University of Alberta

FEATURE LEARNING USING STATE DIFFERENCES

by

MESUT KIRCI

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©MESUT KIRCI
Spring 2010
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

Jonathan Schaeffer, Computing Science

Nathan Sturtevant, Computing Science

Michael Buro, Computing Science

Sean Gouglas, Humanities Computing and History & Classics

Abstract

Domain-independent feature learning is a hard problem. This is reflected by lack of broad research in the area. The goal of General Game Playing (GGP) can be described as designing computer programs that can play a variety of games given only a logical game description. Any learning has to be domain-independent in the GGP framework. Learning algorithms have not been an essential part of all successful GGP programs. This thesis presents a feature learning approach, GIFL, for 2-player, alternating move games using state differences. The algorithm is simple, robust and improves the quality of play. GIFL is implemented in a GGP program, Maligne. The experiments show that GIFL outperforms standard UCT algorithm in nine out of fifteen games and loses performance only in one game.

Acknowledgements

Dr. Jonathan Schaeffer my co-supervisor. His motivation and vision helped me greatly.

Dr. Nathan Sturtevant my co-supervisor. His input and help made this project what it is today.

My family and friends, for their support through the years.

Table of Contents

1 Introduction **1**

2 Background Information **5**
2.1 GGP Competition . . . 5
2.2 UCT . . . 6
2.3 GGP Programs . . . 8
2.3.1 ClunePlayer . . . 8
2.3.2 FluxPlayer . . . 9
2.3.3 CadiaPlayer . . . 10
2.3.4 Ary . . . 11
2.3.5 Comments . . . 11
2.4 Domain-Independent Feature Learning Algorithms . . . 11
2.4.1 Zenith . . . 11
2.4.2 Knowledge Extraction in GGP . . . 12
2.5 Domain-Independent Feature Selection Algorithms . . . 13
2.6 Conclusion . . . 13

3 Feature Learning **14**
3.1 An Example of Feature Learning . . . 14
3.2 Feature Learning . . . 17
3.2.1 Offensive Feature Learning . . . 19
3.2.2 Defensive Feature Learning . . . 20
3.2.3 Backtracking the 2-ply Tree . . . 22
3.2.4 Implementation Details . . . 25
3.3 Using Features . . . 27

4 Implementation Details **30**

5 Experiments **34**
5.1 Learning GIFL Features . . . 35
5.2 Effectiveness of GIFL With Fixed Number of Simulations . . . 37
5.3 Effects of Computation Time Spent By Using Features on the Number of UCT Simulations . . . 40
5.4 Effectiveness of GIFL Over Length of the Simulations . . . 41
5.5 Effectiveness of GIFL With Fixed Time Per Move . . . 42
5.6 Effectiveness of the Opponent Modeling . . . 44
5.7 Effectiveness of Different Feature Matching Methods . . . 46
5.8 Conclusions . . . 47

6 Conclusions and Future Work **48**
6.1 Suggestions for Improvements on GIFL . . . 48
6.2 Future Work . . . 51

Bibliography **54**

A Constants **56**

List of Tables

5.1 Number of features that is learned by GIFL in two minutes. 36
5.2 Effectiveness of using GIFL. 38
5.3 Computational cost of using GIFL. 40
5.4 Average length of simulations. 42
5.5 Effectiveness of using GIFL with 30 seconds per move. 43
5.6 Effectiveness of using GIFL with 30 seconds per move. 45
5.7 Effectiveness of the opponent modeling. 45
5.8 Average number of simulations for 100 moves in connect4. 47

A.1 The constant, C, is used to determine the move values 56
A.2 The constant, τ, is used in the Gibbs Distribution formula 56

List of Figures

2.1 Tic-Tac-Toe game description in GDL from [10]. 7
2.2 The UCT algorithm. 8

3.1 The complete game sequence. 14
3.2 An offensive feature. 15
3.3 A defensive feature. 15
3.4 No feature is learned in this state. 16
3.5 An offensive feature. 16
3.6 A defensive feature. 16
3.7 2-ply game tree at the end of the game sequence. 17
3.8 2-ply game tree and the features that can be learned from that tree. 18
3.9 Finding terminal predicates. (a) is terminal state, (b) is the state after removing the first predicate, (c) removing the second predicate, (d) removing the third predicate, (e) removing fourth the predicate and (f) removing the fifth predicate. 19
3.10 Finding the predicates for an offensive feature. (a) is the terminal state, (b) is the middle state, (c) is the middle state after removing the first predicate, (d) removing the second predicate, (e) removing the third predicate and (f) removing the fourth predicate. 20
3.11 Finding the defensive-feature moves. (a) middle state, (b) root state and (c) are the legal moves at the root state. 21
3.12 Finding defensive-feature predicates using intersection. 22
3.13 Finding the new 2-ply tree for further learning. 23
3.14 Finding the new 2-ply tree for further learning. 24
3.15 The learning algorithm. 26
3.16 The function to find feature predicates. 27
3.17 The algorithm to use the features in the UCT search. 29

4.1 Incrementally constructed state. 31
4.2 State that has same size during the game. 31

5.1 A defensive feature from pawn whopping. 36
5.2 An offensive feature from connect4. 37
5.3 A defensive feature from checkers. 37

6.1 A board position. 49
6.2 An offensive feature. 50
6.3 An example of the extended GIFL feature. 50
6.4 A terminal state. 52
6.5 Generated terminal states. 52
6.6 A number of GIFL features. 52
6.7 An abstract new GIFL feature. 53

List of Abbreviations

UCT	Upper Confidence Bounds Applied to Trees
GIFL	Game Independent Feature Learning
GGP	General Game Playing
AI	Artificial Intelligence
GDL	Game Definition Language
UCB	Upper Confidence Bounds
TD	Temporal Difference

Chapter 1

Introduction

Playing games that involve strategies improves and exercises intellectual skills. A similar motivation leads researchers to use games as a testbed for Artificial Intelligence (AI) research. However, researchers have focused on the techniques for playing specific games very well. This work gave rise to programs that can play a game at a world-class level like Deep Blue [2], Chinook [19] and TD-Gammon [24]. Deep Blue beat the world chess champion in 1997, Chinook became world Checkers champion in 1994, and TD-Gammon is at world-champion level in Backgammon. However, these programs cannot play other games at all. More importantly, most of the game analysis and programming design is done by the programmer. Thus, these program solutions have limited value for gaining insights into generally-applicable AI [16].

General Game Playing (GGP), where programs aim to play more than one type of game, is used as a testbed for Artificial Intelligence research because it is perceived that it requires more general intelligence skills [9]. A general game player accepts a game description as input at runtime, analyzes it, and then plays the game without human intervention. Therefore, a general game player cannot use algorithms specific to a particular game and must rely on the intelligence of the program rather than the knowledge of the programmer. Also, general game players should be able to play different classes of games which may have a variable number of players, simultaneous or alternating moves, or significantly different rules (*e.g.,* board game and card game) [9].

General game playing was first discussed by Pell [16]. He offered a concept, Metagame, to address the issue of evaluation of the AI research in current game specific programs. Metagame is a class of games that are defined in a description language and Metagamer is able to play all games defined in Metagame. His definition of Metagame is restricted to chess-like games. Pell used an abstract experiment which he called "The Gamer's Dilemma" to show the importance of Metagame:

> Suppose that a researcher is informed that he will soon be given the rules of a game, G, played by a group of humans and/or programs. They all are considered to play G at a high performance level. The researcher is given a fixed amount of time to produce a program which will compete against random members of the group (the researcher is

> not be allowed to communicate with them beforehand). Finally, the researcher will be paid based on the number of games the program managed to win (or draw) out of some prespecified total amount of games. The researcher's goal is, of course, to maximise the money he expects to receive from his program's play. [16]

Until recently AI researchers chose to design game-specific programs, but this design approach has disadvantages. First, much human effort is needed every time a program for a different game needs to be developed. Second, it is difficult to evaluate the research because the success of the program may demonstrate how well the researcher analyzed the game, not necessarily how intelligent the program is. Third, a solution to a specific game may not lead to insights for other games and even AI in general.

Current GGP programs rely heavily on search algorithms, specifically UCT [11] and alpha-beta search. Programs that use UCT do not need to evaluate arbitrary states in a game tree because they simulate a game until a terminal node (end of game) is reached and use the value of the terminal node as an evaluation (often just win, lose, or draw). This search-intensive approach is suitable for GGP, since knowledge-based approaches have not yielded much in the way of performance. Results of the annual GGP competition also reflects this fact. Cadia Player [7] is an example of a program that uses UCT; it has won two of the last three GGP competitions. Cadia Player uses simple heuristics to guide the UCT simulation. The other approach is alpha-beta pruning with an evaluation function. FluxPlayer [20] and Clune Player [5] are examples of this type of approach. Both players create an evaluation function to guide the search. FluxPlayer's evaluation function calculates the degree of truth using fuzzy logic to evaluate leaf and goal states. Clune player, along with UTexas player [13], uses automatically extracted features to calculate the evaluation function. These are simple features, such as the number of pieces on the board and the number of legal moves. Clune Player was the winner of the first GGP competition in 2005 before UCT was introduced to the GGP community.

In addition to the approaches mentioned above, learning methods have been suggested by Sharma *et al.* from University of Windsor [21]. One approach is to use Temporal Difference learning [23]. It learns a domain-independent knowledge base and uses the knowledge base to guide the UCT search. This technique has not been tested in the competitions and the experimental results have shown only slight improvements for some games.

The success of all programs mentioned can be increased by improving the search using knowledge about a game. However, GGP programs must extract information automatically and domain-independent knowledge extraction is a very hard problem. The results of the last two competitions reflect this issue. UCT, which is less dependent on knowledge, has been very successful and the quality of play of GGP programs depends on how fast they are with the UCT algorithm. These searches could be more effective with the same search speed if the GGP program extracted useful knowledge about the domain.

The main contribution of this thesis is that domain-independent learning can be done and em-

ployed to improve the quality of play of a GGP program. To prove the statement;

- a simple learning algorithm is proposed,
- it is implemented into one of the successful GGP programs, Maligne, and
- it is shown that the quality of play has increased when using the learned information.

The approach described in the thesis is called Game Independent Feature Learning, *GIFL*. GIFL learns information and uses it to do more intelligent search in two player, general sum, alternating move games. It learns features, similar to the well-known history heuristic [18], using state differences in 2-ply (one move by each player) game trees. After that, learned features are used to guide the random move selection in a UCT simulation. In short, the algorithm has two parts:

- learning the features, and
- using the features in UCT search.

In the AI literature, a *feature* is defined as a subset of a state instantiated with values. The term feature is used differently in this thesis. It is a knowledge chunk that consists of a state subset and a set of moves. Therefore, the features of GIFL are similar to regular features and include moves. In this thesis, the term feature represents the GIFL features from this point on.

Features are learned from the differences of the game states in 2-ply game trees. In GGP, a state consists of predicates which are the facts that are present. The predicates that are required for a state to be a goal state are referred as *terminal predicates* and all predicates in a state are referred to as *state predicates* throughout the thesis.

GIFL identifies the terminal predicates first. Starting from the last move, moves that add terminal predicates to a state are combined with the predicates that help the state to get close to the goal condition. This combination is an *offensive feature*. For each offensive feature, a combination of predicates and moves are learned as a *defensive feature* aiming to prevent the opponent from using the offensive features.

After features are learned, they are used in guiding the random playout process of UCT. During the random playout, when a feature's predicates are present in a state and the moves associated with the feature are legal, a value is assigned to that move. After finding all features that are true in a state, a move is selected using Gibbs Distribution [3]. Intuitively, the higher the value given to a move, the higher the chance it will be selected.

The algorithm has been tested on fifteen different games and has performed well in most of them. The algorithm outperforms the standard UCT completely in nine games, does not effect the results in three games, loses by a slight margin to UCT in two games and loses badly to UCT in only one game.

The thesis is structured as follows:

Chapter 2 gives information about the General Game Playing domain and the annual GGP competition. Also, the programs that have won the competition and two domain-independent learning algorithms are briefly explained. Chapter 3 describes GIFL and how it can be applied to GGP. Chapter 4 focuses on implementation issues that are encountered when GIFL is implemented in the GGP player Maligne. The experiments results are shown in Chapter 5. Chapter 6 presents the conclusions and the future work.

Chapter 2

Background Information

General Game Playing is the domain in which Game Independent Feature Learning (GIFL) is tested. Thus, a certain amount of knowledge about General Game Playing makes it easier to understand how the algorithm works. In addition, several algorithms related to GGP and domain-independent knowledge extraction are briefly explained to demonstrate the state of the art.

2.1 GGP Competition

There is a General Game Playing Competition that has been held annually since 2005. State-of-the-art GGP programs are tested under the competition settings. Maligne, the GGP player that uses GIFL, also follows the competition format while playing games.

At the start of a GGP game, a process connects to each player and sends the game description along with the time limits for the game. Players have the duration of the start-clock (typically 5 seconds to 1 minute) to analyze the game description before the game begins and they get the duration of the play-clock (typically 10 seconds to 30 seconds) to make each move. The general game player must be a complete and fully autonomous agent because no human intervention is allowed during the game [13]. The game description is in a format called GDL which was developed by the Stanford Logic Group [10].

The Game Description Language (GDL) is an axiomatic language which can be used to represent finite and deterministic games with complete information. All information about the game is encoded into GDL and no prior knowledge is assumed. GDL is a variant of Datalog which is a query and rule language similar to Prolog, and it uses first order logic. In addition to the general logic functions, the language is based on a set of keywords to describe game states, legal moves and terminal conditions. Some of the keywords are explained below and the sample lines from the GDL description of tictactoe is presented in Figure 2.1.

- **role:** describes the players. *e.g.,* **(role xplayer)** : xplayer is the name of one of the players.
- **init:** describes the initial state of the game. *e.g.,* **(init (cell 1 1 b))** : the position located at x-coordinate 1 and y-coordinate 1 is a “b” (blank) at the beginning of the game.

- **legal:** describes legal moves for a player. *e.g.,* **(legal oplayer action)** : action is legal for oplayer.
- **does:** describes the move made by the specified player. *e.g.,* **(does xplayer b)** : informs players that xplayer made move b in his turn.
- **terminal:** describes the end of the game. *e.g.,* **(terminal diagonal)** : diagonal is a predefined condition for tictactoe, where the game ends.
- **goal:** describes the reward a player gets if the state is a terminal state. *e.g.,* **((goal oplayer 100) line)** : line is a terminal condition for Tic-Tac-Toe and if it is present in the state then oplayer gets a reward of 100.
- **true** and **next:** describe the conditions that are true in the current state and in the next state respectively. *e.g.,* **(true (cell 1 3 X))** : the position located at x-coordinate 1 and y-coordinate 3 contains mark X in this state.

This is a simple set of keywords that allows for a rich variety of games to be defined. These games are one, two, or multiple player games of perfect information. Imperfect information and stochastic games are currently not supported in GDL.

2.2 UCT

The UCT algorithm has become popular with GGP programs because it is effective when an evaluation function is hard to come up with. Go is the primary example of a domain where UCT is successful [8].

UCT, UCB applied to trees, is a variation of the Upper Confidence Bounds (UCB1) algorithm [11]. UCB1 provides a solution to the exploration-exploitation tradeoff. UCT applies UCB to trees by keeping track of the average returns of all states and sampling the one with the highest upper confidence bound. The formula for calculating the upper confidence bound in UCT is given as:

$$a_t = argmax_{a \in A_t} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}$$

where a_t represents the action taken at time t and $Q(s,a)$ gives the average return for the next states after action a is taken. In addition, $N(s)$ returns the number of visits to a state and $N(s,a)$ is the number of times action a has been explored in a state. The left part of the formula ($Q(s,a)$) is the estimated value of a state and the right part of the formula ($C\sqrt{\frac{\ln N(s)}{N(s,a)}}$) is called the UCT bonus. The formula matches the UCB formula when the parameter C is $\frac{1}{\sqrt{2}}$, but the UCT algorithm allows the parameter to be changed to increase or decrease exploration.

If a state has one or more unexplored actions, they are first explored at least once before the UCT rule is used. However, in the general case UCT selects the action that gives the highest average estimated value plus the UCT bonus. States build up bonus when they are not visited ($N(s)$ increases)

```
(role xplayer)
(role oplayer)
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= (next (cell ?m ?n o))
    (does oplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
...
(<= (diagonal ?x))
    (true (cell 1 1 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 3 ?x)))
(<= (diagonal ?x)
    (true (cell 1 3 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 1 ?x)))
(<= (line ?x)
    (row ?m ?x))
...
(<= (goal xplayer 100)
    (line x))
(<= (goal xplayer 50)
    (not (line x))
    (not (line o))
    (not open))
...
(<= terminal
    (line x))
(<= terminal
    (line o))
(<= terminal
    (not open))
```

Figure 2.1: Tic-Tac-Toe game description in GDL from [10].

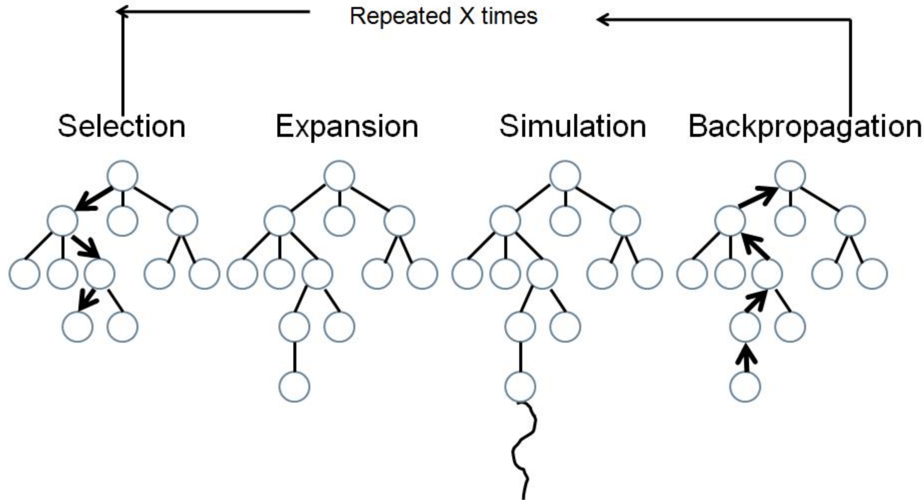

Figure 2.2: The UCT algorithm.

and their bonus drops after each visit (($N(s, a)$ increases)). States that do not look promising are explored when their bonus gets high enough to compensate for their lack of average estimated values. This ensures that all states are explored even though they do not have high average returns in the initial visits.

After a state is selected using the UCT rule to be explored, a sequence of moves is randomly made from that state until a terminal state is reached. This state has a value (win, loss or draw, but it could be any numerical value). The value of the terminal state is propagated up to the root of the UCT tree and the values along the path are updated. The selected state is added to the UCT tree. This cycle is repeated until the whole tree is explored or the algorithm is stopped. Figure 2.2 shows this.

The estimated values for the possible moves are used as evaluation function by the game programs including GGP programs. The move with the highest estimated value is selected when a move selection is required. Accuracy of the value estimations depends on the allocated time for the simulations.

2.3 GGP Programs

There are three published programs that have been successful in the past GGP competitions. They are described briefly in the order that they won the GGP competition.

2.3.1 ClunePlayer

ClunePlayer was the winner of the first GGP competition in 2005. ClunePlayer simplifies a game to the core aspects and uses the result of this simplified game as a heuristic value for the original game [5]. The simplified game retains the core aspects of the game, such as expected payoff,

mobility (*e.g.,* pieces) and expected game termination (estimation of the length of the game).

ClunePlayer analyzes the game description and extracts expressions which are part of a state, such as pieces, legal moves or position of the pieces. Then, the extracted expressions are assigned values according to different interpretations such as how many different solutions an expression has or how many steps it takes to go from one expression to another.

In addition, ClunePlayer calculates how close a state is to a terminal state for games with compound expressions involving multiple conjucts or disjuncts. The number of satisfied conjuncts or disjuncts in a formula over the total number of conjuncts or disjuncts in the formula gives an estimate for satisfying the formula. This is applied to the terminal conditions with compound expressions.

The expressions with values are the features of the evaluation function that ClunePlayer uses in the traditional alpha-beta search.

After the discovery process, the features of ClunePlayer are tested for stability. If the value of a feature does not vary drastically from one state to another, the feature is considered stable and used in the evaluation function.

The stable features that ClunePlayer extracts are combined using a weighted formula in which the weight is proportional to the features' stability. These weighted features form the evaluation function. As the game gets close to termination, the payoff features, which represent the value which a player gets, are favored more by the evaluation function. This evaluation function is used in both single and multi-player games.

2.3.2 FluxPlayer

FluxPlayer was the winner of the second GGP competition in 2006. FluxPlayer uses an automated theorem prover based on the Flux System to derive legal moves, simulate game play and determine terminal states [20]. The game description is changed to Prolog for efficient theorem proving and the Flux System is used for reasoning. The Flux System is based on Fluent Calculus. It uses an explicit state representation and calculates the next state after taking an action using axioms in the description. Positions correspond to the states in Fluent Calculus and fluents describe the features of the game [25].

The search algorithm used by FluxPlayer is iterative deepening with two important enhancements; transposition tables to cache the value of the states seen previously and move ordering that is based on the previous values found in lower depth searches. This method is used in both simultaneous and turn taking games. The paranoid approach [22] is taken to tackle the problem of not seeing the opponents' moves in simultaneous games. FluxPlayer always makes the move first and the other players are serialized to make moves after. Therefore, all other players know the agent's move.

FluxPlayer calculates the degree of truth for a state by estimating how close a state is to terminal and goal states. The standard t-norm and t-co-norm of Fuzzy Logic are used in this calculation, where a state gets a value between 0 and 1 depending on how close it is to a goal completion. After

the degree of truth is calculated in the state, the heuristic tries to avoid states if the goal condition is not satisfied. Otherwise, heuristics try to guide the search to a terminal state.

2.3.3 CadiaPlayer

CadiaPlayer was the winner of the 2007 and 2008 GGP competitions [7]. CadiaPlayer converts the game description into Prolog code. The generated code is compiled into a Prolog library to handle game specific operations such as determining legal moves, finding terminal states and manipulating states.

CadiaPlayer uses iterative deepening A^* (IDA^*) [12] in single player games during the start-clock. There is no heuristic function in the algorithm. If a solution is found (even if a non-optimal), IDA^* continues to look for better solutions during play-clock. Otherwise, CadiaPlayer uses the UCT algorithm on the play-clock.

The UCT algorithm handles all multi-player games. In two-player games, CadiaPlayer minimizes the score difference between players or maximizes its own score depending on the decision made by the programmer before the game. CadiaPlayer maximizes its own score in games with more than two players.

There are several control heuristics that CadiaPlayer extracts to bias the search towards more promising actions during a search [7]. These heuristics are all automatically extracted and domain-independent. They cannot be used at the same time, but a combination of two (the first and the fourth heuristics in the following discussion) was used in the official matches during the 2009 GGP competition.

The first heuristic is a variant of the history heuristic. CadiaPlayer keeps track of average values for moves regardless of which state that the move is made. Assuming that a good move is usually good regardless of which stage of the game, moves are selected proportional to the values they have.

The second heuristic is very similar to the first one. The only difference is that average values are kept only for the moves that are in the UCT tree. Therefore, the moves made during the random simulation are not used.

The third heuristic considers also the context in which the move is made. The move and each statement that is true in the state are paired and an average value is kept for the pair. During the move selection, the value for a state is calculated using the values for each fact in the state.

The fourth heuristic is designed to get more information from a simulation especially when the number of simulations is not high. The average value of a state in the UCT tree is updated with the simulation value as usual, but sibling states that can be reached with an action *a* which is seen during the simulation are updated as well. This heuristic is used more at the initial stage of the game where the number of simulations per state is low.

2.3.4 Ary

Ary is the winner of the 2009 GGP competition and designed by Jean Méhat from Paris University. There is no published work about how the program works, but the author described the program during the 2009 GGP competition. Ary uses the UCT algorithm without any knowledge discovery. The quality of play solely depends on the speed of the program.

2.3.5 Comments

The above discussion shows that the programs are quite simple. Most rely on search, using minimal knowledge. The result should not be suprising: all these programs play a game at a weak level as compared to human performance.

2.4 Domain-Independent Feature Learning Algorithms

There have been few attempts to create algorithms that extract features automatically. Even fewer of them can be considered as a success. The algorithm designed by Fawcett is the most promising and is described in this section. It is almost 20 years old, and the fact that there has not been a significant advance on this work shows that little progress has been made on feature discovery. Also, an approach that is most similar to the research presented in this thesis is explained.

2.4.1 Zenith

Zenith, Fawcett's system, generates an evaluation function from the description of a problem in an iterative fashion [6]. It has three main steps: creating new training instances using the evaluation function, extracting new features from existing ones and selecting the features that will be in the evaluation function.

Zenith creates features from existing ones using four different transformations:

- Decomposition. A decomposition transformation recognizes a syntactic form, such as an arithmetic inequality like $A < B$, that can be decomposed into new features, A and B. This transformation provides features that may differentiate between neighboring states more accurately.

- Goal regression. Goal regression investigates the relations between features and operators. A new feature is created for each operator that effects the feature. One downside of this transformation is that the domain description has to have properties to support goal regression. GDL does not have this support, thus goal regression cannot be done in the GGP context.

- Abstraction. Abstraction generalizes a feature by removing some details. Abstraction provides features that are less expensive to evaluate and may be close to the original feature in the accuracy of information, yet are more general in their usage.

- Specialization. Specialization instantiates a subset of a feature. Hence it is not as general, but it is easier to evaluate the features that are created.

Zenith uses these four transformations in a specific manner. Decomposition is applied to every feature, abstraction and specialization are only applied to the features that take more than 10% of the total evaluation time, and goal regression is applied to the features that are used in the evaluation function. Features that take long time to evaluate are not used in the evaluation function, but they are used in the transformations because more efficient features may be discovered by applying transformations.

Fawcett has shown that Zenith, using only a game description, can create a number of features that are similar to the hand-designed features in the game of Othello. However, it has to be said that the description Zenith uses for information was created by Fawcett and contains more information than GDL, including the goal regression formulas. Therefore, Zenith transformations cannot be fully applied to GDL without modification and it can be concluded that Zenith cannot achieve similar success in GGP.

2.4.2 Knowledge Extraction in GGP

There is a recent paper about using domain-independent knowledge extraction to guide the UCT simulation [21]. The algorithm calculates a value for a state from the facts that are true in that state. It calculates a value for the moves similar to the history heuristic. These values are used when selecting moves during the UCT simulation phase.

The value of a state is calculated as a sum of the fact values. Then, the sum is squashed to the range 0-1 to approximate the winning percentages. Each fact value is updated after a random simulation using an update method that is very similar to the TD(0) temporal difference learning update:

$$Value(fact) = Value(fact) + LearningRate * \left(\frac{OutcomeOfSimulation - ValueOfTheState}{NumberOfFacts}\right)$$

The value of a move is also updated after a random simulation using the outcome of the simulation and the depth of the move with respect to the length of the simulation.

The value of the move and the value of the state that the move leads to are added together and used to guide the random simulation to bias the search towards potential high valued states. To balance the exploration and exploitation, the ϵ-greedy [23] approach is taken.

This algorithm was tested on three games from the GGP domain: checkers, breakthrough and connect4. The results are promising for the three mentioned games as can be seen from the percentage of wins against a non-learning player: 57 percent in checkers, 68 percent in breakthrough and 59 percent in connect4. No other experiments were reported and it is unclear how generally applicable the idea is.

2.5 Domain-Independent Feature Selection Algorithms

GIFL extracts features from domain description and all of the features are used during gameplay. However, there are several methods in literature involving how to use the given features. Some of these algorithms find functions to combine features, whereas some of them creates new features using conjuncts, disjuncts and other methods. ELF [26], GLEM [1], MORPH [14], RT algorithm [4], the works by Miwa *et al.* [15] and Samuel [17] are examples of such methods.

2.6 Conclusion

It is still early days for high performance GGP programs. Knowledge extraction is difficult and there has been little in the way of successes to report. Hence GGP program developers have moved towards using search algorithms that require minimal knowledge. The state of the art is UCT search with terminal state values. Clearly there is a long way to go before GGP programs will achieve a high level of performance.

Chapter 3

Feature Learning

This chapter describes what GIFL does and how it works. The algorithm does not use domain-specific information. GGP is a suitable domain to test the effectiveness of the algorithm. GIFL is explained using both GGP and domain-specific examples.

3.1 An Example of Feature Learning

Before going into details of the algorithm, what GIFL learns is explained in the following example from tictactoe. Tictactoe is a two player game that is played on a 3x3 grid. Each player marks one location at each turn. The first player who succeeds in placing three of their pieces in a horizontal, vertical or diagonal row wins the game.

GIFL learns from game sequences. Figure 3.1 is an example of a game sequence where the player with the mark **O**, player-O, wins the game. The game states and the locations of the marks from Figure 3.1 will be used throughout this example. For instance, *a-5* represents the mark at location five of the state Figure 3.1-a.

GIFL looks for state-move combinations which help a player win the game. The most obvious state-move combination is shown in *h* and the mark placed at *h-7*. This move leads to Figure *i* and

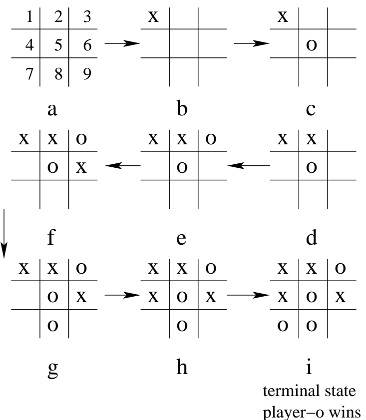

Figure 3.1: The complete game sequence.

player-O wins the game. The move *h-7* can be learned by GIFL and whenever Figure *h* is seen in a game, the mark *h-7* can be placed. However, only a part of the state shown in *h* helps player-O to win the game. Most of the marks in the state, except *h-3* and *h-5* , do not effect the outcome. Thus, only a subset of state *h* is necessary for the move *h-7* to help win the game. This is shown in Figure 3.2. As a result, GIFL learns a feature which advises player-O to mark the location *7* when the locations *3* and *5* are marked with **O**. The complete feature can be seen in Figure 3.2 (circled). This feature is called *offensive feature* because it helps a player to win.

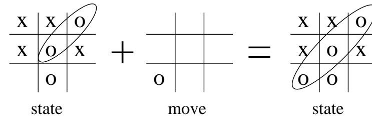

Figure 3.2: An offensive feature.

Since the algorithm learns state-move combinations, every move is inspected for a possible feature. The algorithm learns a feature from *h* and backtracks to try and learn a feature that leads to *h*. Player-X makes the move in this state. If we look at the game sequence, player-O uses the recently learned feature to win the game at *h*. With hindsight, knowing the outcome of the game, player-X will prevent player-O from using the feature and winning the game.

Player-X needs to make player-O's offensive features inapplicable. There are two ways to accomplish that. The first one is to make the state unmatchable for the offensive feature since the player-O looks for certain marks in a state to decide to use the feature. This is not possible in tictactoe because the two marks that are necessary for the player-O to use the offensive feature are present in the state regardless of the move made by player-X. The second one is to make an offensive-feature move illegal. Player-X can achieve this by making the move to *7*, as shown in Figure 3.3. If player-X can satisfy one of the above two conditions, preventing player-O from winning, then the algorithm learns a defensive feature. The defensive feature is also a state-move combination. The defensive-feature state consists of the marks that allow player-O to use a offensive feature in the next state, if not prevented. The defensive-feature move is the one that prevents player-O from using an offensive feature. Figure 3.3 is an example of a defensive feature which is learned from the state in *g*.

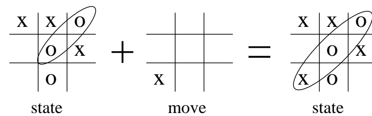

Figure 3.3: A defensive feature.

After learning a defensive feature from *g*, the algorithm continues to look for additional features. GIFL bactracks in the game to see what can be learned for player-O in *f*. The mark placed at *8* does not help player-O to use previously learned offensive features in the future states and win the game. Therefore, the move is not useful and player-O does not learn any features. Figure 3.4 shows that

the move made in Figure 3.1-f does not help to create winning conditions.

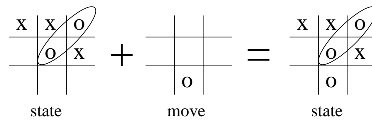

Figure 3.4: No feature is learned in this state.

The algorithm discards any information from Figure *f* and bactracks to *e*. It is player-X's turn again. Since an offensive feature was not learned at *f*, there cannot be a defensive feature at *e*. Thus, GIFL bactracks to *d*.

At the state *d*, player-O makes a move. Therefore, the algorithm looks for a feature that contributes to the winning conditions. As it can be seen in Figure 3.5, the move helps player-O to win the game and is labeled as an offensive-feature move. At state *d*, player-O cannot win the game immediately, but the next state, *e*, includes the marks that allow previously learned offensive features to be used. Thus, the move helps player-O to win the game. The state part of the offensive feature consists of the marks which contribute to the goal conditions or to the conditions of previously learned offensive features with the assistance of the offensive-feature move.

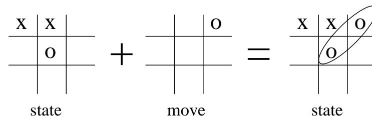

Figure 3.5: An offensive feature.

Since another offensive feature is found, the algorithm looks for a defensive feature in *c*. The algorithm tries to satisfy one of the two conditions that are necessary to make an offensive-feature inapplicable at *d*. The first condition cannot be satisfied because a mark cannot be erased in tictactoe. However, the second condition can be satisfied with the move shown in Figure 3.6. The offensive-feature move will be illegal and the offensive feature will be inapplicable. Therefore, a defensive feature shown in Figure 3.6 is learned by the algorithm. The state at which the defensive-feature move is learned, has the mark that player-O needs further in the game to win and player-X prevents player-O from using it.

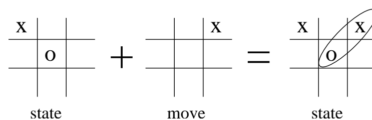

Figure 3.6: A defensive feature.

At Figure *b* and *a*, the algorithm does not learn because of the stopping conditions. The algorithm stops because of two reasons:

- There are no partial goal (winning) conditions present in the state, and

- The algorithm backtracks the game sequence to the initial state.

In this example, the algorithm stops learning because of the first condition, there are no goal conditions remain in Figure 3.1-b.

3.2 Feature Learning

In general, GIFL works by analyzing a 2-ply game tree starting from the terminal state of a randomly generated game sequence and moving backwards towards the start state. GIFL first learns features from a 2-ply tree where the last move is made by the player that won the game. An example 2-ply tree can be seen in Figure 3.7. This tree has a *root state* where the losing side, player 1, makes a move, a *middle state* where the winning side, player 2, makes a move, and several *leaf states*, one of which is terminal (terminal states are shown as boxes and non-terminal states are shown as circles in Figure 3.7). The algorithm can learn two types of features, one from the middle state and one from the root state.

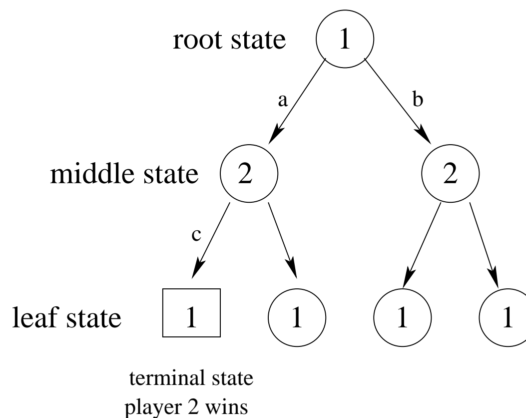

Figure 3.7: 2-ply game tree at the end of the game sequence.

The move made in the middle state, Figure 3.7-c, can be considered a good move for player 2 because it leads to a win. The algorithm learns this move as an *offensive-feature move*. However, the offensive-feature move does not always lead to a win in every state. There are some *state facts*, conditions that are required to be present to lead to a win when the move is applied. These facts in a state are called *predicates*. Required predicates are called *offensive-feature predicates*. Offensive-feature predicates and offensive-feature moves are combined to create a general offensive feature.

In addition to an offensive feature, a defensive feature can also be learned from the same 2-ply game tree. The algorithm assumes that the losing player made a move at the root state, Figure 3.7-a, that allowed the winning player to use the offensive feature to win the game. However, there may be other legal moves at the root state, such as the move leading to Figure 3.7-b. A move that can prevent the opponent from using the offensive feature and winning the game is also considered a good move and learned as a *defensive-feature move*. As an offensive-feature move cannot be used at every

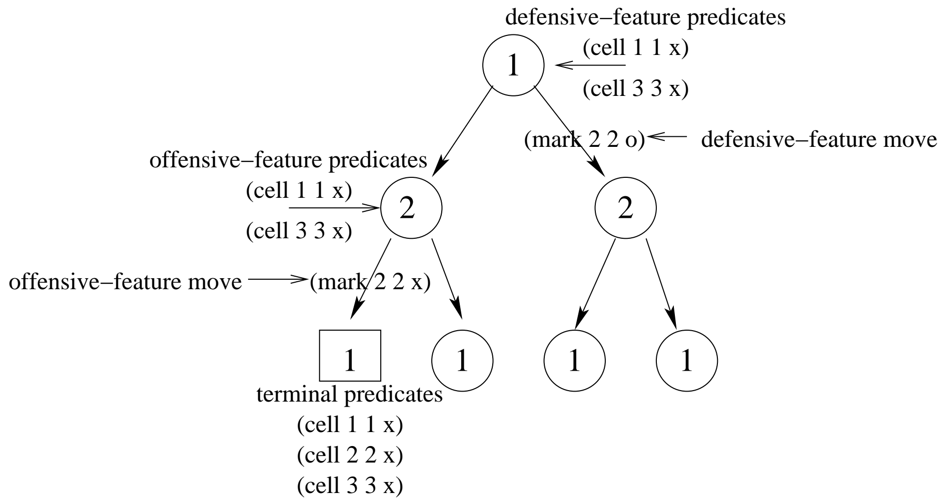

Figure 3.8: 2-ply game tree and the features that can be learned from that tree.

state, the defensive-feature move will not always prevent a loss at every state (*i.e.*,, it is heuristic information - generally useful, but not always true). There is a minimum set of additional state predicates which make the defensive feature necessary to immediately avoid a loss. These predicates are called *defensive-feature predicates*. Defensive-feature predicates and defensive-feature moves are combined to create a general defensive feature.

An example of a 2-ply game tree where an offensive feature and a defensive feature is learned is presented in Figure 3.8. The example is from tictactoe. The tictactoe game is used in all of the examples in this paper. The predicates with a "cell" relation are state predicates and show the state of the game. The first two arguments of the cell relation are the coordinates of the mark and the third argument is the type of the mark located. The "mark" relation represents the moves. Also, the first two arguments of the mark relation are the coordinates of the mark that is to be placed and the third argument is the type of the mark. The features shown consist of predicates and moves.

The algorithm learns state predicates rather than the state itself. This increases the generality of the feature. For instance, the terminal state in Figure 3.8 is unique, but the terminal predicates that make up the state terminal are not. There are different states that have the same terminal predicates. Therefore, the algorithm finds the terminal predicates of the state and uses them in place of the terminal state.

The algorithm finds the terminal predicates by removing the state predicates at the terminal state one by one and checking whether the state is still terminal or not. If removing a predicate does not change the status of the state being terminal, the predicate does not belong to the terminal predicates list. Otherwise the predicate is added to the terminal predicates. In Figure 3.9-b, after we remove the first predicate of (Figure 3.9-a), the state is not terminal. Therefore, the predicate (cell 1 1 x) is a terminal predicate. In Figure 3.9-c, the state is still terminal and (cell 2 1 o) is not a terminal predicate. In the end, terminal predicates are (cell 1 1 x), (cell 2 2 x) and (cell 3 3 x).

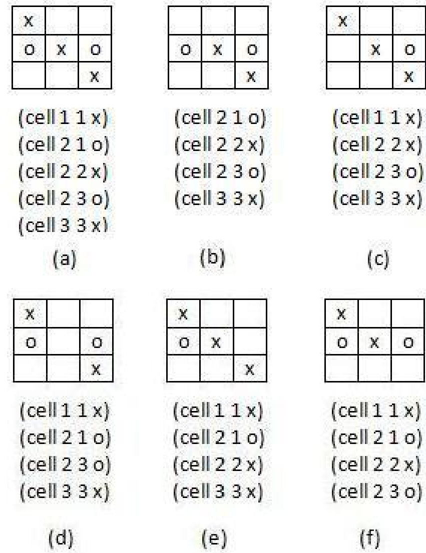

Figure 3.9: Finding terminal predicates. (a) is terminal state, (b) is the state after removing the first predicate, (c) removing the second predicate, (d) removing the third predicate, (e) removing fourth the predicate and (f) removing the fifth predicate.

3.2.1 Offensive Feature Learning

After the terminal predicates are found, the learner focuses on the last 2-ply of the game sequence to discover features. GIFL learns from 2-ply trees. The terminal state is the leaf state of the first 2-ply tree that the algorithm investigates. The leaf state has two conditions: the *leaf predicates* and the *leaf moves*. The aim of the offensive feature is to satisfy *leaf conditions* (make leaf predicates true and a leaf move legal at the leaf state). The leaf predicates for the terminal state are the terminal predicates and there are no leaf moves for the terminal state. If the player who made the move at the middle state won the game, an offensive feature is learned from the 2-ply game tree under examination because the leaf predicates are true in the terminal state and there are no leaf moves. The move which led to a win (and the satisfaction of the leaf conditions) is considered good and is part of an offensive feature.

A feature consists of two parts: moves and predicates. The offensive-feature predicates are required predicates in the middle state to satisfy the leaf conditions after the offensive-feature move is made. To find the offensive-feature predicates, the algorithm removes each of the middle-state predicates one by one and applies the offensive-feature move to the reduced middle state. If the leaf conditions are not satisfied in the resulting leaf state, the removed predicate from the middle state is necessary for the offensive feature to be applied successfully and is a part of the offensive feature. The offensive-feature predicates found are paired with the offensive-feature move to become an offensive feature. In Figure 3.10, the move (mark 2 2 x) is the offensive-feature move that makes the leaf conditions true. There are no leaf actions in this case and the leaf predicates are the terminal

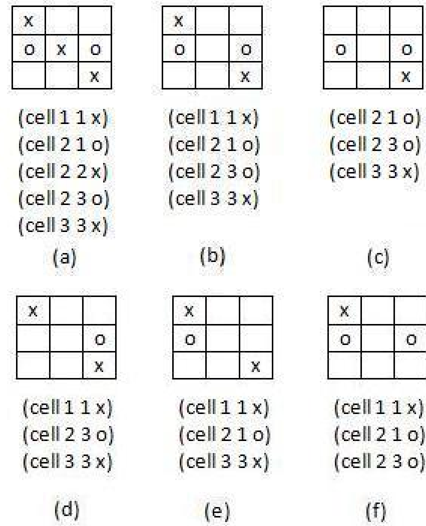

Figure 3.10: Finding the predicates for an offensive feature. (a) is the terminal state, (b) is the middle state, (c) is the middle state after removing the first predicate, (d) removing the second predicate, (e) removing the third predicate and (f) removing the fourth predicate.

predicates (cell 1 1 x), (cell 2 2 x) and (cell 3 3 x). In Figure 3.10-c, the offensive-feature move is legal, but the leaf conditions are not satisfied after the move is applied to the state. Therefore, (cell 1 1 x) is an offensive-feature predicate. However in Figure 3.10-d, the leaf conditions are satisfied after the offensive-feature move is applied. Therefore, (cell 2 1 o) is not an offensive-feature predicate. In the end, the predicates (cell 1 1 x) and (cell 3 3 x) are found to be offensive-feature predicates along with the offensive-feature move (mark 2 2 x).

3.2.2 Defensive Feature Learning

The second type of feature that GIFL looks for is defensive features. A defensive feature tries to prevent the opponent from reaching a state at which an offensive feature can be applied. In the 2-ply tree under examination, the offensive feature that the defensive feature tries to make useless is the one learned from the middle state. To accomplish this, the defensive feature either makes the offensive-feature move illegal or makes the offensive-feature predicates false in the middle state.

A defensive feature also consists of two parts: predicates and moves. First, the algorithm looks if there are possible moves that can be counted as defensive-feature moves. Regardless of which predicates are the defensive-feature predicates, the defensive feature has to make the offensive feature useless in the middle state. Second, if there are any defensive-feature moves, then defensive-feature predicates are looked for.

The algorithm tries all legal moves at the root of the 2-ply tree that is under investigation. If the offensive feature learned at the middle state cannot be used at the resulting middle state after making a move, that move is considered as a possible defensive-feature move. However, if no possible

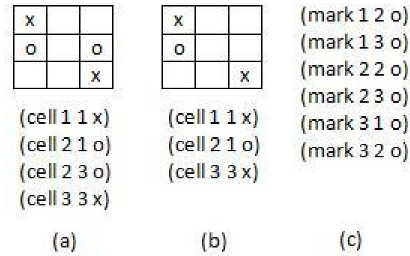

Figure 3.11: Finding the defensive-feature moves. (a) middle state, (b) root state and (c) are the legal moves at the root state.

defensive-feature moves can be found at the present root state, the algorithm backtracks two ply in the game tree leaving the leaf state and the leaf conditions unchanged. The game sequence that GIFL learns from is created by random simulation, therefore some of the moves made by the players may not be related to the terminal predicates and can be considered unimportant. For instance, in the game of breakthrough, the goal condition is related to only one predicate. However, there are over 10 moves on average at any step. Therefore, some moves may not affect the outcome of the game. The learner backtracks the 2-ply tree to find a defensive-feature move until either the offensive feature learned cannot be applied to the middle state (in which case the learning stops due to the lack of a defensive feature) or possible defensive-feature moves are found. To make the offensive feature useless at the middle state, either the offensive-feature move should be illegal or some of the offensive-feature predicates must be made false. The offensive-feature predicates are present at the root state and there is no possibility of making them false at the middle state. Therefore, the defensive-feature move should make the offensive-feature move illegal. The legal moves are listed at Figure 3.11-c. If all moves are applied one by one, it can be seen that the move (mark 2 2 o) is the only one that makes the offensive-feature move illegal at the middle state. Therefore the move (mark 2 2 o) is the defensive-feature move.

After finding some possible defensive-feature moves, the algorithm finds the defensive-feature predicates. Defensive-feature predicates require the player to use a defensive feature. Therefore, defensive-feature predicates are required by the opponent to use the offensive feature at the middle state. Only moves that prevent the opponent from doing that are the possible defensive-feature moves. Therefore, the algorithm finds a set of defensive-feature predicates for each legal move except the possible defensive-feature moves in the root state. This process is the same as finding the offensive-feature predicates. A set of conditions must be satisfied at the next state after making a move. The conditions are leaf predicates and leaf moves when the learning is about offensive-feature predicates. The conditions are predicates and moves of the offensive feature when the learning is about a defensive feature. All legal moves except defensive-feature moves should allow the offensive feature to be applied.

After a set of defensive-feature predicates are found for each legal move (except the defensive-

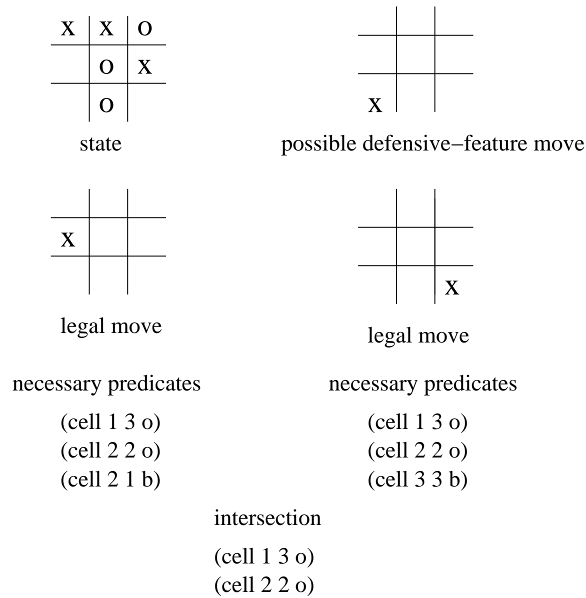

Figure 3.12: Finding defensive-feature predicates using intersection.

feature moves), the predicates for the defensive feature are the intersection of these sets because there may be other requirements for each specific move. However, the intersection eliminates the predicates that are needed for each different move which are not part of the defensive feature. For example, assume that there are three legal moves at the root state of the 2-ply game tree under examination and one of the legal moves is possible defensive-feature move. A set of defensive-feature predicates is found for each of the remaining two legal moves. The defensive-feature predicates are the required pedicates for the offensive feature to be applied at the next state. Each of them may contain predicates that are required for the move to be legal at the state, but the two moves allow the offensive feature to be applied at the middle state. Therefore, the move specific predicates are not needed since regardless of which move is taken the result is the same. The intersection of the two sets of defensive-feature predicates eliminates the move-specific predicates and makes the defensive feature more general. Figure 3.12 illustrates this example. Each set of necessary predicates contains a predicate that makes the move legal, but those predicates are not needed in the defensive-feature predicates and make the defensive feature less general. The intersection finds the defensive-feature predicates.

The predicates found and the possible defensive-feature moves make a defensive feature. In case there are no possible defensive-feature moves that can be found (even after backtracking) or no defensive-feature predicates can be found, the learning stops and another training run begins.

3.2.3 Backtracking the 2-ply Tree

In a 2-ply game tree, both offensive and defensive features can be found. The algorithm investigates higher levels in the game sequence to find more features. The highest level of state investigated in

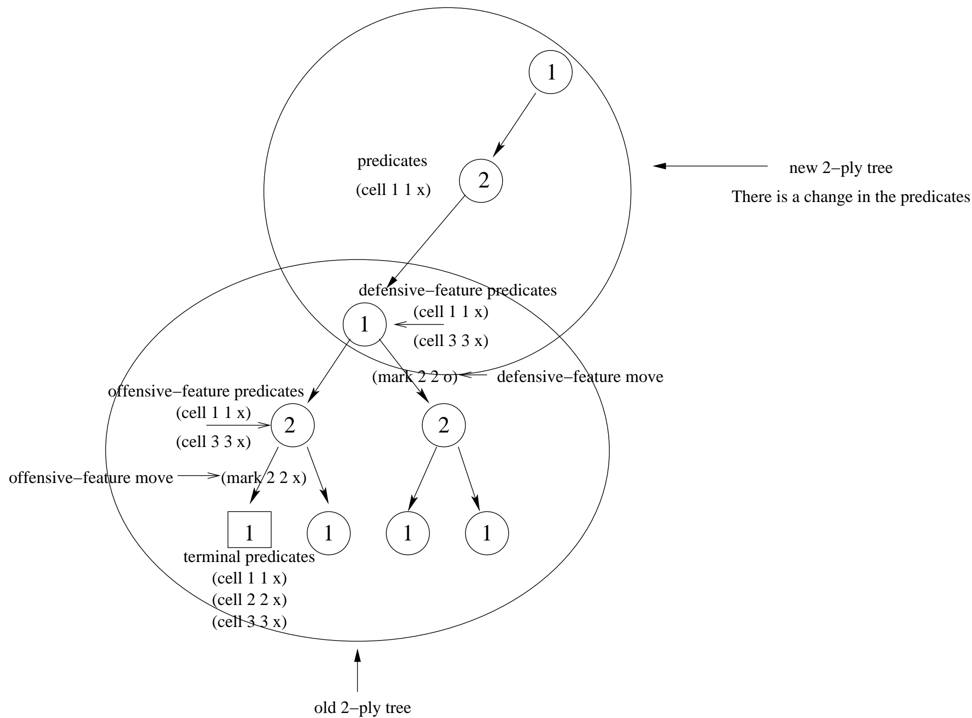

Figure 3.13: Finding the new 2-ply tree for further learning.

the game sequence is the root state of the 2-ply game tree in which the last feature is learned. That state becomes the leaf state of the next 2-ply game tree because the algorithm only learns from the states that are on the path that leads to the terminal state. The middle state and the root state are one and two higher level states, respectively. An example of finding the new 2-ply tree in tictactoe is presented in Figure 3.13.

In the example, the root state of the old 2-ply tree becomes the leaf state of the new 2-ply tree. Also, note that the middle state of the new 2-ply tree does not have all of the predicates of the leaf state ((cell 3 3 x) is not true in the middle state). Therefore, the move made at the new 2-ply tree to reach the leaf state have some contributions and an offensive feature can be learned.

To learn an offensive feature, the move made in the middle state should contribute to the leaf predicates of the new leaf state. However, due to the randomly generated game sequence, the program checks whether the move contributes to the leaf predicates or not. A backtracking process similar to the one in the defensive feature learning can be done if the new middle state contains all of the new leaf predicates. The backtracking is done by going higher up in the game sequence until the new middle state does not have all of the new leaf predicates. The new root state is the state that comes immediately before the new middle state in the game sequence regardless of its suitability to defensive learning. Therefore, backtracking is not done to find a suitable state for defensive feature learning. An example of finding the new 2-ply tree by discarding some states is presented in Figure 3.14.

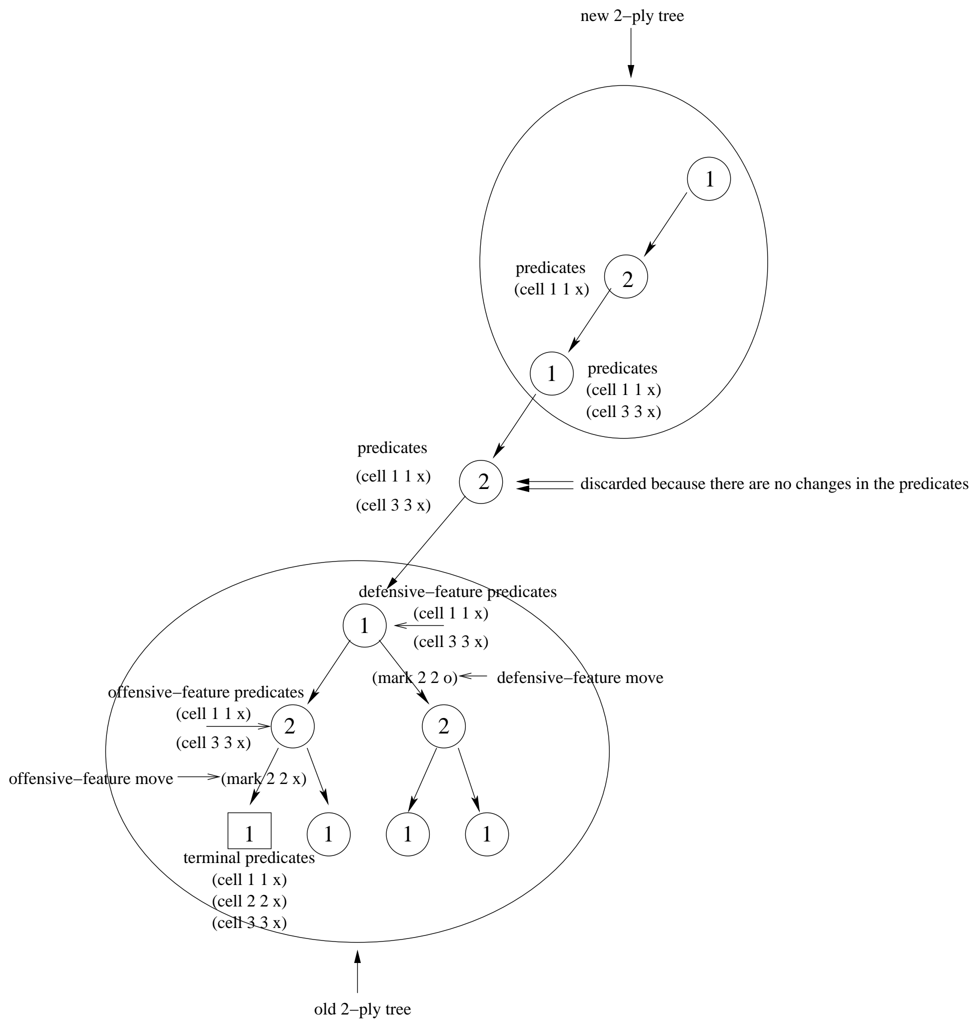

Figure 3.14: Finding the new 2-ply tree for further learning.

In the example, a state is discarded and backtracking is continued because the move made to reach that state does not contribute to the leaf predicates. The new 2-ply tree is found further up in the game tree.

The pseudecode for the feature learning process is presented in Figures 3.15 and 3.16. The function *trainPlayer* is the main learning function. A game sequence is generated using random move selection at each call and GIFL learns features from that game sequence up to the level limit (number of 2-ply trees in which learning occurs) specified by the third parameter. The function *createFeatureUsingStateFacts* finds the feature predicates for both offensive and defensive features. This function takes a 2-ply tree and finds the necessary predicates in the root state for reaching the leaf state using the given actions. Those necessary predicates are the feature predicates.

3.2.4 Implementation Details

The game description language allows flexibility when writing a game. However, there are some implementation details that need to be addressed to deal with different game descriptions.

First, GIFL learns the game type before starting to learn features. In some games, a goal condition does not depend on the predicates that are true in a state and there may not be terminal predicates. Checkers is an example of this type of game. At the terminal state, the goal condition only depends on previously captured pieces which are not present. In other games, there are terminal predicates present in the terminal state. Identifying the type of game is important because if the game type is the one with no terminal predicates, then the terminal predicates become all of the state predicates.

Second, GIFL runs simulations to find the features.

Third, after offensive-feature predicates are found using the algorithm in Figure 3.16, the possible features are checked by creating a new state. This state consists of only the offensive-feature predicates. The offensive-feature move is applied and if the resulting state satisfies the leaf conditions, the feature is added to the feature list. However, removing predicates one by one may result in some of the offensive-feature predicates missing and the feature cannot satisfy the leaf conditions. This type of feature is rejected. For instance, suppose that there are three stones consecutively placed vertically in connect4. If the stone in the middle is removed and a stone is placed in that column, the game description dictates that a stone is placed on top of every stone that has an empty space on top. Therefore, the empty place in the middle is replaced even though it is not supposed to be. This will result in the stone in the middle not being a part of the offensive-feature predicates even though it should be. To solve this problem, GIFL uses another method to find the offensive-feature predicates. If there are leaf predicates that are present in middle state and not present in the offensive-feature predicates, they are added to the offensive-feature predicates. The resulting possible feature is also checked if it helps the player to reach the leaf state or not. If it is useful, it is added to the feature base.

```
    trainPlayer(currentState, knowledgeBase, levelLimit)
1   stateList ← generate a game sequence
2   terminalPredicates ← find the terminal predicates
3   /* the 2-ply tree used in learning */
4   state leaf = stateList(terminal)
5   state middle = stateList(terminal-1)
6   state root = stateList(terminal-2)
7   leafPredicates = terminalPredicates
8   while (level <= levelLimit)
9       /* OFFENSIVE FEATURE DISCOVERY */
10      middleAction ← action made to reach leaf
11      createFeatureUsingStateFacts(middle,middlePredicates,
12      leafPredicates,leafAction,middleAction,winner,
13      rootPredicates,rootAction)
14      if feature is useful
15          add to knowledge base
16      else
17          createFeatureUsingTerminalPredicates(middle,
18          middlePredicates,leafPredicates,middleAction,winner)
19          if feature is useful
20              add to knowledge base
21      /* DEFENSIVE FEATURE DISCOVERY */
22      vector possibleRootActions
23      do
24          createFeatureUsingLegalActions(root,
25          possibleRootActions,leafPredicates,
26          leafAction,middlePredicates,middleAction,loser)
27          if possibleRootActions.size() == 0
28              middle = middle - 2
29              root = root - 2
30              if (not contains(getStateVector(middle),
31                          middlePredicates))
32              ||
33                 canPreventReachLeaf(middle,
34                    middleAction,leafPredicates,
35                    leafAction,winner)
36                             stop learning
37      while(possibleRootActions.size() == 0)
38      for all moves except possibleRootActions
39          /* find necessary predicates */
40          createFeatureUsingStateFacts(
41              root,possibleRootPredicates[i],
42              middlePredicates,middleAction,
43              possibleWrongAction,loser,
44              leafPredicates,leafAction)
45      rootPredicates ← intersection(possibleRootPredicates)
46      add to knowledge base
47      /* FIND NEXT LEAF, MIDDLE, ROOT, */
48      do
49          leaf = root
50          middle = middle - 2
51          root = root - 2
52      while(contains(getStateVector(middle),
53                      middlePredicates))
54      leafPredicates = rootPredicates
55      leafAction = rootAction
56      /* clear middle, root predicates and actions */
57      level++
58  end while
```

Figure 3.15: The learning algorithm.

```
     createFeatureUsingStateFacts(state middle,
     vector middlePredicates, vector leafPredicates,
     leafAction, action, player,
     vector rootPredicates, rootAction)
1      temp = middle
2      middleStateVector ← predicates of middle
3      for all predicates in temp
4          remove one by one,
5          new state is reducedTemp
6          if isLegal(reducedTemp,action)
7              reducedTemp.performMove(action)
8              stateVector ← predicates of reducedTemp
9              if (not contains(getStateVector(middle),
10                         middlePredicates))
11             ||
12                canPreventReachLeaf(middle,
13                    middleAction,leafPredicates,
14                    leafAction,winner)
15                           middlePredicates.add(predicate)
16         else
17             middlePredicates.add(predicate)
```

Figure 3.16: The function to find feature predicates.

Fourth, the features learned by GIFL are used when playing games.

Chapter 4 explains the mentioned cases and more in detail.

3.3 Using Features

Features are used to guide UCT playouts. The program checks each state during a simulation to see if a feature can be applied or not. If the predicates of a feature are matched in a state then the moves associated with that feature are given a value. After all of the applicable features are found, the program selects a move according to probabilities calculated by Gibbs Distribution. The features are heuristics to bias toward the random simulation with the expectation of achieving a more accurate result instead of doing pure random simulation. Hence, they are not always accurate. Therefore, the move with the highest calculated value is not always the best move. Gibbs Distribution provides a move selection policy to exploit the knowledge gained by using features and to explore the possible better moves with lower values.

If features are used in a random playout, all available moves are assumed to initially have no value. The value of a move can be changed if a feature with the that move can be applied in a state. The value of the move is set according to the formula $C^{level-1}$ where C is a constant between 0-1 and the level is the level of the 2-ply tree (where the level of the terminal state is 0) at which the feature is learned. This formula ensures that the features found close to the terminal state of the training game sequence will have greater value than the features found in higher levels because the lower level features lead to a win in fewer moves.

The value of C affects the probability distribution of the moves. When C is close to 0, more time

is spent on exploring the feature moves that were learned close to the terminal state. The exploration is more equally distributed when C is close to 1. The experiments determine that having a higher C value yields better performance in most games. For example, the winning percentage of the learning player against the non-learning player drops from 75% when C is 0.9 to 60% when C is 0.5 in knighttrough.

After all of the possible features are matched, the move is selected according to Gibbs Distribution except if there are any level 1 feature moves. A level 1 feature may mean a situation of immediate win or loss because the level 1 feature is learned from the 2-ply tree with the terminal state as the leaf state. Therefore, the move selection is done from the set of moves with value C, if there are any. If there are no level 1 features matched, then a probability is calculated for each possible move according to the Gibbs Distribution. The move is selected according to the probabilities calculated. The Gibbs Distribution provides a good exploration-exploitation balance to the move selection. Even though higher valued feature leads to a win in fewer moves, the outcome depends on the opponent's response. Therefore, other possible moves are explored. This exploration-exploitation problem is very common in Reinforcement Learning and Gibbs Distribution is one of the techniques used to tackle it. The formula for the Gibbs Distribution is:

$$p(a) = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}}$$

where $Q_t(a)$ gives the value estimate for the action a and τ is the parameter to regulate exploration-exploitation balance. High values of τ cause the action to be nearly equiprobable, whereas low values cause a greater difference in selection probability for actions that differ in their value estimates [23]. GIFL uses 0.5 for τ after experimenting with different values to get the best performance.

One key factor when using features is opponent modeling. The opponent is assumed to use the features with a lower probability than the player who uses GIFL for opponent modeling. If the player who has learned the features assumes that the opponent has the same knowledge, the weaknesses of the opponent may not be exploited. Therefore, a lower probability of using the features is given to the opponent. This ensures that the opponent does not benefit from information it does not have. However, if the opponent is assumed to not use the features, that may lead to a over-optimistic play that reduces the performance of the player who uses GIFL.

The pseudocode for how to use the features in UCT search is presented in Figure 3.17.

```
    DoMonteCarloSimulation(state currentState)
1   if features are not used
2       do random move selection
3   if features are used
4       vector statePredicates ← predicates of state
5       for each predicate in the statePredicates
6           features = knowledbase[predicate];
7           for each feature in features
8               if contains(statePredicates,feature(predicates))
9                   if isLegal(feature(move))
10                      moveValues[feature(move)] = C^(level-1)
11      if there are moves with value C
12          select move between them
13      else
14          gibbsDistribution(probabilities,moveValues)
15          selectMove(probabilities)
```

Figure 3.17: The algorithm to use the features in the UCT search.

Chapter 4

Implementation Details

Game Independent Feature Learning (GIFL) is a domain-independent algorithm. Even though GIFL is implemented for a GGP program, the algorithm can be used in other domains with some implementation differences. Therefore, some implementation details that are important to reproduce the work is explained in this chapter.

- Moves that change the state of the game are important for the learning. All moves are simultaneous in GGP. Alternating-move games are handled using a special move (noop) that does not change the state of the game. GIFL learns from alternating-move games. Therefore, identifying the noop move is crucial for implementing the algorithm in a GGP program because GIFL does not learn from noop moves.
  Maligne, the program that uses GIFL, finds the noop move before the learning starts. The move of the player that has a single move at the initial state is considered as a possible noop-move. After that, the possible noop-move is compared with a move of the player that has a single move during a number of simulations. At the end of the simulations, if all of the moves that were compared with the possible noop-move were as same as the possible noop-move, the possible noop-move is declared as the noop move for the game and used in the learning process.

- The learning occurs in alternating-move games. Actually, the algorithm works in simultaneous-move games, but the effectiveness will be much lower because the opponent has to make the same move as in the feature-move during the gameplay to get the desired effect of the learning. For example, a player has five legal moves at a state and chooses move number two. The opponent also has five moves at that state and chooses move number three. GIFL learns a feature from the game sequence. During gameplay, if the player encounters a state at which the feature learned from the example game-sequence is applicable and the move number two is legal, the gain from using the feature depends on the move that the opponent makes. Therefore, if the opponent makes the move number three again during the gameplay, the feature will give an advantage to the player. However, the player that uses learning has no control over the

opponent and the opponent may choose an another move. Therefore, learning is only used in alternating-move games to guarantee the effect that was seen during the learning regardless of the move that the opponent makes.
Maligne checks the number of moves for both players during the simulations before the learning. If both players have more than one move at any stage, the game is declared as being a simultaneous-move game and the learning does not start for the game.

- A GIFL feature consists of predicates that are a subset of a state. The algorithm removes the predicates of a state one by one to find which one belongs to feature predicates. However, the way Maligne represents a state differs between games depending on the game definition (GDL). In some games, states are incrementally constructed as seen in Figure 4.1, whereas some states do not change size as seen in Figure 4.2.

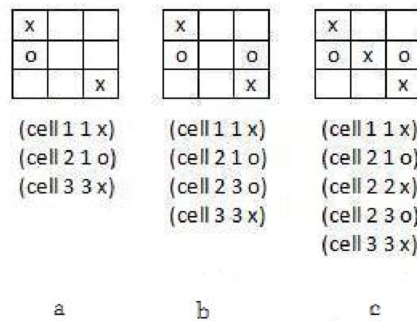

Figure 4.1: Incrementally constructed state.

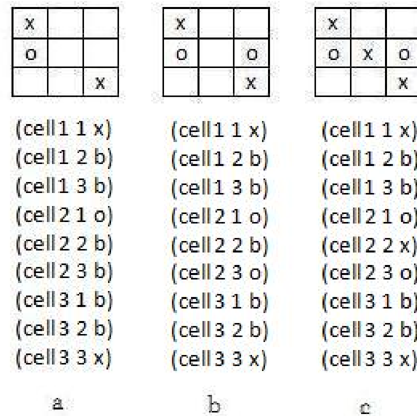

Figure 4.2: State that has same size during the game.

Maligne identifies the type of the state before the learning starts. The number of predicates of the initial state is compared with the number of predicates of randomly selected states during the simulations. The states that are being compared to the initial state are selected with 20%

probability which determined the type of the state most efficiently (accurately while comsuming less time) during the experiments. If the number of predicates of a selected state is not equal to the number of predicates of the initial state, the type of the state is considered as incrementally constructed for the game. Otherwise, the state is considered to have the same number of predicates throughout the game.

Determining the type of the state is important for GIFL because the learning algorithm removes the predicates from a state for the games that have incrementally constructed states. On the other hand, Maligne replaces them with a predicate from the initial state for the other games during the learning because removing a predicate from a state will create an incorrectly constructed state in those games.

- Terminal predicates are essential for GIFL. In fact, first phase of the algorithm is finding the terminal predicates. However, some games do not have terminal predicates and the whole state is considered as the terminal predicates for those games. Therefore, determining if a game has terminal predicates or not is necessary for the learning.

  Maligne determines if a game has terminal predicates or not during the simulations. Maligne looks for the terminal predicates for each simulation. If the total number of simulations that do not have terminal predicates are lower than 20% (experimentally determined) of the total number of simulations, the game is considered to have terminal predicates. Otherwise, the terminal state is considered as terminal predicates for that game.

  Terminal predicates are a subset of the terminal state. GIFL may learn from the terminal state without the need of the terminal predicates. However, the efficiency of the learning will drop considerably because of the lack of generality. That is why GIFL tries to find the terminal predicates whenever it can and discards some training data in favor of finding terminal predicates in some games.

  Some games have more than one terminal conditions and some of them may lead to a terminal state that does not have terminal predicates. That is the reason of counting the simulations that do not have terminal predicates. If this number is lower than 20%, then the learning algorithm discards the terminal conditions that lead to a terminal state that does not have terminal predicates in favor of learning efficiency.

- Looking for the features in a state is important part of GIFL. Maligne uses a hash map to store the features. Features are stored in a hash map with the first predicate of the feature predicates as the key. A state is converted to a predicate list and each predicate is used to access the possible feature matches. The value that is accessed in the feature map is a list of features that share the same predicate as a first predicate of their predicate list. If there are any matches in the feature map for a predicate, for each of the possible features, the rest of the feature's predicates are searched in the state. A feature is matched when all of the feature predicates

are present in the state and the feature move is legal.

The speed of the algorithm is important because the accuracy of the value estimation in UCT is proportional to the number of simulations. However, feature matching is acting as a bottleneck in the performance of GIFL. For example, 30% of total computation time is spent on searching the predicates of a feature in a state by Maligne. Using features are expensive with just searching for predicates in a state. Including looking for feature moves in legal moves, calculating probability distribution for the moves using Gibbs Distribution and accessing the feature map, total computation time spent on using the features needs to be reduced.

The ideal case in which the cost of the feature matching is reduced would be without losing any information. This can be achieved with generalizing the features. The features are learned from different terminal predicates variations. For example, there are 8 different terminal predicates variations in breakthrough, but the only difference between them is the x-coordinate of a piece. The eight different features that are learned from those terminal predicates can be generalized into one feature having x-coordinate as variable. This will reduce the cost of the feature matching by a factor of 8. This idea will be revisited briefly in the Future Work section of the Chapter 6.

Chapter 5

Experiments

The learning algorithm is evaluated empirically in the GGP context in this chapter. The objectives of the experiments are:

- To examine the features learned by GIFL. The number of features learned by GIFL in a fixed time is investigated and a number of features are presented.
- To see the effectiveness of GIFL features using standard, random UCT simulation. The number of simulations for both players are the same number and the learning is done offline (before the start-clock) for this experiment. Therefore, the only difference between the players is the usage of learning.
- To see the effects of GIFL computation time on the number of the UCT simulations. In the UCT search, the number of simulations is crucial to performance. The overhead of looking for the presence of GIFL features may impact the speed of the UCT search.
- To see whether using GIFL features effects the average length of the simulations done during the UCT search. GIFL should be able to identify wins/loses earlier in the search.
- To see the effectiveness of using GIFL features in a fixed time setting. Both players have an equal amount of time and the number of simulations is not restricted. The learning is done during the start-clock.
- To see the effectiveness of the opponent modeling.
- To see the effectiveness of using different feature-matching methods to improve the speed of the algorithm.

The player that uses GIFL features to guide the random simulation is called the learning player. The player that does random simulation without guidance is called the non-learning player. Therefore, the only difference between the learning player and the non-learning player is that the learning player uses the learned features to guide the random simulation during the UCT search.

The experiments were prepared using the game definitions available at the Stanford GGP repository [10] and game definitions used in GGP competitions 2008 and 2009. The games that were used in the 2008 competition had cryptic names like game1, game2, etc. All games are 2-player, alternating move and prefect information.

In some games being the first player or second player may be advantageous. Both the learning player and the non-learning player take the first and second player roles an equal number of times so that the overall result of the experiment is not affected by the placement of the players.

All experiments were run on a PC with Intel(R) Xeon(R) 2.80GHz CPUs and 32 GB of RAM for each player.

5.1 Learning GIFL Features

Learning is done while the UCT is running. When the learning completes, the features are used during the UCT simulations. The number of training games is set by the programmer and GIFL can be used when the learning is completed. For most games in which GIFL is tested, the number of features does not increase continuously because there are not many different variations of the terminal predicates. GIFL learns from terminal predicates. Therefore, the number of features that can be learned is proportional to the number of terminal predicate variations. For example, there are 8 different terminal predicate combinations in breakthrough, but the number is much higher in connect4. Also, the number of GIFL features that is learned with the same number of training games is much higher in connect4 compared to breakthrough.

The number of features learned in a fixed time also depends on the game because GIFL features are learned from game sequences which are created from random simulations. Therefore, the length of the simulations, the time spent on making a move, the average number of predicates in a state, and the average number of legal moves all contribute to differences in the number of features learned.

In addition, the distance of 2-ply tree from the terminal state effects the number of GIFL features that is learned because GIFL can learn one offensive and one defensive feature from each 2-ply tree. Increasing the level of the 2-ply tree in which learning occurs increases the number of features, but the computation time for using the features also increases significantly with the number of features. Therefore, the level of the 2-ply tree is limited to three in the experiments. This ensures that at most 6 features can be learned in one training simulation.

In further experiments, the number of training games is limited to control the number of the features that are learned. To see how fast GIFL learns the features, an experiment was done. The number of features that GIFL learns in two minutes for each game is presented in Table 5.1.

The defensive features are learned to counter offensive features. Thus, the number of defensive features are always lower than the number of offensive features. Table 5.1 confirms this.

GIFL features try to reach to the terminal state where the player wins the game or to avoid the terminal state where the player loses. Figure 5.1, Figure 5.2 and Figure 5.3 are some example GIFL

Games from Stanford GGP repository and GGP Competitions

name	source	n. of features	n. of offensive features	n. of defensive features
game2	2008 competition	135	75	60
pawn whopping	2009 competition	328	185	143
knightthrough	2008 competition	134	79	55
game1	2008 competition	1869	1014	855
breakthrough	2007 competition	120	63	57
checkers	[10]	895	458	437
connect4	[10]	2187	1108	1079
chess	[10]	25	13	12
game5	2008 competition	1357	686	671
pentago	[10]	679	405	274
quarto	[10]	8527	5254	3273
game6	2008 competition	1456	813	643
game3	2008 competition	1555	812	743
checkersbarrelnokings	[10]	5262	2830	2432
game4	2008 competition	1101	558	543

Table 5.1: Number of features that is learned by GIFL in two minutes.

features.

In pawn whopping, the white player aims to reach the bottom of the board and the black tries to reach top part of the board. The white has an advantage in Figure 5.1. The black has to capture the two pieces. Otherwise, the white player can win in two moves. GIFL learns to capture those two pieces after encountering the same position and losing the game during the training simulations because black only has one move, thus can capture one white piece in Figure 5.1.

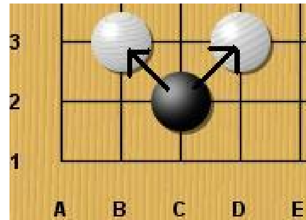

Figure 5.1: A defensive feature from pawn whopping.

In connect4, a player aims to connect 4 pieces in a row, column or diagonally. The red has an advantage in Figure 5.2. GIFL learns to drop a red piece to win the game. Conversely, GIFL also learns to prevent red from winning the game by dropping a blue piece before red has the chance.

In checkers, a player tries to capture opponent pieces by jumping over it to the adjacent vacant square beyond it. The white king has to play backwards to avoid being captured in Figure 5.3. GIFL encounters a game sequence where white loses the game after losing its last piece (the white king in Figure 5.3) and learns to avoid capture.

The number of features that GIFL learns in two minutes is more than the number that is learned in further experiments where the number of training games is limited instead of using a fixed time. Chess is an exception in that regard. Reaching to terminal state takes much more time in chess than other games. The number of simulations per move in chess is also much lower than in other games.

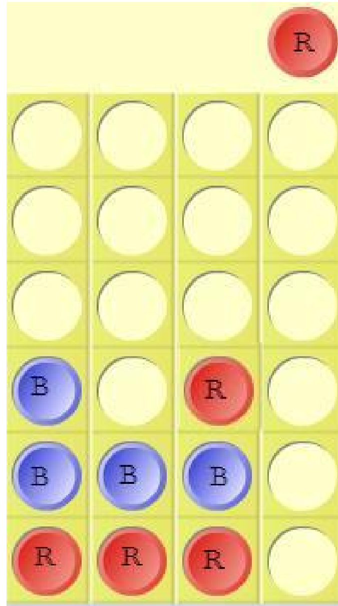

Figure 5.2: An offensive feature from connect4.

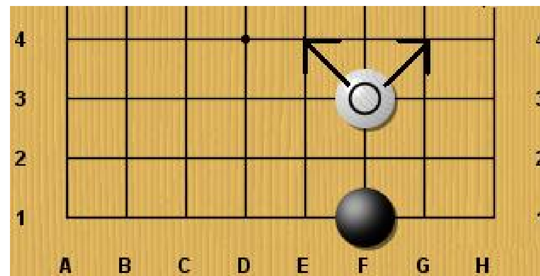

Figure 5.3: A defensive feature from checkers.

5.2 Effectiveness of GIFL With Fixed Number of Simulations

Next we experiment when the number of simulations per move is limited for both players and learning is done offline. This experiment demonstrates the effectiveness of the learned information without worrying about the computation time needed to learn the features and the cost of identifying them in a state. In fact, this is the best-case scenario for GIFL.

In addition, the probability of using features in a random simulation is introduced as described earlier. For this experiment, the learning player uses the features all the time and the opponent uses the features only half of the time during the random simulation phase of the UCT. If the learning player assumes that the non-learning player has the same knowledge, that would prevent the learning player from exploiting the weakness of the random move selection of the non-learning player. However, using the features half of the time allows the learning player to exploit the random play and still play safer assuming the non-learning player has some amount of knowledge. If the learning player assumes the non-learning player has no knowledge at all, the performance of the learning

name	n. of simulations	n. of games	learning-uct	win percentage	win with 95 % confidence
game2	1000	20	193-7	97.0 %	√
pawn whopping	1000	20	190-10	95.0 %	√
knightthrough	1000	20	184-16	92.0 %	√
game1	1000	20	170-30	85.0 %	√
breakthrough	1000	20	165-35	82.5 %	√
checkers	150	20	156-44	78.0 %	√
connect4	1000	100	115-85	57.5 %	√
chess	25	40	102-84	56.0 %	
game5	1000	40	111-94	55.0 %	
pentago	1000	100	100-100	50.0 %	
quarto	1000	100	98-102	49.0 %	
game6	1000	100	96-104	48.0 %	
game3	1000	100	91-109	45.0 %	√
checkersbarrelnokings	1000	100	61-139	30.0 %	
game4	1000	40	100-100	-	

Table 5.2: Effectiveness of using GIFL.

player drops because the learning player tries to exploit every weakness of the random simulation. However, the non-learning player accumulates knowledge using simulations. Therefore, assuming the non-learning player has some knowledge gives best performance for the learning player against the non-learning player.

The number of training runs is limited to 500 unless specified otherwise. 500 training runs is shown to be the best for most of the games used in the experiments making the learning player avoid learning too much information so that the using features is too slow and learning too little so that the using features is not effective. The learning time may vary between 100 training runs per minute in breakthrough and 10 training runs per minute in chess.

The level of 2-ply tree in which the learning occurs is limited to 3. This reduces the number of features learned and the time spent in random simulations. The learning player can learn one offensive and one defensive feature at each level of the 2-play tree. Therefore, the number of features learned from a training run is limited to 6 at the most. However, in practice the number is lower than that due to repetitive features or being unable to learn any features.

The results are in Table 5.2. The number of simulations per move and the number of test games are shown with the name of the game. The number of test games varies from game to game due to the time constraints. Although the number of games played is low for some games, it should be noted that the effectiveness of the learned information is clear when the learning player wins decisively on both sides. Games with close results are usually tested with more games.

The points shown in the table are total of the average scores that the players get playing both sides. For instance, learning player got a total 193 points in game2. This is the total score of 100, the average score of the learning player when it is the first player, and 93, the average score of the learning player when it is the second player. The winning percentage is calculated using all the games.

Of the 15 games that were used, the learning player defeats the non-learning player in nine of the games. The knowledge does not significantly affect the results in three games. In two games,

the learning player loses by a small margin. Using learned knowledge decreases the quality of play in only one game, checkersbarrelnokings.

In seven of the nine games that the learning player has advantage over the non-learning player, the results are statistically significant. In GGP competition setting where games start from the same initial state, the learning player is expected to beat Maligne's UCT player in these games with 95% confidence. This shows that GIFL features improve the performance of Maligne's UCT search.

Starting from the same initial state may result in test games that were played identically. However, UCT search does random simulations. Therefore, the test games were not repetitions of the same game. In the experiments, 12 of the 15 games differentiated in less than five moves and the three remaining games differentiated in less than 10 moves.

It should be noted that the games in which the learning does not affect the results are not very interesting: the first player always wins in pentago, all games are tied in game4 and all games end in less then 10 moves in quarto. Also note that, chess and game5 have terminal conditions that give both players 100 points or 0 points. Therefore, the total points for the learning player and the non-learning player do not add up to 200 (total points that can be taken for most games) for these games.

One game that GIFL does not improve in the performance over that of UCT search is checkersbarrelnokings. Although the checkesbarrelnokings game is similar to the original checkers at which the learning player has a clear advantage, the learning player loses badly in checkersbarrelnokings as shown in Table 5.2.

In checkersbarrelnokings, due to lack of kings and forced capture moves, the number of legal moves per step is low. Therefore, the non-learning player does less unnecessary exploration. For example, there are 20 legal moves in a state in breakthrough. When the learning player uses a GIFL feature, this gives an advantage over the non-learning player because the non-learning player explores all of the 20 moves. However, the average number of legal moves for checkersbarrelnokings is low and the advantage gained from using GIFL features is also lower than gained from the breakthrough.

In addition, learning capture moves which are the most important moves to win the game is not useful because they are forced moves. Therefore, GIFL can only make difference by learning defensive feature moves. Learning defensive features is also harder in checkersbarrelnokings because escaping an imminent capture is done by capturing the opponent piece (forced move). GIFL learns features that help the player to avoid getting captured in the next turn. However, GIFL features do not contain information about other pieces of the opponent and opponent has 12 pieces at the beginning. Opponent may have already a piece other than the one being avoided, in position to capture the player's piece. Further experiments support this theory. The non-learning player still beats the learning player decisively when the learning player has only first level defensive features which try to avoid getting captured in the next turn.

name	n. of simulations (learner/uct)
game2	46 %
pawn whopping	65 %
knightthrough	93 %
game1	104 %
breakthrough	79 %
checkers	36 %
connect4	20 %
chess	74 %
game5	32 %
pentago	156 %
quarto	34 %
game6	58 %
game3	99 %
checkersbarrelnokings	38 %
game4	47 %

Table 5.3: Computational cost of using GIFL.

In checkersbarrelnokings, using the standard UCT simulations instead of exploring the moves that are suggested by GIFL features is more effective because GIFL feature do not explore the moves related to the other opponent pieces on the board.

5.3 Effects of Computation Time Spent By Using Features on the Number of UCT Simulations

As it can be seen in Table 5.2, the learned features are clearly effective. However, in actual gameplay the computation time is also a factor. To measure the difference in the number of simulations per move each player can make in a fixed time, another experiment has been performed with 30 seconds per move. The number of training games is the same as in the previous section. The results of Table 5.3 show that in half of the games, the computation time is not a big factor, but in 5 games (checkers, connect4, game5, quarto, checkersbarrelnokings) the learning player can only make $1/3$ or less of the simulations that regular UCT can make. The number of simulations is crucial for the performance of UCT search. Therefore, the performance of the learning player is affected adversely in those five games because of the usage of GIFL features. However, the quality of play should not suffer as much as when the non-learning player does less number of simulations. The accuracy of the simulations for the learning player and the non-learning player are different due to GIFL features. This issue is also investigated in later sections of this chapter.

Table 5.3 shows the number of simulations that the learning player can make given the same amount of time that the non-learning player has. The number is expressed as a ratio of the number of UCT simulations that both players make. For example, in connect4, learning slows the program down by a factor of 5 – to 20% the speed of the non-learning player. Note that in two games learning

had the pleasant side effect of speeding up the calculations as a result of finding early wins in the simulation phase instead of lengthening the games with random moves.

The slowdown is caused by the cost of identifying if a GIFL feature is present in a position. The discovery of GIFL features is done in parallel with the simulations at the start-clock and is completed during before the first move in all of the tested games.

Using features hurts most in connect4. The number of features, how frequent these features's predicates are present in a state and the average number of predicates in a feature are the important factors that effect the computation time of using features.

GIFL learns around 2000 features in connect4. This is the third biggest number out of all games. This number increases the number of features that are checked at each turn of the simulations. However, there are two other games (checkersbarrelnokings and quarto) that have more features but have been effected less than connect4. The reason lies in the structure of connect4. The predicates stay true until the end once placed. Therefore, if a predicate of a feature becomes true in a state, that feature will be checked at every turn until the game ends. The predicates can become false in later turns in the other two games. This is the reason that connect4 is adversely affected by using features.

The third possible reason is the average number of predicates in a feature. The length of the features is shorter than quarto and same as checkersbarrelnokings. Therefore, this cannot be the reason of low number of simulations with the learning.

Breakthrough and knightthrough are two games that GIFL does not decrease the number of simulations per move significantly. The number of features is low for these two games. The feature predicates become true when game is close to termination. Therefore, the number of features investigated per turn is also low. The length of the features is short in these two games with average of lower than 2 predicates.

5.4 Effectiveness of GIFL Over Length of the Simulations

GIFL aims to get more accurate estimates from the simulations of UCT using the information that is learned. An outcome of using the learned information is that simulations should be on average shorter in length. For example, one advantage of using features is that trivial wins are not missed in the simulations. However, if the moves are selected randomly, the probability of selecting a winning move is $1/(\#oflegalmoves)$. Therefore, making random moves will result in unnecessary exploration because no skilled player would miss a move that results in a win for the player.

To see the effectiveness of the learning over the length of the simulations, the average length of the simulations that are done by the players over one hundred moves are calculated and are shown in Table 5.4. Both players do enough simulations in one hundred moves to get an unbiased estimate of the length of the simulations. The learning is done during the start-clock and completed before the first move.

name	length of simulations (learner/uct)	n. of simulations (learner/uct)	simulation speed estimate (learner/uct)
game2	51 %	46 %	23 %
pawn whopping	82 %	65 %	53 %
knightthrough	45 %	93 %	42 %
game1	53 %	104 %	55 %
breakthrough	61 %	79 %	48 %
checkers	73 %	36 %	26 %
connect4	95 %	20 %	19 %
chess	99 %	74 %	73 %
game5	101 %	32 %	32 %
pentago	68 %	156 %	106 %
quarto	100 %	34 %	34 %
game6	97 %	58 %	56 %
game3	52 %	99 %	51 %
checkersbarrelnokings	178 %	38 %	68 %
game4	116 %	47 %	55 %

Table 5.4: Average length of simulations.

As it can be seen in Table 5.4, the length of the simulations is shortened for most games, especially for the games in which GIFL is shown to be effective. In effect, by the help of GIFL features, the learning player finds short winning sequences that the non-learning player overlooks in UCT simulations.

Having shorter simulations as a result of the learning also decreases the damage done by the computation time of the feature matching. For example, the learning player does the same number simulations as the non-learning player in game3. However, identifying features in a state takes computation time and should decrease the number of simulations of the learning player. The length of the simulations can compensate for the time spent in identifying features in a state. The simulations are shorter for the learning player in game3 as it is shown in Table 5.4. Therefore, although the simulations of the learning player is half as slower than the non-learning player, the number of simulations that both player makes per move is the same because the length of the simulations of the learning player is half as shorter than the non-learning player.

Checkersbarrelnokings is adversely affected from using features in this category. It is explained in Section 5.2 that learning defensive features does not help the learning player to avoid capture. However, the learning player can avoid capture when there are less opponent pieces in the game, especially in the cases where each player has a piece left. The opponent player is also assumed to use features for opponent modeling and this causes longer games during the learning player's simulations. In the experiments, the length of the simulations of the learning player is found to be closer to the length of the gameplay.

5.5 Effectiveness of GIFL With Fixed Time Per Move

The second section of this chapter shows that using learning improves the performance with the same number of UCT simulations. This shows that the value estimation for the moves resulting from UCT simulations is more accurate with the help of GIFL features. On the other hand, the computation time needed for using features reduces the number of simulations that the learning

name	n. of games	learning-uct	win percentage
pawn whopping	20	190-10	95.0 %
game2	20	140-60	70.0 %
checkers	20	110-90	55.0 %

Table 5.5: Effectiveness of using GIFL with 30 seconds per move.

player can do during a fixed time as compared to the non-learning player for most games. This section investigates that even though the number of simulations that the learning player performs is smaller, the quality of these simulations may still give an advantage to the learning player over the non-learning player.

To show the practical advantages of GIFL, three sample games that the learning has effected in various degrees (number of simulations, length of the simulations, etc.) were selected. The effectiveness of GIFL in a fixed time setting for the remaining games can be inferred from the results of those three games by using the results from the previous experiments as heuristic. GIFL was tested on the selected three games using a fixed time per move rather than a fixed number of simulations. Each player is given 30 seconds per move and the learning is also done online. That means that the non-learning player starts the UCT simulations during the start-clock using all the computational resources, whereas the learning player allocates some computational resources to the learning along with the regular (non-learning) UCT simulations. When the learning is done, the learning player uses the features to guide the UCT simulations.

The result of the experiment can be seen in Table 5.5.

The first game is from the 2009 GGP competition, pawn whopping. This is an example where the usage of features does not hurt the number of simulations much and the average length of the simulations is shorter with the learning, thus the learning is very effective. The results of the experiment shows the overwhelming superiority of the learning player over the non-learning player.

The second game is from 2008 GGP competition. Game2 is an example of a game where the usage of features hurts the number of simulations significantly, but the average length of the simulations is significantly shorter with the learning. This game is a case where the lower number of simulations may hurt more than the more accurate exploration can compensate. However, as can be seen in Table 5.5, the learning player still wins over the non-learning player decisively. It should be noted that the low number of simulations still hurts the learning player because the winning percentage drops from 97.0% when using a fixed number of simulations per move (Table 5.2) to 70.0% when using a fixed time per move.

The third game is the well-known checkers game. Using features in this game hurts the number of simulations (the number of simulations for the learning player is $\frac{1}{3}$ of the non-learning player) and does not shorten the average length of the simulations substantially. This game is the one with the lowest expectations from learning among the three games tested. The results of the experiments

also reflect that the learning player barely maintains the advantage over the non-learning player. However, with the fixed number of simulations per move, the learning player dominates the non-learning player undisputed. Thus, anything that can speed up the GIFL simulations will yield better checkers performance.

The results from Table 5.5 show that computation time problem of using features does not hurt the performance of the learning drastically. A more efficient way of using features will improve the performance of the learning player, as it can be seen in the results of using features with the same number of simulations that the non-learning player does.

The results for the remaining games in a fixed time setting can be inferred by using the effects of the learning on the number of simulations per move and the length of the simulations as heuristic of the effectiveness of the learning. This is explained with the three selected games above. If the learning is effective in a game, the learning player does not lose the advantage even in a fixed time setting, but if the learning is ineffective, the learning lowers the performance of the player more in a fixed time setting. Two more games were tested with a fixed time to confirm this hypothesis.

The first game is connect4. The number of simulations per move is lower than checkers in connect4 and the length of the simulations is shortened less than checkers. The heuristic predicts GIFL will lose performance in a fixed time setting just like in checkers. The experiment confirms that with the learning player is only able to win 45% of the games instead of winning 58% in a fixed number of simulations setting.

The second game is breakthrough. The number of simulations is decreased by 20% in breakthrough, but the length of the simulations is shortened by close to half. The heuristics predicts that GIFL will not lose performance in a fixed time setting. The experiment also confirms that with an overwhelming victory of the learning player, winning 85% of the games.

The five games that were tested give insight on how to predict the effectiveness of the learning in a game by looking at how learning effects the computation time and the length of the simulations. The remaining of the 15 available games are also tested in a fixed time setting and the results are shown in Table 5.6. The number of the test games for all of the games is 20.

Note that the heuristic holds for all of the games.

5.6 Effectiveness of the Opponent Modeling

The non-learning player does not extract any features from the game and relies on the UCT search. The learning player models the play of the non-learning player and can optionally allow the opponent access to the learned features. The model used by the learning player influences the value estimations.

The amount of knowledge that is given to the non-learning player is determined experimentally in this section. The experiment was done in a fixed time setting. The average scores of the learning player with three different opponent models against the non-learning player are shown in Table 5.7.

name	learning-uct	win percentage
game2	140-30	70.0 %
pawn whopping	190-10	95.0 %
knightthrough	150-50	75.0 %
game1	160-40	80.0 %
breakthrough	170-30	85.0 %
checkers	110-90	55.0 %
connect4	90-110	45.0 %
chess	75-125	35.0 %
game5	40-160	15.0 %
pentago	100-100	50.0 %
quarto	80-120	05.0 %
game6	70-130	35.0 %
game3	100-100	50.0 %
checkersbarrelnokings	30-170	15.0 %
game4	100-100	-

Table 5.6: Effectiveness of using GIFL with 30 seconds per move.

name	no knowledge	half access	full access
game2	30.0	70.0	70.0
pawn whopping	70.0	95.0	85.0
knightthrough	50.0	75.0	60.0
game1	55.0	80.0	70.0
breakthrough	45.0	85.0	70.0
checkers	60.0	55.0	58.0
connect4	10.0	45.0	35.0
chess	40.0	38.0	65.0
game5	48.0	20.0	25.0
pentago	50.0	50.0	50.0
quarto	42.0	40.0	33.0
game6	33.0	35.0	40.0
game3	25.0	50.0	48.0
checkersbarrelnokings	08.0	15.0	10.0
game4	50.0	50.0	50.0

Table 5.7: Effectiveness of the opponent modeling.

A full access model means the opponent is modeled as having full usage of the learned features. The half access model is only allowed to use the features half of the time. We report the average score over 20 games.

Table 5.7 shows that in almost every game assuming that the non-learning player has half of the learned information gives better performance for the learning player. In pawn whopping, the learning player does better when it models the opponent with half access to GIFL features. Modeling the opponent with full access to GIFL features is the best choice in chess, but in checkers there is not much difference in the performance of the learning player when changing the opponent models.

The best opponent modeling technique is game dependent as it can be seen from the results of pawn whopping, chess and checkers. However, it is beyond the scope of this thesis and left for future work.

5.7 Effectiveness of Different Feature Matching Methods

The first section of this chapter shows that using features improves a GGP program's performance for most games. The second section shows that using features is expensive for some games. The fourth section shows that using features may hurt performance because of the computation expense. The GNU profiler software (gprof) shows that most computation time (38.0% of the overall computation time for a simulation) is spent on looking if the predicates of a feature are present in a state or not. Therefore, if feature matching can be done more efficiently, the performance of the program will improve. This section investigates the performance of different methods of feature matching.

Along with the baseline method that is explained in this thesis, there are four different feature matching methods which are tested. These methods are:

- baseline method. Each predicate of a state is used as a key for the hash table of features. The hash table returns a number of features whose first predicate is the key. Then, all of the predicates of the features are searched in the state using linear search. If all of the predicates of a feature are present in the state, feature matching is completed.

- binary search method. A binary search algorithm is used to find if a predicate is present in a state or not. This will asymptotically reduce the time spent on search from linear time to logarithmic time.

- state map. A hash table is used to store the predicates of a state. Then, predicates of the feature can be used as keys to look if they are present in a state or not. This method asymptotically will reduce the time spent on search even lower than binary search, but there is computation time spent on allocating and deleting memory for the hash table.

- state map and move map. Although not as frequent as the search of the predicates in a state, the moves of the features are searched to find if they are legal in a state or not. This is also done using linear search in the baseline method. However, just like the predicates of a state can be mapped into a hash table, the legal moves of a state can be mapped into a hash table. Then the moves of the predicates that are matched to a state can be used as keys to look if they are legal in the state.

Connect4 is used to test the effectiveness of these four methods because the penalty for feature matching is highest in this game. The learning player can only do 20.0% of the simulations that the non-learning player does in the same amount of time.

Given 30 seconds per move, Maligne (using each of the four different feature matching methods) is tested on connect4 over 10 games. The average number of simulations of the first 10 moves is shown in Table 5.8.

As it can be seen in Table 5.8, there are no significant improvement with the three different methods over the baseline method. In fact only one method does slightly better than the baseline

name of the method	n. of simulations
baseline	18617
binary search	18218
state map	20113
state map + move map	18920

Table 5.8: Average number of simulations for 100 moves in connect4.

method. This shows that the search of a predicate in a state or the search of a move in legal moves of a state does not take much computation time. Regardless of the method that is being used, the computation time does not differ substantially. However, the total computation time of all the searches hurts the performance of the program. This is confirmed by the fact that the properties of the GIFL features of connect4. The number is high and once a predicate of a feature becomes true, that feature will be searched in all future states. Therefore, it can be concluded that more drastic measures have to be taken to improve the performance of the feature matching such as changing the way a feature is stored or the way a feature is defined as well. These changes are left for future work.

5.8 Conclusions

GIFL is clearly shown to increase the accuracy of the value estimation of the UCT search by defeating the non-learning player when both players use the same number of simulations and by shortening the average length of the simulations by getting rid of the unnecessary moves. It should be noted that using GIFL may require a significant amount of computation overhead that can reduce the number of simulations completed in a fixed time. However, even with a lower number of simulations, GIFL is still shown to be effective.

Learning GIFL features is not computationally expensive and is completed during the start-clock for most of the tested games. In addition, the discovery of GIFL features during the beginning of the game does not cause performance loss even it extends to the play-clock because GIFL is not very effective at the beginning of a game as it is in the end sequence since GIFL features are learned starting from the terminal state and are not encountered as frequently as in the beginning.

Chapter 6

Conclusions and Future Work

Researchers use games as a testbed for Artificial Intelligence research. Most of the effort has focused on playing specific games very well. Even though this has led to the creation of a number of world-class level game programs, the value of these programs as AI research is limited because each of these programs can only play one specific game and the analysis of the game is done by the programmer.

To play different games with the same program requires domain-independent learning. However, there has been little or no improvement on the state of the art in the last fifteen years. This may be attributed to the fact that knowledge extraction is a very hard problem. To encourage researchers to investigate this issue, General Game Playing was suggested. GGP programs receive a game description as input, analyzes it and then plays the game without human intervention.

However, domain-independent learning is still a hard problem and researchers tend to use algorithms that do not require any information about the domain. The primary example of this is Upper Confidence Bounds applied to trees (UCT). Moreover, UCT search is the state of the art in the GGP domain as domanstrated by the last three winners of the annual GGP competition.

This thesis shows that a simple domain-independent learning algorithm can improve the state of the art in 2-player, alternating move games in GGP domain. The algorithm, Game Independent Feature Learning, that is presented in this thesis learns knowledge chunks consisting of a subset of a state and legal moves. GIFL has been implemented into the program, Maligne, that came third at the 2009 GGP competition and the algorithm has shown promising results in some of the games that are frequently used in GGP competitions.

GIFL is an effective but simple algorithm that demonstrates the importance of the domain-independent learning. The remaining part of the chapter presents the issues and improvements to GIFL in addition to the future work that can be done related to domain-independent learning.

6.1 Suggestions for Improvements on GIFL

The following suggestions can be used to extend and/or enhance GIFL..

	1	2	3
8			o
7		o	o
6	x		
5		x	

Figure 6.1: A board position.

- GIFL currently works on alternating move games, but it can be extended to simultaneous-move games. A simultaneous move can be turned into an alternating move. The result would be two moves: the move of a player from the original simultaneous move and a no-operation move, and a no-operation move and the move of the remaining player from the simultaneous move. Then, GIFL will learn features as if the game is an alternating move game. Using the features may not be as efficient as in the alternating move games though, considering that the GIFL features do not contain any information about the opponent counter-moves that is required in the simultaneous games. However, this issue is also addressed in this section.

- GIFL learns a defensive feature to make an offensive feature ineffective. The algorithm stops when a defensive feature cannot be found. However, learning an offensive feature does not depend on defensive features. Therefore, the learning algorithm can be changed to continue learning offensive features. This will lead to more efficient use of the learning data.

- The learning starts at the terminal state with the terminal predicates. However, there are no terminal predicates in some games. GIFL considers the whole terminal state as the terminal predicates in those games. This leads to slow learning due to the large number of predicates, requires pre-processing to determine whether there are terminal predicates or not, and postulates a more complicated implementation. To solve the need for the terminal predicates issue, learning can start at the state that leads to the terminal state. GIFL can find the offensive-feature predicates and move without the terminal predicates by checking if the offensive feature will yield the same result as the terminal state.

- GIFL assigns values to the feature moves without considering an opponent response. There may be a defensive feature for every offensive feature. Therefore, the opponent may respond to make the offensive feature ineffective. GIFL features can be extended to include information about the opponent's possible responses. This information can be used to assign lower values to the offensive feature when the opponent has a counter-measure.

  Figure 6.1 shows a board position from the game breakthrough to illustrate the potential gain from this modification. The piece "x" located at x-coordinate 1 and y-coordinate 6 aims to reach to y-coordinate 8. The offensive feature which is shown in Figure 6.2, helps a piece to get closer to the goal condition. However, the opponent has a piece "o" located at x-coordinate

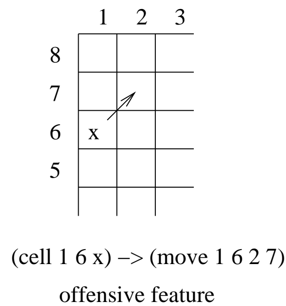

Figure 6.2: An offensive feature.

3 and y-coordinate 8 that can capture piece “x” after the offensive-feature move is made. GIFL does not contain any information about the opponent’s moves, thus the offensive-feature move will get the same amount of exploration with the move (move 1 6 1 7) from x-coordinate 1 and y-coordinate 6 to x-coordinate 1 and y-coordinate 7, which leads to a forced win.

To prevent spending more time on a move that can be countered easily, GIFL features can be extended to include information about the opponent’s counter moves as shown in Figure 6.3. The algorithm will reduce the score of the offensive-feature move with the information contained in the new offensive feature if the opponent has a defensive-feature move to counter.

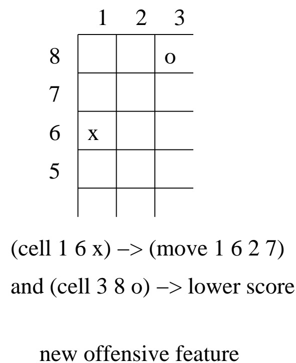

Figure 6.3: An example of the extended GIFL feature.

- GIFL can only learn features from a game sequence if the player that wins the game makes the last move. The learning algorithm cannot be applied to the games when the losing side makes the last move. Lose checkers is an example of this type of game. The players aim to lose all the pieces instead of trying to capture them. To handle games where the losing side makes the last move, the order of learning for the features should be changed. Instead of learning an offensive feature first, the player learns a defensive feature to not to lose the game. However, the basic idea behind GIFL stays the same.

6.2 Future Work

GIFL is suggested as a stepping stone to creating more effective domain-independent learning algorithms. There remains a lot of work to be done, but a few research extensions to GIFL come to mind. The improvements suggested in this section are not as easy to implement as in the previous section and have to be investigated further.

- Feature refinement is an important process of game playing programs. However, there has not been much research done on autonomous feature refinement because the programmer manually selects the features for game-specific programs. GIFL does not do any feature refinement and uses all the learned features. Feature refinement is a possible extension to GIFL to prune the features. This will lead to better results in two ways: it will increase the speed of the program because there will be less features to look in a state and it will provide more exploration to the moves that are potentially more helpful.

  Feature refinement may be done in GIFL by keeping an average score of the simulations in which the feature is used. At the end of a simulation, the scores of each feature that is used in that simulation can be updated. After a number of simulations, the features with low score can be discarded. This would speed up the search.

- The learning concepts of GIFL are heavily dependent on the terminal conditions. If the goal conditions of a game are too specific, the features may not be encountered frequently during the gameplay. Thus, GIFL may not be effective. For instance, the terminal conditions of chess has many variations depending on the position, number and type of pieces. GIFL learns one of these variations at each step of the algorithm. The occurrence of that specific terminal position during a simulation is necessary for the learned feature to be used.

  To solve the problem of learning specific terminal conditions. The learning data can be increased to cover more terminal state variations, but this will be time consuming and may not be very helpful because the learning data is generated randomly. Another way of solving this problem is generating terminal state variations from a terminal state. Figure 6.4 shows a terminal state from breakthrough. The terminal condition for player-X is to have a piece located at y-coordinate 8. New terminal states can be generated from Figure 6.4 by substituting different coordinate positions for the piece "x" and checking if the state is terminal or not. Figure 6.5 shows some of the generated terminal states.

  Each different terminal state variation can lead to different features. This will reduce the need for a large set of training data. However, having a large number of features may not help the performance of the program. The number of features may be large enough that the feature matching is too expensive to calculate.

  To do more efficient feature matching, abstract features can be introduced. Features that are similar can be abstracted into one feature. This will lead to a less expensive feature matching.

Figure 6.4: A terminal state.

Figure 6.5: Generated terminal states.

Figure 6.6 shows a variety of features and Figure 6.7 shows the abstract feature that can be learned from those variations.

(cell 2 7 x) –> (move 2 7 1 8) (move 2 7 2 8) (move 2 7 3 8)

(cell 1 7 x) –> (move 1 7 1 8) (move 1 7 2 8)

(cell 8 7 x) –> (move 8 7 7 8) (move 8 7 8 8)

Figure 6.6: A number of GIFL features.

	1	2	3
8			
7		x	o
6		o	
5	o	x	

(cell ? 7 x) –> (move ? 7 ? 8)

Figure 6.7: An abstract new GIFL feature.

Bibliography

[1] Michael Buro. From simple features to sophisticated evaluation functions. In *CG '98: Proceedings of the First International Conference on Computers and Games*, pages 126–145, London, UK, 1999. Springer-Verlag.

[2] Murray Campbell, Joseph Hoane, and Feng Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[3] George Casella and Edward I. George. Explaining the Gibbs Sampler. *The American Statistician*, 46(3):167–174, 1992.

[4] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. In *IJCAI'91: Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

[5] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139, 2007.

[6] Tom Fawcett. Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12:42–64, 1996.

[7] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI'08: 23rd National Conference on Artificial Intelligence*, pages 259–264, 2008.

[8] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, December 2006.

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

[10] Stanford Logic Group. http://logic.stanford.edu/.

[11] Levante Kocsis and Csaba Szepesvri. Bandit Based Monte-Carlo Planning. *ECML*, pages 282–293, 2006.

[12] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[13] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *AAAI'06: 21st National Conference on Artificial Intelligence*, pages 1457–1462. AAAI Press, 2006.

[14] Robert Levinson and Richard Snyder. Adaptive Pattern-Oriented Chess. In *AAAI'91: 9th National Conference on Artificial Intelligence*, pages 601–606, 1991.

[15] Makoto Miwa, Daisaku Yokoyama, and Takashi Chikayama. Automatic construction of static evaluation functions for computer game players. *Discovery Science*, pages 332–336, 2006.

[16] Barney Pell. METAGAME: a new challenge for games and learning. Technical Report UCAM-CL-TR-276, University of Cambridge, Computer Laboratory, 1992.

[17] A. L. Samuel. Some studies in machine learning using the game of checkers. *Computers & Thought*, pages 71–105, 1995.

[18] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

[19] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. CHINOOK: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[20] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *AAAI'07: 22st National Conference on Artificial Intelligence*, pages 1191–1196. AAAI Press, 2007.

[21] Shiven Sharma, Ziad Kobti, and Scott Goodwin. Knowledge generation for improving simulations in UCT for general game playing. In *AI 2008: Advances in Artificial Intelligence*, pages 49–55. 2008.

[22] Nathan Sturtevant and Rich Korf. On Pruning Techniques for Multi-Player Games. In *AAAI*, pages 201–207, 2000.

[23] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[24] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, 1995.

[25] Michael Thielscher. Flux: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5(4-5):533–565, 2005.

[26] P. E. Utgoff. ELF: An evaluation function learner that constructs its own features. Technical report, Amherst, MA, USA, 1996.

Appendix A

Constants

When using the features for UCT playouts, the value of the move which determines the probability of taking the move is determined by $C^{level-1}$. The constant, C, in the formula is determined experimentally from a small sample of games. Here are the experiment results:

name	0.3	0.6	0.9
game2	30.0 %	60.0 %	**70.0 %**
knightthrough	30.0 %	65.0 %	**75.0 %**
checkers	40.0 %	50.0 %	**55.0 %**
chess	30.0 %	**45.0 %**	38.0 %

Table A.1: The constant, C, is used to determine the move values

The probability of taking a move is determined using the Gibbs Distribution when using the features during the UCT playout:

$$p(a) = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}}$$

The constant, τ, tunes the balance between exploration and exploitation in the Gibbs Distribution. It is also determined experimentally from a small sample of games. Here are the experiment results:

name	0.1	0.5	0.9
game2	65.0 %	**70.0 %**	40.0 %
knightthrough	**80.0 %**	75.0 %	35.0 %
checkers	50.0 %	**55.0 %**	35.0 %
chess	35.0 %	**38.0 %**	30.0 %

Table A.2: The constant, τ, is used in the Gibbs Distribution formula