Ts_1 = Ts_2$. *If two turn-move sequences are equal, they are also turn equal.*

Definition 5.5. *A local concept C is **satisfied** (indicated by $Seq \models C$) in a turn-move sequence Seq iff the local concept's evaluation in the turn-move sequence's evaluation set is true. Otherwise, we call the local concept is not satisfied in this turn-move sequence.*

Definition 5.6. *Turn-move sequence $Seq_1 = (Ts_1, Ms_1, Es_1)$ is **evaluation dominated** by turn-move sequence $Seq_2 = (Ts_2, Ms_2, Es_2)$ **under** a set of local concepts Cs iff*

- *Seq_1 and Seq_2 are turn equal,*
- *$\forall_{C \in Cs}(Seq_1 \models C \Rightarrow Seq_2 \models C)$.*

*If $\forall_{C \in Cs}(Seq_1 \models C \Leftrightarrow Seq_2 \models C)$, then we also call Seq_1 is **evaluation equal** to Seq_2 under local concepts Cs.*

In this subgame search method, the value for each subgame state is a set of turn-move sequences. For each terminal state, there is only one one empty turn-move sequence with local concept evaluations. For each nonterminal state, we need to check if any terminal concept is satisfied in the state, if so we will add an empty turn-move sequence with local concept evaluations for this state. Then for each player who has legal move in the state, we get all the turn-move sequences for the player by adding his turn and move in each sequence getting from its successor states reached by his moves. After we get all the turn-move sequences for a player, we need to simplify them by removing evaluation dominated sequences (only if these sequences are evaluation equal under local terminal concepts) under the player's local goal concepts. However, simplification might be not possible in some cases, for example, the goal value is related to *action independent subgame*, i.e. the length of turn-move sequence is related to goal concept evaluations. Algorithm 5.2 will return a set of turn-move sequences instead of a best evaluation value for each player.

Algorithm 5.2: SubgameSearchIIForAlternatingMove

```
Input: State: subgame state, CurrentDepth: current search depth,
       MaxDepth: max search depth
Output: TurnSequences: a set of turn-move sequences, AllTerminal:
        True/False
begin
    TurnSequences ← ∅;
    AllTerminal ← True;
    if State is global terminal then
        TurnSequences ← empty sequence with local concept evaluations;
        AllTerminal ← True;
        Return;
    else if CurrentDepth ≥ MaxDepth then
        TurnSequences ← empty sequence with local concept evaluations;
        AllTerminal ← False;
    else if at least one terminal concept is satisfied in the state then
        TurnSequences ← empty sequence with local concept evaluations;
    end
    Players ← all players;
    foreach P ∈ Players do
        MyTurnSequences ← ∅;
        LegalMoves ← all legal moves of P in State;
        foreach M ∈ LegalMoves do
            NextState ← update(State,M);
            NextDepth ← CurrentDepth+1;
            (TurnSequences1,AllTerminal1) ← SubgameSearchIIForAlternat-
            ingMove(NextState,NextDepth,MaxDepth);
            if AllTerminal1 == False then AllTerminal == False;
            TurnSequences2 ← {(P,M)·Seq | Seq ∈ TurnSequences1};
            MyTurnSequences ← MyTurnSequences ∪ TurnSequences2;
        end
        if MyTurnSequences ≠ ∅ then
            MyBestTurnSequences ← simplify sequences for player P;
            TurnSequences ← TurnSequences ∪ MyBestTurnSequences;
        end
    end
    if TurnSequences == ∅ then
        /* terminal state as no legal move for any player        */
        TurnSequences ← empty sequence with local concept evaluations;
    end
end
```

5.1.2 Global Game Search

Method I

With subgame search method I, we can get a best value for each player from each subgame. Using these subgame values, global game search is looking for a global best move for my player in current state. From a subgame state value, we know what is the possible best value (i.e. local goal concept evaluation) for each player in the subgame. The best value for each player also tells him who should move next in that subgame in order to get that value.

If it is my player's turn to go next, and there are more one subgame in the condition that my player have to move next in order to get my player's best value in those subgames, then my player can choose the most urgent one to go. The standard of judging if a subgame is more urgent than another can be different. For instance, we can use the difference between the best value my player might get if he moves next in a subgame and the worst value he might get if other players move next in this subgame; we say a subgame which has the biggest difference is the most urgent one. We can also combine with goal and terminal predicates to judge which subgame is most urgent.

The main algorithm of this global game search is iterative deepening depth-first search. The search stops only if all subgame searches are finished (i.e. all leaf nodes are terminal), and returns current best move for the player. The algorithm starts with a global game state, the player whom the best move is looking for, initial search depth and maximum search depth as inputs (see algorithm 5.3). The most important part of this global game search is the best move selection part which is going to find out the most urgent subgame to play next, and algorithm 5.4 shows one possible solution by using value difference.

Algorithm 5.3: GlobalGameSearchIForAlternatingMove

```
Input: State: global game state, Player, Depth: current search depth,
       MaxDepth: max search depth
Output: BestMove
begin
    StateValueList ← ∅;
    BestMoveValueListList ← ∅;
    AllTerminal ← ∅;
    AllSubgames ← all action dependent subgames;
    foreach Subgame∈AllSubgames do
        SubState ← subgame state for Subgame in State;
        (StateValue,BestMoveValueList,AllTerminal1) ←
        SubgameSearchIForAlternatingMove(SubState,0,Depth);
        Add StateValue to StateValueList;
        Add BestMoveValueList to BestMoveValueListList;
        if AllTerminal1==False then AllTerminal==False;
    end
    CurBestMove ←
    SelectBestMove(Player,StateValueList,BestMoveValueListList);
    Save CurBestMove;
    if AllTerminal==False then
        NextDepth ← Depth + 1;
        BestMove ← GlobalGameSearchIForAlternating-
        Move(State,Player,NextDepth,MaxDepth);
    else
        BestMove ← CurBestMove;
    end
end
```

Algorithm 5.4: SelectBestMove

Input: Player, StateValueList: a list of subgame state values, BestMoveValueListList: list of list of bestmove-value pair for each player in each subgame
Output: BestMove
begin
  Subgames ← all subgames where Player has legal moves;
  Subgames2 ← {SG | SG∈ Subgames and Player's best value comes from Player's move in SG};
  if *Subgames2* $\neq \oslash$ **then**
    Order Subgames2 by the difference between Player's best value and worst value;
    Choice ← the subgame which has the biggest difference in Subgames2;
  else
    Choice ← a subgame in Subgames in which Player can get a better value than playing in other subgames;
  end
  Return the best move for Player in Choice;
end

Method II

In single player game decomposition search in [Gue07], subgame search finds local plans (sequences of moves) in each subgame; combining local plans global game search returns global plan which is again a sequence of moves from local plans by keeping the original order of moves in local plans. They can do this for single player games, because there is only one player and the player can control every single move. In multi-player games, we can not get local plans from subgames, because we can not assume any turn order in subgame until we can solve nondeterministic choices of subgames. However, we can get a set of turn-move sequences instead of local plans from each subgame; with all these turn-move sequences, global game search will search on these sequences to find out which subgame and which move the player should play next.

Algorithm 5.5 will combine all subgames' turn-move sequences to search an optimal global plan. The main difference of this algorithm compared to the one in method I is the best move selection part. In method I the best move selection doesn't need too much search, while the best move selection in this method will do a lot of search with turn-move sequences. The search algorithms can be used in this selection part are the same as the ones used in normal alternating move game search, e.g. minimax.

One possible solution of this best move selection search is that it does search on states which only have turn-move sequences instead of fluents and legal subgames instead of normal legal moves; i.e. the subgames where the player can take a normal move in. Thus, in each state, we don't know what moves have been played to reach this state, but we know who have moved in each subgame with what kind of turn order (called turn prefix). We can store turn prefix of each subgame instead of turn-move sequences. With these turn prefixes we can easily get all turn-move sequences for current state and so the legal subgames for the player in current state. After a player taking a move in a subgame, in next state we add the player to the turn prefix of this subgame and continually search the new state. When search goes deeper, subgame prefixes become longer and the number of turn-move sequences of the state become smaller. In some states, we will get some empty turn-move sequences from each subgame by removing the turn prefix of this subgame (according to subgame search method II, we can only get a empty turn-move sequence if a subgame state is terminal or at least one terminal concept is satisfied). We combine all these empty sequences together with exactly one empty sequence from each subgame, and we can evaluate each combination (by using either their local concept evaluations or doing their move sequences to get current state) which is actually one state of the normal game tree. After we have evaluations of all combinations, we can remove some dominated combinations . After we select the best combination for each terminal state (or leaf node), i.e. we know what are the best moves for each player to reach the state, then we backward compute the best move for all intermedial states till current global state (say algorithms 5.6 and 5.8 for more details).

In the above best move selection search, as the search graph branches are subgames, each branch contains different number (at least one) of moves. Thus one path in this compact graph can be mapped to many paths of the normal search graph of the game, and we have to choose one best to return for each branch. The path selection method 5.8 used in this method is that, given two paths, they must share some moves or at least one point, so the player who made the first difference in these two paths can choose the best one for him and remove the other one. However, given more than two paths at the same time, we can not simply compare any two of them to select one and then use the one selected to compare with others. We have to do this backward from leaf nodes. The following example tells us the reason.

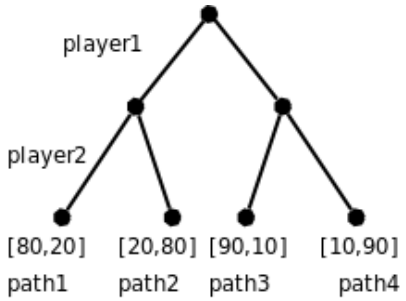

Figure 5.1: Path Selection Example

Example 5.1. In figure 5.1, there are four paths with length of 2. In each path *player1* plays the first move and *player2* takes the second one. There are two values in each path with the first one for *player1* and the second for *player2*. Now we get these four paths together to choose one best for all players (every player wants to get the best value for himself), if we first compare *path1* and *path3*, it is *player1*'s choice, so he will choose *path3*, then compare *path3* with *path2*, it is still *player2*'s choice and he will choose *path3* again. Finally, *player2* gets choice to choose between *path3* and *path4*, and *path4* will be selected. You can say the result is not optimal, the optimal path should be *path2*. To get optimal solution, we have to first compare *path1* with *path2* to get *path2*, and *path3* with *path4* to get *path4*, then *player1* chooses the best between *path2* and *path4* to get *path2*.

Moreover, to find the correct path comparison order might be very expensive if there are too many paths in one node, especially, we have to do this backward in each node. Another similar solution would be using normal game search but with legal moves getting from subgames' turn-move sequences instead of the ones from legal definitions (say algorithm 5.7). With this solution, we will get bigger game graph compared with algorithm 5.6, but we save the time to do path comparison and selection. In principle, the complexity of them will be the same.

One important property of algorithms 5.6 and 5.7 is that they both can always find optimal strategy as the one found by normal game search without decomposition. The following definition will be used in algorithm 5.6.

Definition 5.7. *Player P is **free** in a turn-move sequence $Seq_1 = (Ts_1, Ms_1, Es_1)$ of game (or subgame) G iff there exists another turn-move sequence $Seq_2 = (Ts_2, Ms_2, Es_2)$ of G such that*

- $Ts_2 = Ts_1 \circ Tail$, *where* $Tail \neq \oslash$,
- $Ms_1 = Ms \circ M_{1i} \circ \cdots \wedge\ Ms_2 = Ms \circ M_{2i} \circ \cdots \wedge\ M_{1i} \neq M_{2i} \wedge T_{1i} = P \wedge$ $\nexists_{Seq_3 \in G}(Seq_3 = (Ts_3, Ms_3, Es_3) \wedge Ts_3 = Ts_1 \wedge Ms_3 = Ms \circ M_{2i} \circ \cdots)$

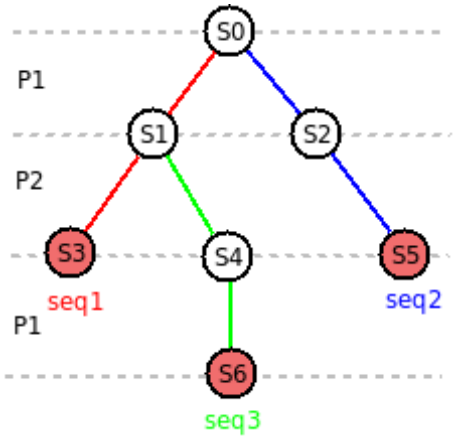

Figure 5.2: Small Game Tree

Example 5.2. The game in figure 5.2 has three sequences, *seq1*, *seq2* and *seq3*. There are two players *P1* and *P2*. *P2* is free in *seq1*; you might think *P1* is free in *seq2*, actually it is not, because of *seq1*, so no player is free in *seq2*; no one is free in *seq3* either.

Algorithm 5.5: GlobalGameSearchIIForAlternatingMove

```
Input: State: global game state, Player, Depth: current search depth,
       MaxDepth: max search depth
Output: BestMove
begin
    SubgameTurnSequences ← ∅;
    AllTerminal ← True;
    AllSubgames ← all action dependent subgames;
    foreach Subgame∈AllSubgames do
        SubState ← subgame state for Subgame from State;
        (TurnSequences1,AllTerminal1) ←
        SubgameSearchIIForAlternatingMove(SubState,0,Depth);
        SubgameTurnSequences ← SubgameTurnSequences∪TurnSequences1;
        if AllTerminal1==False then AllTerminal==False;
    end
    CurBestMove ← SelectBestMoveII_A or SelectBestMoveII_B to get
    bestmove;
    Save CurBestMove;
    if AllTerminal==False then
        NextDepth ← Depth + 1;
        BestMove ← GlobalGameSearchIIForAlternating-
        Move(State,Player,NextDepth,MaxDepth);
    else
        BestMove ← CurBestMove;
    end
end
```

Algorithm 5.6: SelectBestMoveII_A

```
Input: Player, SubPrefixes, Turns, TurnMoveSeqs
Output: BestMove, RestBestMoveSeqs
begin
    EmptySeqs ← all empty sequences under SubPrefixes;
    if EmptySeqs ≠ ⊘ then
        SeqCombinations ← evaluate combinations of all sequences in
        EmptySeqs with one from each subgame;
        NotFreeComs ← all C ∈ SeqCombinations and Player is not free in C;
        NotFreeBestCom ← ⊘;
        if NotFreeComs ≠ ⊘ then
            NotFreeBestCom ← ChooseBestCom(Turns,NotFreeComs);
            if value for last player in NotFreeBestCom is maximum then
                BestMove ← ⊘;
                RestBestMoveSeqs ← all move sequences in NotFreeBestCom;
            end
        else
            FreeComs ← all C ∈ SeqCombinations and only Player is free in C;
            FreeBestCom ← ⊘;
            if FreeComs ≠ ⊘ then
                FreeBestCom ← ChooseBestCom(Turns,FreeComs);
                if value for Player in FreeBestCom is maximum then
                    BestMove ← ⊘;
                    RestBestMoveSeqs ← all move sequences in FreeBestCom;
                end
            end
        end
        RestComs ←
        (SeqCombinations\(NotFreeComs∪FreeComs))∪{NotFreeBestCom,FreeBestCom};
    else
        RestComs ← ⊘;
    end
    ChildSeqs ← ⊘;
    LegalSubgames ← all subgames where Player can move next;
    NewTurns ← append Player to HistoryTurns;
    foreach Subgame ∈ LegalSubgames do
        NextPlayer ← get next player;
        NewSubPrefixes ← add Player in prefix of Subgame in SubPrefixes;
        RestBestMoveSeqs1 ←
        SelectBestMoveII_A(NextPlayer,NewSubPrefixes,NewTurns,TurnMoveSeqs);
        ChildSeqs ← ChildSeqs ∪ {RestBestMoveSeqs1};
    end
    AllComs ← RestComs ∪ ChildSeqs;
    BestCom ← ChooseBestCom(NewTurns,AllComs);
    Get BestMove and RestBestMoveSeqs from BestCom;
end
```

Algorithm 5.7: SelectBestMoveII_B

```
Input: Player, State, TurnMoveSeqs
Output: BestMove, Value
begin
    if State is terminal then
        Value ← get goal value for all players;
        BestMove ← ⊘;
    else
        BestMove ← ⊘;
        Value ← ⊘;
        LegalMoves ← get all legal moves for Player from TurnMoveSeqs;
        if LegalMoves ≠ ⊘ then
            foreach Move ∈ LegalMoves do
                RestTurnMoveSeqs ← rest sequences in TurnMoveSeqs after
                doing move Move;
                NextPlayer ← get next player;
                NextState ← update(State,Move);
                ChildValue ←
                SelectBestMoveII_B(NextPlayer,NextState,RestTurnMoveSeqs);
                if BestMove == ⊘ then
                    BestMove ← Move;
                    Value ← ChildValue;
                else
                    Value ← choose the best between Value and ChildValue for
                    Player;
                    BestMove ← choose the move following Value;
                end
            end
        else
            Value ← use heuristic function to get goal value for all players;
            BestMove ← ⊘;
        end
    end
end
```

Algorithm 5.8: ChooseBestCom

```
Input: Turns, SeqCombinations
Output: BestCom
begin
    Player ← last turn in Turns;
    NextTurns ← Turns without last turn;
    RestComs ← SeqCombinations;
    while RestComs has more than one element do
        Group RestComs by the history moves during NextTurns;
        RestComs ← for each group keep one best for Player;
        Player ← last turn in NextTurns;
        NextTurns ← NextTurns without last turn;
    end
    BestCom ← the last element in RestComs;
end
```

5.2 Simultaneous Move Games

In simultaneous move games, all players have to do a move in each turn. This expressed in GDL is that no player can do *noop* move which is the only move for player who is not in control in alternating move games. In each turn, every player only knows what legal moves other players can do, but the player does not know what exact moves they will choose. If a simultaneous move game is decomposable, each player, in each turn, has to choose between subgames, so more than one player might choose to move in one subgame. This makes the subgame search is different from the one used for alternating move games, and so is the global game search.

5.2.1 Subgame Search

The only difference between this subgame search and the one for alternating move games is that, in each state, this subgame has to consider joint moves of all players. We can easily adapt subgame search method I and II of alternating move games by adding joint moves in each state's expanding. This adapting results in that in method I, a player might get a best value from a joint move, and that in method II, turn-move sequences will have joint moves and joint turns, so the simplification of sequences has to handle these joint turns as well. In principle, we don't need to change subgame search too much.

5.2.2 Global Game Search

We can also reuse global game search methods for alternating move games except the best move selection part. In method I's best move selection part we might give priority to single moves and joint moves, and change the standard of urgency judgment. In method II's best move selection part we can use the normal search methods for simultaneous move games. We skip the details of these two methods here.

5.3 Complexity Comparison

The complexity of alternating move subgame search is the complexity of DFS. The time complexity is $O(\mid V \mid + \mid E \mid)$ where V and E are the number of states and edges of subgame respectively. The space complexity is the largest path of the game tree. Assume a game G has n subgames, G_1, G_2, ..., G_n with V_1, V_2, ..., V_n states and E_1, E_2, ..., E_n edges respectively. Then the number of states for game G is $SG = V_1 * V_2 * ... * V_n$. If we also use DFS to search the whole game G, the time complexity will be $O(\mid V \mid + \mid E \mid)$ where V = SG and E ($> E_1 + E_2 + ... + E_n$) is the number of edges of game G. If we use above decomposition search for game G, the time complexity will be $O(\mid V \mid + \mid E \mid)$ with $V = V_1 + V_2 + ... + V_n$ and $E = E_1 + E_2 + ... + E_n$. Suppose the average number of states for each subgame is SV, then the time complexity of normal game search with DFS or IDDFS is $O(\mid SV^n \mid + \mid E_1 \mid)$, while the time complexity of decomposition search is $O(\mid n * SV \mid + \mid E_2 \mid)$ where n is the number of subgames and $E_1 > E_2$.

Example 5.3. An alternating move game has two normal tictactoe games with two players (called *double tictactoe*, say the GDL definition of it in appendix A), and every time only one player can move and can only choose one subgame to move. The approximate number of states of the game is 18!, and the approximate number of states for each subgame is not the same as normal tictactoe as 9! (i.e. $362,880$), because the number of 'X' cells and the number of 'O' cells in the subgame do not have the constraint like in normal tictactoe. The approximate number of states of each subgame is $18 * 16 * 14 * ... * 2$ which is $17 * 15 * 13 * ... * 1$ times smaller than 18!. An additional time complexity in subgame game search method II is turn-move sequences simplification that varies from game to game. The worst case is that no sequence can be removed, so in each state we will get b^d sequences where b is the average branching factor and d is the average depth from the state to leaf nodes. Adding this simplification complexity to each state, we will get the worst time complexity for subgame search method II: $O(d * b^d)$ where b is the average branching factor and d is the average depth of the subgame. But in most

games, many sequences will be ruled out by simplification, e.g. subgame game search method II will return 912 simplified turn-move sequences which is very small compared to the total number of sequences (about $18 * 16 * 14 * ... * 2$). If subgame search does not use sequence simplification, then in global game search's best move selection will have more states (combinations) to evaluate, which just transfers the complexity from one place to another.

In global game search method II, the time complexity of *BestMoveSelectII_A* is $O(n^d)$, where n is the number of subgames and d is the depth of the game, pluses the time complexity of sequence combination evaluation and path selection which depends on how many sequences each subgame has, e.g. if there are n subgames and each subgame has s_i (i is the index of subgame) sequences, then the worst sequence combination evaluation complexity would be $O(s_1 * s_2 * ... * s_n)$. The time complexity of *BestMoveSelectII_B* is the same as *BestMoveSelectII_A*, because *BestMoveSelectII_B* just does sequence combination evaluation during graph expanding instead of in leaf nodes, and it does path selection during the graph expanding as well, but it will spend more time on graph expanding.

5.4 Experimental Results

We have implemented and tested method II with the double tictactoe example in previous section. The following four tables show the results: the first two show the subgame search time cost (in second) for one subgame (i.e. a single tictactoe), and the last two show the time cost of finding the first optimal best move for normal search and decomposition search with different search depth (the depth in these tables is related to subgame search depth, so mapped to normal game search graph the depth will be the sum of all subgame search depths).

One Subgame of Double-Tictactoe					
Search Depth	1	2	3	4	5
All Sequences	18	288	4032	47328	483840
Simplified Seqs	2	4	10	26	64
Time Cost(s)	0.17	2	10	28	55

Table 5.1: Subgame Search Method II Results for Double-Tictactoe

One Subgame of Double-Tictactoe				
Search Depth	6	7	8	9
All Sequences	3870720	23224320	92897280	185794560
Simplified Seqs	148	324	674	912
Time Cost(s)	127	469	678	790

Table 5.2: Subgame Search Method II Results for Double-Tictactoe cont.

Time Cost(s)	**Search Depth**				
	1	2	3	4	5
Decomposition Search	0.36	4.36	24	80	179
Normal Search	< 1800			$> 3600 * 4$	

Table 5.3: Search Results Comparison for Double-Tictactoe

Time Cost(s)	**Search Depth**			
	6	7	8	9
Decomposition Search	301	1022	1530	1793
Normal Search	$> 3600 * 4$			

Table 5.4: Search Results Comparison for Double-Tictactoe cont.

Chapter 6

Special Cases for Parallel and Serial Games

In principle, parallel and serial games are not decomposable by using the subgame detection algorithm in chapter 3. These two types of games do have subgames, but the subgames are not independent as they are loosely connected with some special relations. Thus if we could find out these special relations, then we can use these relations to detect subgames and use subgame search to fasten game search.

6.1 Parallel Games

The main property of parallel games is that there are at least two *action dependent subgames* (subgames are not action independent), but these subgames are not independent, because they are connected, according to the subgame independence definition in chapter 3, by the compound moves of the game. A compound move is a move that contains one move for each subgame. For example, a parallel game contains two tictactoe subgames, and one compound move form in this game is *(mark ?x1 ?y1 ?x2 ?y2)* where *?x1* and *?y1* give the move position in one tictactoe and *?x2* and *?y2* give the move postion in another one. See appendix B for one possible GDL definition of parallel tictactoe.

6.1.1 Special for Subgame Detection

We can not simply use subgame detection algorithm in chapter 3 for parallel games because of compound moves connecting all subgames together. Thus parallel games

are not decomposable by using our subgame detection algorithm until we can split compound moves to subgame moves.

Given a normal game, how can we detect that a move of the game is compound (also call it decomposable), i.e. it can be splitted into submoves. The fact is that if a move is decomposable, the move has at least two arguments, each argument will be used in some (but not all) of its effect rules (namely *next* rules). One argument of a move is used in a *next* rule iff the occurrence of the argument in the rule is greater than one.

Example 6.1. One next rule of compound move *(mark ?x1 ?y1 ?x2 ?y2)* in parallel tictactoe is the following. We can see that the first two arguments of this move are used in this rule and that the last two arguments are not used.

```
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
    (distinctcell ?x ?y ?x1 ?y1))
```

For each move, which has more than one argument, of a game, we collect all the move's *next* rules, then for each rule we can find out what arguments of the move are used in the rule. Thus each *next* rule depends on some argument(s) of the move. With this argument dependency relation, we can find all connected components of each move's next rules. If there are more than one component for a move, then we can only say that the move has chance to be splitted, because it also depends on the move's *legal* definition(s). If we find out that two arguments from two different next rule components occur in one fluent in the move's legal definition, then we can not split the move because we don't want to split a fluent as splitting fluents is not helpful in parallel games.

Example 6.2. The following rules are all *next* rules for move **mark/4** in parallel tictactoe in appendix B. We give a name to each argument of the move, from left to right they are a_1, a_2, a_3 and a_4. We can find that rules 1, 2, 3 and 4 depend on a_1; rules 1, 2, 3 and 4 depend on a_2; rules 5, 6, 7 and 8 depend on a_3; and rules 5, 6, 7 and 8 depend on a_4. Now the connected components are easy to see, where rules 1, 2, 3 and 4 with arguments a_1 and a_2 belong to one component and rules 5, 6, 7 and 8 with arguments a_3 and a_4 belong to another one.

```
(1).(<= (next (cell1 ?x1 ?y1 x))
        (does xplayer (mark ?x1 ?y1 ?x2 ?y2)))
(2).(<= (next (cell1 ?x1 ?y1 o))
        (does oplayer (mark ?x1 ?y1 ?x2 ?y2)))
(3).(<= (next (cell1 ?x ?y ?mark))
        (true (cell1 ?x ?y ?mark))
        (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
        (distinctcell ?x ?y ?x1 ?y1))
```

```
(4).(<= (next (cell1 ?x ?y ?mark))
        (true (cell1 ?x ?y ?mark))
        (does oplayer (mark ?x1 ?y1 ?x2 ?y2))
        (distinctcell ?x ?y ?x1 ?y1))
(5).(<= (next (cell2 ?x2 ?y2 x))
        (does xplayer (mark ?x1 ?y1 ?x2 ?y2)))
(6).(<= (next (cell2 ?x2 ?y2 o))
        (does oplayer (mark ?x1 ?y1 ?x2 ?y2)))
(7).(<= (next (cell2 ?x ?y ?mark))
        (true (cell2 ?x ?y ?mark))
        (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
        (distinctcell ?x ?y ?x2 ?y2))
(8).(<= (next (cell2 ?x ?y ?mark))
        (true (cell2 ?x ?y ?mark))
        (does oplayer (mark ?x1 ?y1 ?x2 ?y2))
        (distinctcell ?x ?y ?x2 ?y2))
```

After we get the above two components, the move **mark/4** has chance to be splitted into two submoves: one with first two arguments and the other with last two arguments. The final decision will be made by *legal* definition of the move: the *legal* definition of this move is the following, where no two arguments from different components occur in one fluent, so the move can be splitted.

```
(<= (legal ?player (mark ?x1 ?y1 ?x2 ?y2))
    (true (control ?player))
    (true (cell1 ?x1 ?y1 b))
    (true (cell2 ?x2 ?y2 b)))
```

After we find all decomposable moves, we can either split (or rewrite) corresponding *legal* and *next* rules or keep the rules unchanged. If we rewrite the rules, then we probably need to redo other stuff which is based on the game's original rules. A better way would be keeping the rules unchanged, but split precondition, positive and negative effects of compound moves to the corresponding relations of submoves, and use these new relations to detect subgames. If we can get subgames with submoves, then we keep submoves for each subgame. Afterwards, global game search extracts subgame states and submoves from global game state and global legal moves respectively, and combines best submoves got from subgame search together to get the best move of global game.

6.1.2 Special for Game Search

Depending on whether a game is an alternating move or a simultaneous move game, subgames will also be the same type (alternating or simultaneous) as the game, so

the subgame search methods are just normal alternating move and simultaneous move game search algorithms. The only additional work the search has to do is extracting subgame legal moves (i.e. submoves) from global legal moves and rebuilding global game moves back from submoves. One constraint of the global game search is that it has to find equal length best solution (or plan) in each subgame to get global best solution.

6.2 Serial Games

A serial game has at least two *action dependent subgames*, but these subgames are not independent because all subgames are played in a special order specified in the game, i.e. end conditions of one subgame are the preconditions of the one after it. Except this order relation, all subgames are independent, thus if we can detect this order, then we can search the game by searching its subgames one by one, which will be much faster than do the whole game search.

6.2.1 Special for Subgame Detection

We know that the subgames connect each other only from one's end conditions to another's preconditions, i.e. in both games' *legal* definitions, they share some preconditions but in a different manner: one precondition required to be true in one subgame is required to be false in another one. So for each pair of moves, they either share some preconditions in the same manner (both negative or positive) or in a different manner (positive in one and negative in another one), or do not share any precondition. Given any two moves, if they share any precondition (except control fluent, which we mentioned the reason in subgame detection in chapter 3) in the same manner, they must belong to one subgame; if they do not share any precondition, they may or may not belong to one subgame depending on if they share any positive or negative effect, if so, they belong to one subgame; if they only share precondition(s) in a different manner, that means they will never become legal at the same time, then they belong to different subgames. Using above rules, we can group all moves which share any precondition in the same manner or any positive or negative effect. Then if we get more than one group, the remain relations between moves from different groups are that they either only sharing precondition(s) in a different manner or not sharing any precondition. If moves in two groups share preconditions, then we check that if all moves in one group share the same preconditions with all moves in another group in the same manner, and that if the sharing precondition(s) is only positive or negative effect(s) of one group

(actually this is fulfilled in the first grouping step). If this is the case, then we say moves in these two groups are ordered and the group which has some sharing preconditions as positive or negative effects is played first. Finally for each pair of groups, we will get the order if they have and each group is actually one subgame we want to detect.

Example 6.3. In appendix C, you can find a serial game example including two tictactoe subgames which are played in order. The two tictactoe subgames share preconditions *game1over* (including a set of ground instances of fluents) in a different way. The game with fluent *cell1/3* (called *game1*) requires that *game1over* is false to get legal moves of *game1*, while the game with fluent *cell2/3* (called *game2*) requires that *game1over* is ture to get legal moves of *game2*. Thus they will never get legal moves at the same time. Moreover, the ground instances of fluents in *game1over* only accur as positive or negative effects of *game1*, so we say these two subgames are ordered and *game1* is played first.

6.2.2 Special for Game Search

The subgames of a serial game have exactly the same type, namely alternating or simultaneous, as the serial game. So the subgame search methods will be the same as the ones used for normal alternating move or simultaneous move games. Global game search will just need to follow the order of subgames to control when start to search which subgame and directly return best move got from subgame search.

6.3 Complexity Comparison

For Parallel Games

Assume there are N subgames of a parallel game and for subgame i there are V_i states, then the time complexity of normal game search without decomposition is $O(V_1 * V_2 * ... * V_n)$, while the time complexity with decomposition search will be $O(V_1 + V_2 + ... + V_n)$ which is exponentially smaller than the first one.

For Serial Games

Assume there are N subgames of a serial game and for subgame i there are V_i states and T_i terminal states, then the time complexity of decomposition will be $O(V_1 + V_2 + ... + V_n)$, whereas the time complexity without decomposition will be $O(V_1 + T_1 * V_2 + T_1 * T_2 * V_3 + ... + T_1 * T_2 * ... * T_{n-1} * V_n)$. Subgame terminal states T_i depends on how terminal condition is defined in each subgame; and the terminal

states of one subgame can be many. If every subgame has exactly one terminal state, then decomposition search state space is the same as normal game search without decomposition. Moreover, subgame states only containt fluents for one subgame, while the states in normal search will contain all fluents of all subgames. Therefor, in any case, it is worth to do decomposition search, though we might get the same time complexity as normal search.

Chapter 7

Conclusion

During this decomposition research of multi-player games, we have analyzed four classes of multi-player games: impartial games, general partial games, parallel games and serial games. These four classes are not completely independent, e.g. a parallel game can contain impartial games or general partial games as its subgames. Thus we can apply subgame detection again on subgames to get smaller subgames of subgames if they are decomposable. We have extended subgame detection algorithm in [Gue07] to handle the arguments of fluents and moves, and the arguments of fluents and moves will be instantiated, if possible, before subgame detection. However, subgame detection in chapter 3 is not working for parallel and serial games. For those games, we have proposed possible adaptions (with compound move and game order detection) for subgame detection in chapter 6.

We have proposed different decomposition search methods for these four classes of games:

- **Impartial Games**
  The time complexity of decomposition search is linear of the biggest subgame size, because every finite impartial game is equivalent to a Nim heap (in normal play) that yields the same outcome when played in parallel with this impartial game. Thus we can use the strategies of playing game Nim to play all impartial games. The main advantage is that, with subgame search results (nimbers), we are able to use nim addition to find global best move very quickly. The quality of the strategy found by decomposition search is as optimal as the one found by normal search (without decomposition).

- **General Partial Games**
  We have mentioned two decomposition search methods for these games. The

complexity of method I is much smaller compared to method II, but the quality of the strategy found by method I is not as optimal as the one found by method II. The main assumption of subgame search method I is that the goal concepts for each player in each subgame are monotonic. Although we can easily change this method to remove the assumption, it does not improve the quality of its solution. This monotonic assumption is not required for method II as the turn-move sequences of subgames will return all sequences if the evaluations of sequences are not comparable (i.e. no one is evaluation dominated by another under some local concepts). However, the subgame search method II will take some time to do turn-move sequences selection or tree expanding if there are too many sequences been returned by subgame search. Subgame search method II can also be improved by applying optimizations, like alpha-beta pruning. Nevertheless, the time complexity of method II is exponentially smaller than the one of normal search, and the quality of the strategy found by decomposition search is as optimal as the one found by normal search.

- **Parallel Games**
  The subgame search methods for parallel games are just normal algorithms we used for normal games; the global game search of these games will controll subgame search and make sure that subgame search finds equal length best strategy in all subgames. Moreover, the game search has to extract submoves from global legal moves and rebuild global moves from submoves. The time complexity of decomposition search is also exponentially smaller than the one of normal search and the quality of the strategy found by decomposition search is also optimal.

- **Serial Games**
  The subgame search is the same as for parallel games, and the global game search will just follow the order of subgames to search one after another and directly return best move got from subgame search as global best move. The time complexity of decomposition search is much smaller than the one of normal search. The quality of strategy found by decomposition search is also optimal.

Based on *Fluxplayer*[1], we have implemented subgame detection (see chapter 3) which can detect subgames of normal partial and impartial games (the adaptions for parallel and serial games can be updated in the future). Impartial property checking and decomposition search for impartial games are also successfully implemented and tested (results can found in chapter 4). For general partial games, we have implemented method II for alternating move games (results in chapter 5). All the

[1] `http://www.fluxagent.org`

experimental results show that decomposition search is much faster than normal search without decomposition, at least for the games we have tested. Although there might be some decomposable games where decomposition search does not fasten the game search too much, the time complexity will not be worse than normal search in principle.

Appendix A: Double TicTacToe in GDL

```
(role xplayer)
(role oplayer)

(init (cell1 1 1 b))
(init (cell1 1 2 b))
(init (cell1 1 3 b))
(init (cell1 2 1 b))
(init (cell1 2 2 b))
(init (cell1 2 3 b))
(init (cell1 3 1 b))
(init (cell1 3 2 b))
(init (cell1 3 3 b))
(init (cell2 1 1 b))
(init (cell2 1 2 b))
(init (cell2 1 3 b))
(init (cell2 2 1 b))
(init (cell2 2 2 b))
(init (cell2 2 3 b))
(init (cell2 3 1 b))
(init (cell2 3 2 b))
(init (cell2 3 3 b))
(init (control xplayer))

(<= (legal ?w (mark1 ?x ?y))
    (true (cell1 ?x ?y b))
    (true (control ?w))
    (not terminal1))
(<= (legal ?w (mark2 ?x ?y))
    (true (cell2 ?x ?y b))
    (true (control ?w))
    (not terminal2))
(<= (legal xplayer noop)
    (true (control oplayer)))
(<= (legal oplayer noop)
    (true (control xplayer)))

(<= (next (cell1 ?m ?n ?x))
    (does ?p (mark2 ?x2 ?y2))
    (true (cell1 ?m ?n ?x)))
(<= (next (cell1 ?m ?n x))
    (does xplayer (mark1 ?m ?n))
    (true (cell1 ?m ?n b)))
(<= (next (cell1 ?m ?n o))
    (does oplayer (mark1 ?m ?n))
    (true (cell1 ?m ?n b)))
(<= (next (cell1 ?m ?n ?w))
    (true (cell1 ?m ?n ?w))
    (distinct ?w b))
(<= (next (cell1 ?m ?n b))
    (does ?w (mark1 ?j ?k))
    (true (cell1 ?m ?n b))
    (or (distinct ?m ?j)
        (distinct ?n ?k)))
(<= (next (control xplayer))
    (true (control oplayer)))
(<= (next (control oplayer))
    (true (control xplayer)))

(<= (row1 ?m ?x)
    (true (cell1 ?m 1 ?x))
    (true (cell1 ?m 2 ?x))
    (true (cell1 ?m 3 ?x)))
(<= (column1 ?n ?x)
    (true (cell1 1 ?n ?x))
    (true (cell1 2 ?n ?x))
    (true (cell1 3 ?n ?x)))
(<= (diagonal1 ?x)
    (true (cell1 1 1 ?x))
    (true (cell1 2 2 ?x))
    (true (cell1 3 3 ?x)))
(<= (diagonal1 ?x)
    (true (cell1 1 3 ?x))
    (true (cell1 2 2 ?x))
    (true (cell1 3 1 ?x)))

(<= (line1 ?x) (row1 ?m ?x))
(<= (line1 ?x) (column1 ?m ?x))
(<= (line1 ?x) (diagonal1 ?x))

(<= open1
    (true (cell1 ?m ?n b)))

(<= (next (cell2 ?m ?n ?x))
    (does ?p (mark1 ?x1 ?y1))
    (true (cell2 ?m ?n ?x)))
(<= (next (cell2 ?m ?n x))
    (does xplayer (mark2 ?m ?n))
    (true (cell2 ?m ?n b)))
(<= (next (cell2 ?m ?n o))
    (does oplayer (mark2 ?m ?n))
    (true (cell2 ?m ?n b)))
(<= (next (cell2 ?m ?n ?w))
```

```
93      (true (cell2 ?m ?n ?w))
94      (distinct ?w b))
95  (<= (next (cell2 ?m ?n b))
96      (does ?w (mark2 ?j ?k))
97      (true (cell2 ?m ?n b))
98      (or (distinct ?m ?j)
99          (distinct ?n ?k)))
100
101 (<= (row2 ?m ?x)
102     (true (cell2 ?m 1 ?x))
103     (true (cell2 ?m 2 ?x))
104     (true (cell2 ?m 3 ?x)))
105 (<= (column2 ?n ?x)
106     (true (cell2 1 ?n ?x))
107     (true (cell2 2 ?n ?x))
108     (true (cell2 3 ?n ?x)))
109 (<= (diagonal2 ?x)
110     (true (cell2 1 1 ?x))
111     (true (cell2 2 2 ?x))
112     (true (cell2 3 3 ?x)))
113 (<= (diagonal2 ?x)
114     (true (cell2 1 3 ?x))
115     (true (cell2 2 2 ?x))
116     (true (cell2 3 1 ?x)))
117
118 (<= (line2 ?x) (row2 ?m ?x))
119 (<= (line2 ?x) (column2 ?m ?x))
120 (<= (line2 ?x) (diagonal2 ?x))
121
122 (<= open2
123     (true (cell2 ?m ?n b)))
124
125 (<= (goal xplayer 100)
126     (line1 x)
127     (line2 x))
128 (<= (goal xplayer 75)
129     (not (line1 x))
130     (not (line1 o))
131     (not open1)
132     (line2 x))
133 (<= (goal xplayer 75)
134     (line1 x)
135     (not (line2 x))
136     (not (line2 o))
137     (not open2))
138 (<= (goal xplayer 50)
139     (not (line1 x))
140     (not (line1 o))
141     (not open1)
142     (not (line2 x))
143     (not (line2 o))
144     (not open2))
145 (<= (goal xplayer 50)
146     (line1 x)
147     (line2 o))
148 (<= (goal xplayer 50)
149     (line2 x)
150     (line1 o))
151 (<= (goal xplayer 25)
152     (not (line1 x))
153     (not (line1 o))
154     (not open1)
155     (line2 o))
156 (<= (goal xplayer 25)
157     (line1 o)
158     (not (line2 x))
159     (not (line2 o))
160     (not open2))
161 (<= (goal xplayer 0)
162     (line1 o)
163     (line2 o))
164 (<= (goal oplayer 100)
165     (line1 o)
166     (line2 o))
167 (<= (goal oplayer 75)
168     (not (line1 x))
169     (not (line1 o))
170     (not open1)
171     (line2 o))
172 (<= (goal oplayer 75)
173     (line1 o)
174     (not (line2 x))
175     (not (line2 o))
176     (not open2))
177 (<= (goal oplayer 50)
178     (line1 o)
179     (line2 x))
180 (<= (goal oplayer 50)
181     (line1 x)
182     (line2 o))
183 (<= (goal oplayer 50)
184     (not (line1 x))
185     (not (line1 o))
186     (not open1)
187     (not (line2 x))
188     (not (line2 o))
189     (not open2))
190 (<= (goal oplayer 25)
```

```
191       (not (line1 x))
192       (not (line1 o))
193       (not open1)
194       (line2 x))
195   (<= (goal oplayer 25)
196       (line1 x)
197       (not (line2 x))
198       (not (line2 o))
199       (not open2))
200   (<= (goal oplayer 0)
201       (line1 x)
202       (line2 x))
203
204   (<= terminal
205       terminal1
206       terminal2)
207   (<= terminal1
208       (line1 x))
209   (<= terminal1
210       (line1 o))
211   (<= terminal1
212       (not open1))
213   (<= terminal2
214       (line2 x))
215   (<= terminal2
216       (line2 o))
217   (<= terminal2
218       (not open2))
```

Appendix B: Parallel TicTacToe in GDL

```
(role xplayer)
(role oplayer)
(init (cell1 1 1 b))
(init (cell1 1 2 b))
(init (cell1 1 3 b))
(init (cell1 2 1 b))
(init (cell1 2 2 b))
(init (cell1 2 3 b))
(init (cell1 3 1 b))
(init (cell1 3 2 b))
(init (cell1 3 3 b))
(init (cell2 1 1 b))
(init (cell2 1 2 b))
(init (cell2 1 3 b))
(init (cell2 2 1 b))
(init (cell2 2 2 b))
(init (cell2 2 3 b))
(init (cell2 3 1 b))
(init (cell2 3 2 b))
(init (cell2 3 3 b))
(init (control xplayer))
(<= terminal
    (line1 x))
(<= terminal
    (line1 o))
(<= terminal
    (not open1))
(<= terminal
    (line2 x))
(<= terminal
    (line2 o))
(<= terminal
    (not open2))
(<= (goal xplayer 100)
    goalxplayer1 -100
    goalxplayer2 -100)
(<= (goal xplayer 75)
    goalxplayer1 -100
    goalxplayer2 -50)
(<= (goal xplayer 75)
    goalxplayer1 -50
    goalxplayer2 -100)
(<= (goal xplayer 50)
    goalxplayer1 -100
    goalxplayer2 -0)
(<= (goal xplayer 50)
    goalxplayer1 -50
    goalxplayer2 -50)
(<= (goal xplayer 50)
    goalxplayer1 -0
    goalxplayer2 -100)
(<= (goal xplayer 25)
    goalxplayer1 -50
    goalxplayer2 -0)
(<= (goal xplayer 25)
    goalxplayer1 -0
    goalxplayer2 -50)
(<= (goal xplayer 0)
    goalxplayer1 -0
    goalxplayer2 -0)
(<= (goal oplayer 100)
    goaloplayer1 -100
    goaloplayer2 -100)
(<= (goal oplayer 75)
    goaloplayer1 -100
    goaloplayer2 -50)
(<= (goal oplayer 75)
    goaloplayer1 -50
    goaloplayer2 -100)
(<= (goal oplayer 50)
    goaloplayer1 -100
    goaloplayer2 -0)
(<= (goal oplayer 50)
    goaloplayer1 -50
    goaloplayer2 -50)
(<= (goal oplayer 50)
    goaloplayer1 -0
    goaloplayer2 -100)
(<= (goal oplayer 25)
    goaloplayer1 -50
    goaloplayer2 -0)
(<= (goal oplayer 25)
    goaloplayer1 -0
    goaloplayer2 -50)
(<= (goal oplayer 0)
    goaloplayer1 -0
    goaloplayer2 -0)
(<= goalxplayer1 -100
    (line1 x))
(<= goalxplayer1 -50
    (not (line1 x))
    (not (line1 o))
```

```
93        (not open1))
94    (<= goalxplayer1-0
95        (line1 o))
96    (<= goalxplayer1-0
97        (not (line1 x))
98        (not (line1 o))
99        open1)
100   (<= goaloplayer1-100
101       (line1 o))
102   (<= goaloplayer1-50
103       (not (line1 x))
104       (not (line1 o))
105       (not open1))
106   (<= goaloplayer1-0
107       (line1 x))
108   (<= goaloplayer1-0
109       (not (line1 x))
110       (not (line1 o))
111       open1)
112   (<= goalxplayer2-100
113       (line2 x))
114   (<= goalxplayer2-50
115       (not (line2 x))
116       (not (line2 o))
117       (not open2))
118   (<= goalxplayer2-0
119       (line2 o))
120   (<= goalxplayer2-0
121       (not (line2 x))
122       (not (line2 o))
123       open2)
124   (<= goaloplayer2-100
125       (line2 o))
126   (<= goaloplayer2-50
127       (not (line2 x))
128       (not (line2 o))
129       (not open2))
130   (<= goaloplayer2-0
131       (line2 x))
132   (<= goaloplayer2-0
133       (not (line2 x))
134       (not (line2 o))
135       open2)
136   (<= (row1 ?x ?mark)
137       (true (cell1 ?x 1 ?mark))
138       (true (cell1 ?x 2 ?mark))
139       (true (cell1 ?x 3 ?mark)))
140   (<= (column1 ?y ?mark)
141       (true (cell1 1 ?y ?mark))
142       (true (cell1 2 ?y ?mark))
143       (true (cell1 3 ?y ?mark)))
144   (<= (diagonal1 ?mark)
145       (true (cell1 1 1 ?mark))
146       (true (cell1 2 2 ?mark))
147       (true (cell1 3 3 ?mark)))
148   (<= (diagonal1 ?mark)
149       (true (cell1 1 3 ?mark))
150       (true (cell1 2 2 ?mark))
151       (true (cell1 3 1 ?mark)))
152   (<= (line1 ?mark)
153       (row1 ?x ?mark))
154   (<= (line1 ?mark)
155       (column1 ?y ?mark))
156   (<= (line1 ?mark)
157       (diagonal1 ?mark))
158   (<= open1
159       (true (cell1 ?x ?y b)))
160   (<= (row2 ?x ?mark)
161       (true (cell2 ?x 1 ?mark))
162       (true (cell2 ?x 2 ?mark))
163       (true (cell2 ?x 3 ?mark)))
164   (<= (column2 ?y ?mark)
165       (true (cell2 1 ?y ?mark))
166       (true (cell2 2 ?y ?mark))
167       (true (cell2 3 ?y ?mark)))
168   (<= (diagonal2 ?mark)
169       (true (cell2 1 1 ?mark))
170       (true (cell2 2 2 ?mark))
171       (true (cell2 3 3 ?mark)))
172   (<= (diagonal2 ?mark)
173       (true (cell2 1 3 ?mark))
174       (true (cell2 2 2 ?mark))
175       (true (cell2 3 1 ?mark)))
176   (<= (line2 ?mark)
177       (row2 ?x ?mark))
178   (<= (line2 ?mark)
179       (column2 ?y ?mark))
180   (<= (line2 ?mark)
181       (diagonal2 ?mark))
182   (<= open2
183       (true (cell2 ?x ?y b)))
184   (<= (distinctcell ?x1 ?y1 ?x2 ?y2)
185       (cell ?x1 ?y1)
186       (cell ?x2 ?y2)
187       (distinct ?x1 ?x2))
188   (<= (distinctcell ?x1 ?y1 ?x2 ?y2)
189       (cell ?x1 ?y1)
190       (cell ?x2 ?y2)
```

```
191     (distinct ?y1 ?y2))
192 (cell 1 1)
193 (cell 1 2)
194 (cell 1 3)
195 (cell 2 1)
196 (cell 2 2)
197 (cell 2 3)
198 (cell 3 1)
199 (cell 3 2)
200 (cell 3 3)
```

```
201 (<= (legal ?player (mark ?x1 ?y1 ?x2 ?y2))
202     (true (control ?player))
203     (true (cell1 ?x1 ?y1 b))
204     (true (cell2 ?x2 ?y2 b)))
205 (<= (legal xplayer noop)
206     (true (control oplayer)))
207 (<= (legal oplayer noop)
208     (true (control xplayer)))
209 (<= (next (cell1 ?x1 ?y1 x))
210     (does xplayer (mark ?x1 ?y1 ?x2 ?y2)))
211 (<= (next (cell1 ?x1 ?y1 o))
212     (does oplayer (mark ?x1 ?y1 ?x2 ?y2)))
213 (<= (next (cell1 ?x ?y ?mark))
214     (true (cell1 ?x ?y ?mark))
215     (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
216     (distinctcell ?x ?y ?x1 ?y1))
217 (<= (next (cell1 ?x ?y ?mark))
218     (true (cell1 ?x ?y ?mark))
219     (does oplayer (mark ?x1 ?y1 ?x2 ?y2))
220     (distinctcell ?x ?y ?x1 ?y1))
221 (<= (next (cell2 ?x2 ?y2 x))
222     (does xplayer (mark ?x1 ?y1 ?x2 ?y2)))
223 (<= (next (cell2 ?x2 ?y2 o))
224     (does oplayer (mark ?x1 ?y1 ?x2 ?y2)))
225 (<= (next (cell2 ?x ?y ?mark))
226     (true (cell2 ?x ?y ?mark))
227     (does xplayer (mark ?x1 ?y1 ?x2 ?y2))
228     (distinctcell ?x ?y ?x2 ?y2))
229 (<= (next (cell2 ?x ?y ?mark))
230     (true (cell2 ?x ?y ?mark))
231     (does oplayer (mark ?x1 ?y1 ?x2 ?y2))
232     (distinctcell ?x ?y ?x2 ?y2))
233 (<= (next (control xplayer))
234     (true (control oplayer)))
235 (<= (next (control oplayer))
236     (true (control xplayer)))
```

Appendix C: Serial TicTacToe in GDL

```
(role xplayer)
(role oplayer)
(init (cell1 1 1 b))
(init (cell1 1 2 b))
(init (cell1 1 3 b))
(init (cell1 2 1 b))
(init (cell1 2 2 b))
(init (cell1 2 3 b))
(init (cell1 3 1 b))
(init (cell1 3 2 b))
(init (cell1 3 3 b))
(init (control1 xplayer))
(init (cell2 1 1 b))
(init (cell2 1 2 b))
(init (cell2 1 3 b))
(init (cell2 2 1 b))
(init (cell2 2 2 b))
(init (cell2 2 3 b))
(init (cell2 3 1 b))
(init (cell2 3 2 b))
(init (cell2 3 3 b))
(init (control2 xplayer))
(<= (legal ?player (mark1 ?x ?y))
    (not gamelover)
    (true (control1 ?player))
    (true (cell1 ?x ?y b)))
(<= (legal xplayer noop)
    (not gamelover)
    (true (control1 oplayer)))
(<= (legal oplayer noop)
    (not gamelover)
    (true (control1 xplayer)))
(<= (legal ?player (mark2 ?x ?y))
    gamelover
    (true (control2 ?player))
    (true (cell2 ?x ?y b)))
(<= (legal xplayer noop)
    gamelover
    (true (control2 oplayer)))
(<= (legal oplayer noop)
    gamelover
    (true (control2 xplayer)))
(<= (next (cell1 ?x ?y x))
    (does xplayer (mark1 ?x ?y)))
(<= (next (cell1 ?x ?y o))
    (does oplayer (mark1 ?x ?y)))
(<= (next (cell1 ?x1 ?y1 ?mark))
    (true (cell1 ?x1 ?y1 ?mark))
    (does xplayer (mark1 ?x2 ?y2))
    (distinctcell ?x1 ?y1 ?x2 ?y2))
(<= (next (cell1 ?x1 ?y1 ?mark))
    (true (cell1 ?x1 ?y1 ?mark))
    (does oplayer (mark1 ?x2 ?y2))
    (distinctcell ?x1 ?y1 ?x2 ?y2))
(<= (next (cell1 ?x ?y ?mark))
    (true (cell1 ?x ?y ?mark))
    gamelover)
(<= (next (control1 xplayer))
    (not gamelover)
    (true (control1 oplayer)))
(<= (next (control1 oplayer))
    (not gamelover)
    (true (control1 xplayer)))
(<= (next (control1 ?player))
    (true (control1 ?player))
    gamelover)
(<= (next (cell2 ?x ?y x))
    (does xplayer (mark2 ?x ?y)))
(<= (next (cell2 ?x ?y o))
    (does oplayer (mark2 ?x ?y)))
(<= (next (cell2 ?x1 ?y1 ?mark))
    (true (cell2 ?x1 ?y1 ?mark))
    (does xplayer (mark2 ?x2 ?y2))
    (distinctcell ?x1 ?y1 ?x2 ?y2))
(<= (next (cell2 ?x1 ?y1 ?mark))
    (true (cell2 ?x1 ?y1 ?mark))
    (does oplayer (mark2 ?x2 ?y2))
    (distinctcell ?x1 ?y1 ?x2 ?y2))
(<= (next (cell2 ?x ?y ?mark))
    (true (cell2 ?x ?y ?mark))
    (not gamelover))
(<= (next (control2 xplayer))
    gamelover
    (true (control2 oplayer)))
(<= (next (control2 oplayer))
    gamelover
    (true (control2 xplayer)))
(<= (next (control2 ?player))
    (true (control2 ?player))
    (not gamelover))
(<= (next (gamelooverlock))
    gamelttermcond)
```

```
(<= (next (game1overlock))
    (true (game1overlock)))
(<= terminal
    game2termcond)
(<= (goal xplayer 100)
    goalxplayer1 -100
    goalxplayer2 -100)
(<= (goal xplayer 75)
    goalxplayer1 -100
    goalxplayer2 -50)
(<= (goal xplayer 75)
    goalxplayer1 -50
    goalxplayer2 -100)
(<= (goal xplayer 50)
    goalxplayer1 -100
    goalxplayer2 -0)
(<= (goal xplayer 50)
    goalxplayer1 -50
    goalxplayer2 -50)
(<= (goal xplayer 50)
    goalxplayer1 -0
    goalxplayer2 -100)
(<= (goal xplayer 25)
    goalxplayer1 -50
    goalxplayer2 -0)
(<= (goal xplayer 25)
    goalxplayer1 -0
    goalxplayer2 -50)
(<= (goal xplayer 0)
    goalxplayer1 -0
    goalxplayer2 -0)
(<= (goal oplayer 100)
    goaloplayer1 -100
    goaloplayer2 -100)
(<= (goal oplayer 75)
    goaloplayer1 -100
    goaloplayer2 -50)
(<= (goal oplayer 75)
    goaloplayer1 -50
    goaloplayer2 -100)
(<= (goal oplayer 50)
    goaloplayer1 -100
    goaloplayer2 -0)
(<= (goal oplayer 50)
    goaloplayer1 -50
    goaloplayer2 -50)
(<= (goal oplayer 50)
    goaloplayer1 -0
    goaloplayer2 -100)
(<= (goal oplayer 25)
    goaloplayer1 -50
    goaloplayer2 -0)
(<= (goal oplayer 25)
    goaloplayer1 -0
    goaloplayer2 -50)
(<= (goal oplayer 0)
    goaloplayer1 -0
    goaloplayer2 -0)
(<= game1over
    game1termcond)
(<= game1over
    (true (game1overlock)))
(<= game1termcond
    (line1 x))
(<= game1termcond
    (line1 o))
(<= game1termcond
    (not open1))
(<= game2termcond
    (line2 x))
(<= game2termcond
    (line2 o))
(<= game2termcond
    (not open2))
(<= goalxplayer1 -100
    (line1 x))
(<= goalxplayer1 -50
    (not (line1 x))
    (not (line1 o))
    (not open1))
(<= goalxplayer1 -0
    (line1 o))
(<= goalxplayer1 -0
    (not (line1 x))
    (not (line1 o))
    open1)
(<= goaloplayer1 -100
    (line1 o))
(<= goaloplayer1 -50
    (not (line1 x))
    (not (line1 o))
    (not open1))
(<= goaloplayer1 -0
    (line1 x))
(<= goaloplayer1 -0
    (not (line1 x))
    (not (line1 o))
    open1)
```

```
(<= goalxplayer2-100
    (line2 x))
(<= goalxplayer2-50
    (not (line2 x))
    (not (line2 o))
    (not open2))
(<= goalxplayer2-0
    (line2 o))
(<= goalxplayer2-0
    (not (line2 x))
    (not (line2 o))
    open2)
(<= goaloplayer2-100
    (line2 o))
(<= goaloplayer2-50
    (not (line2 x))
    (not (line2 o))
    (not open2))
(<= goaloplayer2-0
    (line2 x))
(<= goaloplayer2-0
    (not (line2 x))
    (not (line2 o))
    open2)
(<= (row1 ?x ?mark)
    (true (cell1 ?x 1 ?mark))
    (true (cell1 ?x 2 ?mark))
    (true (cell1 ?x 3 ?mark)))
(<= (column1 ?y ?mark)
    (true (cell1 1 ?y ?mark))
    (true (cell1 2 ?y ?mark))
    (true (cell1 3 ?y ?mark)))
(<= (diagonal1 ?mark)
    (true (cell1 1 1 ?mark))
    (true (cell1 2 2 ?mark))
    (true (cell1 3 3 ?mark)))
(<= (diagonal1 ?mark)
    (true (cell1 1 3 ?mark))
    (true (cell1 2 2 ?mark))
    (true (cell1 3 1 ?mark)))
(<= (line1 ?mark)
    (row1 ?x ?mark))
(<= (line1 ?mark)
    (column1 ?y ?mark))
(<= (line1 ?mark)
    (diagonal1 ?mark))
(<= open1
    (true (cell1 ?x ?y b)))
(<= (row2 ?x ?mark)
    (true (cell2 ?x 1 ?mark))
    (true (cell2 ?x 2 ?mark))
    (true (cell2 ?x 3 ?mark)))
(<= (column2 ?y ?mark)
    (true (cell2 1 ?y ?mark))
    (true (cell2 2 ?y ?mark))
    (true (cell2 3 ?y ?mark)))
(<= (diagonal2 ?mark)
    (true (cell2 1 1 ?mark))
    (true (cell2 2 2 ?mark))
    (true (cell2 3 3 ?mark)))
(<= (diagonal2 ?mark)
    (true (cell2 1 3 ?mark))
    (true (cell2 2 2 ?mark))
    (true (cell2 3 1 ?mark)))
(<= (line2 ?mark)
    (row2 ?x ?mark))
(<= (line2 ?mark)
    (column2 ?y ?mark))
(<= (line2 ?mark)
    (diagonal2 ?mark))
(<= open2
    (true (cell2 ?x ?y b)))
(<= (distinctcell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
    (cell ?x2 ?y2)
    (distinct ?x1 ?x2))
(<= (distinctcell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
    (cell ?x2 ?y2)
    (distinct ?y1 ?y2))
(cell 1 1)
(cell 1 2)
(cell 1 3)
(cell 2 1)
(cell 2 2)
(cell 2 3)
(cell 3 1)
(cell 3 2)
(cell 3 3)
```

List of Algorithms

3.1 SubgameDetection 9
4.1 ImpartialCheck 16
4.2 LegalImpartialCheck 16
4.3 NextImpartialCheck 17
4.4 CorrespondencyCheck 17
4.5 PredCheck 18
4.6 SubgameSearchImpartial 21
4.7 GlobalGameSearchImpartial 23
4.8 CalculateBestMove 24
5.1 SubgameSearchIForAlternatingMove 30
5.2 SubgameSearchIIForAlternatingMove 33
5.3 GlobalGameSearchIForAlternatingMove 35
5.4 SelectBestMove 36
5.5 GlobalGameSearchIIForAlternatingMove 40
5.6 SelectBestMoveII_A 41
5.7 SelectBestMoveII_B 42
5.8 ChooseBestCom 43

List of Tables

4.1 Search Results Comparison for Nim with 4 Heaps 26

5.1 Subgame Search Method II Results for Double-Tictactoe 46
5.2 Subgame Search Method II Results for Double-Tictactoe cont. . . . 46
5.3 Search Results Comparison for Double-Tictactoe 46
5.4 Search Results Comparison for Double-Tictactoe cont. 46

List of Figures

4.1 Hackenbush 20

5.1 Path Selection Example 38
5.2 Small Game Tree 39

Bibliography

[Con76] John Horton Conway. *On Numbers and Games*. Number 6 in London Mathematical Society Monographs. Academic Press, 1976.

[Gue07] Martin Guenther. Decomposition of single player games. 2007.

[MG05] Nathaniel Love Michael Genesereth. General Game Playing: Overview of the AAAI competition. 2005.

Declaration

I confirm that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research, and that I have acknowledged all main sources of help and all material from other sources.

Date	Place	Dengji Zhao