Feature Learning Using State Differences

Mesut Kirci and **Jonathan Schaeffer** and **Nathan Sturtevant**
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{kirci,nathanst,jonathan}@cs.ualberta.ca

Abstract

The goal of General Game playing (GGP) can be described as designing computer programs that can play a variety of games when given a game description. Learning algorithms have not been an essential part of all successful GGP programs. This paper presents a feature learning approach, GIFL, for 2-player, alternating move games in GGP using state differences. The algorithm is simple, robust and improves the quality of play.

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. However, researchers have focused on techniques for playing specific games very well rather than creating more intelligent programs. This choice gives rise to programs that can play a specific game very well like Deep Blue (Campbell, Hoane, and Hsu 2002), Chinook (Schaeffer et al. 1996) and TD-Gammon (Tesauro 1995). However, these programs cannot play other games. More importantly, most of the analysis and design is done by the programmer. Thus, these games have limited value for gaining insights into generally-applicable AI (Pell 1992).

General Game Playing (GGP), where programs aims to play more than one type of game, is used as a testbed for Artificial Intelligence and requires more general intelligence (Genesereth, Love, and Pell 2005). General game players accept game descriptions as inputs at runtime, analyze them, and then play the games without human intervention. Thus, general game players cannot use algorithms specific to a particular game and must rely on the intelligence of the program rather than the modifications by the programmer. Also, general game players should be able to play different classes of games, such as varying the number of players, simultaneous or alternating action games, games with small number of states, and games with large number of states (Genesereth, Love, and Pell 2005).

Current GGP programs perform search using UCT (Kocsis and Szepesvri 2006) or alpha-beta search algorithms. Programs that use UCT do not need an evaluation function because they simulate a game until a terminal node is reached. Cadia Player (Finnsson and Björnsson 2008) is an example of a program that uses UCT; it has won the last two GGP competitions. Cadia Player only uses move-based history heuristics to guide the UCT simulation. The other approach is alpha-beta pruning with an evaluation function. FluxPlayer (Schiffel and Thielscher 2007) and Clune Player (Clune 2007) 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 (Kuhlmann and Stone 2006) uses automatically extracted features to calculate the evaluation function. These are simple features like the number of pieces on the board and the number of legal moves. Clune Player was the winner of the first GGP competition in 2006.

In addition to the approaches mentioned above, a new learning method has recently appeared (Sharma, Kobti, and Goodwin 2008). This approach uses Temporal Difference learning. It learns a domain-independent knowledge base and uses the knowledge base to guide the UCT search. This technique has not been tested in competition and the experimental results have shown slight improvements for some games.

The success of all programs mentioned can be increased by improving the search. However, domain-independent knowledge extraction is a very hard problem. The results of the last two competitions reflect this. UCT, which is less dependent on knowledge, has been very successful. The approach described in the paper is called *GIFL*, Game Independent Feature Learning. The algorithm learns useful information and uses it to do more intelligent search in two player, alternating move games. It learns features, similar to the well-known history heuristic, using state differences in 2-ply game trees. After that, learned features are used to guide the otherwise 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 AI literature *feature* is described as a subset of state instantiated with values. The term feature is used differently in this paper. It is a knowledge chunk consists of a subset of state as in the general description of the feature and moves.

Therefore, the features of the GIFL are more general than regular features and include moves. In this paper, the term feature represents the GIFL features from this point on.

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

After features are learned, features 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 Sampling (Casella and George 1992).

The algorithm has been tested on fourteen different games and has performed well in most of them. The algorithm outperforms the standard UCT completely in eight games, does not effect the results in three games, loses slight advantage to UCT in 2 games and loses badly to UCT in only one game.

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 toward the next state. GIFL first learn 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 1. 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. The algorithm can learn two types of features, one from the middle state and one from the root state.

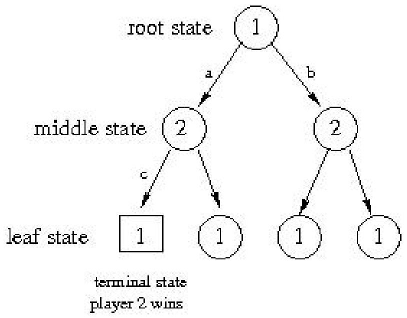


Figure 1: 2-ply game tree at the end of the game sequence

The move made at the middle state, Figure 1-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.

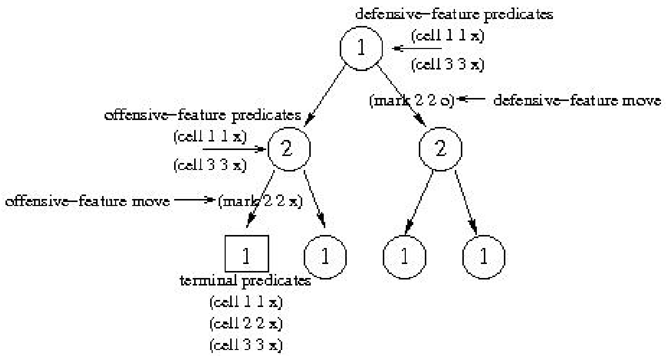


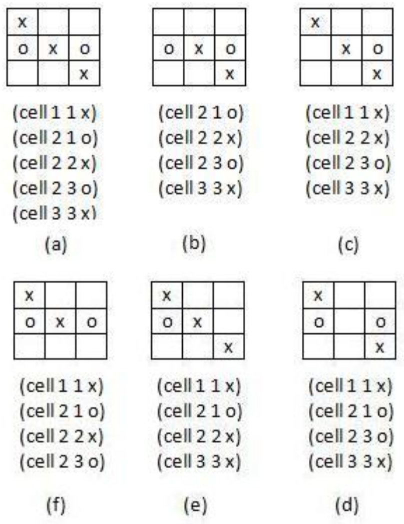
Figure 2: 2-ply game tree and the features can be learned from that tree

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 1-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 Figure 1-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 state, the defensive-feature move will not always prevent a loss at every state. 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 2. 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 to increase the generality. For instance, the terminal state in Figure 2 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-b, after we remove the first predicate of (a), the state is not terminal. Therefore, the predicate (cell 1 1 x) is a terminal predicate. In Figure 3-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).

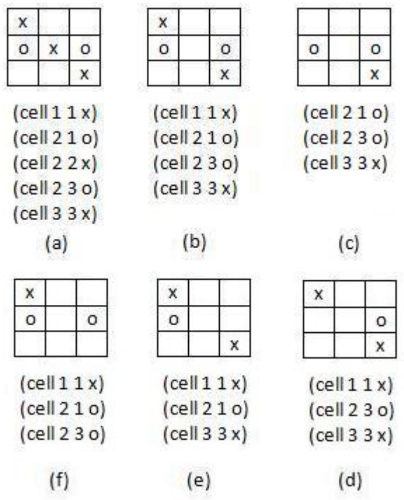


(cell 1 1 x) (cell 2 1 o) (cell 2 2 x) (cell 2 3 o) (cell 3 3 x)
(a)
(cell 2 1 o) (cell 2 2 x) (cell 2 3 o) (cell 3 3 x)
(b)
(cell 1 1 x) (cell 2 2 x) (cell 2 3 o) (cell 3 3 x)
(c)
(cell 1 1 x) (cell 2 1 o) (cell 2 2 x) (cell 2 3 o)
(f)
(cell 1 1 x) (cell 2 1 o) (cell 2 2 x) (cell 3 3 x)
(e)
(cell 1 1 x) (cell 2 1 o) (cell 2 3 o) (cell 3 3 x)
(d)

Figure 3: Finding terminal predicates. (a) is terminal state, (b) is the state after removing the first predicate, (c) is the state after removing the second predicate, (d) is the state after removing the third predicate, (e) is the state after removing fourth the predicate and (f) is the state after removing the fifth predicate.

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.



(cell 1 1 x) (cell 2 1 o) (cell 2 2 x) (cell 2 3 o) (cell 3 3 x)
(a)
(cell 1 1 x) (cell 2 1 o) (cell 2 3 o) (cell 3 3 x)
(b)
(cell 2 1 o) (cell 2 3 o) (cell 3 3 x)
(c)
(cell 1 1 x) (cell 2 1 o) (cell 2 3 o)
(f)
(cell 1 1 x) (cell 2 1 o) (cell 3 3 x)
(e)
(cell 1 1 x) (cell 2 3 o) (cell 3 3 x)
(d)

Figure 4: 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) is the middle state after removing the second predicate, (e) is the middle state after removing the third predicate and (f) is the middle state after removing the fourth predicate.

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 4, 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 predicates (cell 1 1 x), (cell 2 2 x) and (cell 3 3 x). In Figure 4-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 4-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).

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

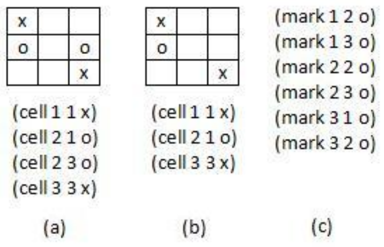


Figure 5: Finding the defensive-feature moves. (a) is the middle state, (b) is the root state and (c) are the legal moves at the root state.

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 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 breakthrough game, 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 5-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 are the predicates that would 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-feature moves), the predicates for the defensive feature are the intersection of these sets because there may be some other requirements for each specific move. However, the intersection eliminates specific predicates for each different move which are not part of the defensive feature. For instance, assume that there are six legal moves at the root state of the 2-ply game tree under examination and two of the legal moves are possible defensive-feature moves. A set of defensive-feature predicates is found for each of the remaining four legal moves. Each of them may contain predicates that are required for the move to be legal at the state, but all of the four 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 four sets of defensive-feature predicates eliminates the move-specific predicates and makes the defensive feature more general.

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.

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 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. 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

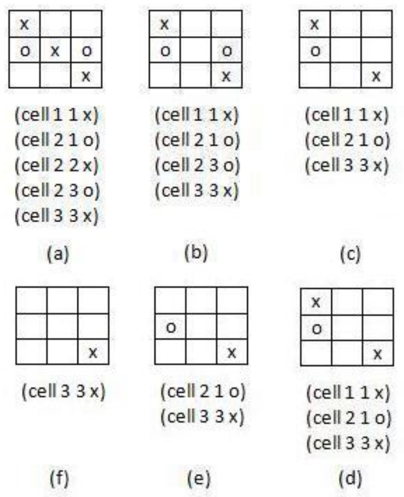


Figure 6: Finding the new 2-ply tree for further learning. (a) is the previous leaf state, (b) is the previous middle state, (c) is the previous root state, (d) is the new leaf state, (e) is the new middle state and (f) is the new root state.

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 assigned to the one higher state of the new middle state regardless of its suitability to defensive learning. An example of finding the new 2-ply tree in tictactoe is presented in Figure 6.

In the example, the root state becomes the new leaf state of the new 2-ply tree. Also, the state at Figure 6-e is the middle state since not all of the new leaf predicates are present in that state. Therefore, the move made to reach the leaf state have some contributions and an offensive feature can be learned.

The pseudecode for feature learning process is presented in Figures 7 and 8. 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 specified by the third parameter. The function *createFeatureUsingStateFacts* finds the feature predicates for both offensive and defensive features. The predicates and moves that the features use are the parameters of this function because the necessary predicates are found from whether or not leaf state of the 2-ply tree can be reachable from root state.

Implementation Details

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

First, the algorithm runs a number of simulations using random move selection and analyzes 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, after offensive-feature predicates are found using the algorithm in Figure 8, 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 some of the offensive-feature predicates missing and the feature cannot satisfy the leaf conditions. This type 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 is useful or not. If it is useful, it is added to the feature base.

Using Features

Features are used to guide the random simulation in a UCT search. The program checks each state during a simulation to see if a feature can be applied or not. If the predicates of a feature is 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 Sampling. This will bias toward the random simulation with the expectation of achieving a more accurate result instead of doing pure random simulation.

If features are used in a random playout, all available moves are assumed to have no value. The value of a move can be changed if a feature with the that move can be applied in a state. 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. If there are any matches in the feature map for a certain 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. Then, the value of the move is set according to the formula $100 * C^{level-1}$ where C is the 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.

After all of the possible features are matched, the move is selected according to the Gibbs Sampling 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 100, 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 Sampling. The move is selected according to the probabilities calculated. The Gibbs Sampling 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 opponent respond. Therefore, other possible moves are explored. This exploration-exploitation problem is very common in Reinforcement Learning and the Gibbs Sampling is one f the techniques which are used to tackle it.

In addition, a probability of using features for each player is introduced as a part of an opponent modeling technique. 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 the information it does not have.

The pseudecode for how to use the features in UCT search is presented in Figure 9.

Experiments

The experiments were prepared using the game definitions presented at the Stanford GGP repository and game definitions used in the GGP competition 2008. The games that are used in the 2008 competition are named arbitrarily like game1, game2, etc. All games are 2-player, alternating move and prefect information. Also, in some games being first player or second player may be advantageous. Therefore, experiments are conducted so that this does not effect the results.

The player that uses the features to guide the random simulation is compared against a UCT player with random simulation and is called as the learning player. Therefore, the only difference between the learning player and the non-learning player is that learning player uses the learned features to guide the random simulation phase during the UCT search.

The number of simulations per move is limited due to time constraints and same for both players. This shows the effectiveness of the learned information without worrying about the computation time needed to match the features. Furthermore, the probability of using features in random simulation is introduced so that the learning player can exploit the fact that the opponent does not have the same knowledge. Learning player uses the features all the time and the opponent uses features only half of the time during the random simulation phase of the UCT.

Games from Stanford GGP repository and 2008 GGP Competition

name-n. of simulations-n. of games	learning-uct	win percentage
(2008 competition) game2-1000-20	193-7	97.0 %
knightthrough-1000-20	184-16	92.0 %
(2008 competition) 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 %
(2008 competition) game5-1000-40	111-94	55.0 %
pentago-1000-100	100-100	50.0 %
quarto-1000-100	98-102	49.0 %
(2008 competition) game6-1000-100	96-104	48.0 %
(2008 competition) game3-1000-100	91-109	45.0 %
checkersbarrelnokings-1000-100	61-139	30.0 %
(2008 competition) game4-1000-40	100-100	-

Table 1: Effectiveness of using GIFL

The number of training runs is limited to 500 unless specified otherwise. The learning time may vary between 100 training runs per minute in breakthrough and 20 training runs per minute in checkers. The level of 2-ply tree in which the learning is occurring is limited to 3. This reduces the number of the features and the time spend in random simulations. The number of tests run is specified with the name of the game in the table below. It 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 as both sides. Games with close results are usually tested with more games.

The results are promising as shown in the Table 1. The number of test games and the number of simulations per move are shown with the name of the game. Of the 14 games that the player is tested on, the learning player defeats the non-learning player in 8 of the games. The knowledge does not affect the results in 3 games. In 2 games, the learning player loses by a small margin. Using knowledge has a poor result in only one game, checkersbarrelnokings. 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 Game5 and all games end in less then 10 moves in Quarto.

Although the checkesbarrelnokings game is similar to the original checkers at which learning player has a clear advantage, the learning player loses badly. Due to the lack of kings and forced jumps in the checkersbarrelnokins game, the number of legal moves per step is low. Therefore, the non-learning player does less unnecessary exploration. This may be the cause of losing advantage of the learning player against non-learning player.

As it can be seen in the Table 1, the learned features are clearly effective. However, in actual game play the computation time is also a factor. To measure the difference in number of simulations per move, another experiment has been performed with fixed time, 30 seconds per move. The results show that in half of the games, the computation time is not a big factor, but in 5 games the learning player can only make $1/3$ of the simulations that regular UCT can make.

Games from Stanford GGP repository and 2008 GGP Competition

name	n. of simulations (learner/uct)
(2008 competition) game2	46 %
knightthrough	93 %
(2008 competition) game1	104 %
breakthrough	79 %
checkers	36 %
connect4	20 %
chess	74 %
(2008 competition) game5	32 %
pentago	156 %
quarto	34 %
(2008 competition) game6	58 %
(2008 competition) game3	99 %
checkersbarrelnokings	38 %
(2008 competition) game4	47 %

Table 2: Computational cost of using GIFL

However, the quality of game does not suffer as much. The two games in which using features hurt most are Connect4 and Checkers. In 30 seconds fixed time settings, the learning player still wins 60% percent of time in Checkers and the learning player only loses 55% percent of time in Connect4 with only $1/5$ of simulations that regular UCT makes.

Table 2 shows the number of simulations that the learning player can make when given the same amount of time as regular UCT. The number is expressed as a percentage of the UCT result. For example, in Connect 4, learning slows the program down by a factor of 5 – to 20% the speed of UCT. 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.

Conclusion

The learning algorithm learns predicate-move combinations and uses them to guide random UCT simulation. The concepts are simple and domain independent which is essential for GGP algorithms. Including the 2008 GGP competition, learning algorithms have not been an essential part of a successful GGP program because domain-independent learning is a very hard problem. However, this paper presents a simple but effective method that shows very promising results in some of the games that are frequently used in GGP competitions.

The algorithm shows promising results in GGP, but the learning concepts are heavily depended on the terminal conditions. If the goal conditions of a game is too specific, the features may not be encountered frequently. 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 effective. However, most of the GGP games in which GIFL is successful, have less number of different possible terminal conditions. In conclusion, the effectiveness of GIFL depends on how many variations terminal conditions of a game can have.

In addition, the computation time problem when using features is important for the future. The primary focus of GIFL is the effectiveness of the features, therefore time has not been spent to develop more efficient ways of feature matching and feature pruning. Some of the learned features may not be effective and can be removed. This will help with the computation time problem. Also, it should be mentioned that the effectiveness of the features is not proportional to the number of simulations as it is shown in the experiments with Connect4.

The algorithm has room for improvements. First, the features can be used as a part of an evaluation function. A minimax approach can be tried with this evaluation function instead of the UCT search.

Second, the algorithm 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 games when the losing side makes the last moves. Lose Checkers is an example of these types of games. The players aim to lose all the pieces instead of trying to capture them. This problem may be solved by changing the leaf of the 2-ply tree where the learning occurs.

In addition, the frequency of features seen in the learning process can be included when the values for the feature moves are calculated. Right now, all of the features have the same importance.

References

Campbell, M.; Hoane, J.; and Hsu, F. 2002. Deep blue. *Artif. Intell.* 134(1-2):57–83.

Casella, G., and George, E. I. 1992. Explaining the gibbs sampler. *The American Statistician* 46(3):167–174.

Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*, 1134–1139.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*, 259–264.

Genesereth, M. R.; Love, N.; and Pell, B. 2005. General game playing: Overview of the aaai competition. *AI Magazine* 26(2):62–72.

Kocsis, L., and Szepesvri, C. 2006. Bandit based monte-carlo planning. *ECML* 282–293.

Kuhlmann, G., and Stone, P. 2006. Automatic heuristic construction in a complete general game player. In *AAAI*.

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

Schaeffer, J.; Lake, R.; Lu, P.; and Bryant, M. 1996. CHINOOK: The world man-machine checkers champion. *AI Magazine* 17(1):21–29.

Schiffel, S., and Thielscher, M. 2007. *Automatic Construction of a Heuristic Search Function for General Game Playing*.

Sharma, S.; Kobti, Z.; and Goodwin, S. 2008. Knowledge generation for improving simulations in uct for general game playing. In *AI 2008: Advances in Artificial Intelligence*. 49–55.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Commun. ACM* 38(3):58–68.

```
trainPlayer(currentState, knowledgeBase, levelLimit)
stateList ← generate a game sequence
terminalPredicates ← find the terminal predicates
/* the 2-ply tree used in learning */
state leaf = stateList(terminal)
state middle = stateList(terminal-1)
state root = stateList(terminal-2)
leafPredicates = terminalPredicates
while (level <= levelLimit)
    /* OFFENSIVE FEATURE DISCOVERY */
    middleAction ← action made to reach leaf
    createFeatureUsingStateFacts(middle,middlePredicates,
    leafPredicates,leafAction,middleAction,winner,
    rootPredicates,rootAction)
    if feature is useful
        add to knowledge base
    else
        createFeatureUsingTerminalPredicates(middle,
        middlePredicates,leafPredicates,middleAction,winner)
        if feature is useful
            add to knowledge base
    /* DEFENSIVE FEATURE DISCOVERY */
    vector possibleRootActions
    do
        createFeatureUsingLegalActions(root,
        possibleRootActions,leafPredicates,
        leafAction,middlePredicates,middleAction,loser)
        if possibleRootActions.size() == 0
            middle = middle - 2
            root = root - 2
            if (not contains(getStateVector(middle),
                        middlePredicates))
            ||
               canPreventReachLeaf(middle,
                  middleAction,leafPredicates,
                  leafAction,winner)
                        stop learning
    while(possibleRootActions.size() == 0)
    for all moves except possibleRootActions
        /* find necessary predicates */
        createFeatureUsingStateFacts(
            root,possibleRootPredicates[i],
            middlePredicates,middleAction,
            possibleWrongAction,loser,
            leafPredicates,leafAction)
    rootPredicates ← intersection(possibleRootPredicates)
    add to knowledge base
    /* FIND NEXT LEAF, MIDDLE, ROOT, */
    do
        leaf = root
        middle = middle - 2
        root = root - 2
    while(contains(getStateVector(middle),
                    middlePredicates))
    leafPredicates = rootPredicates
    leafAction = rootAction
    /* clear middle, root predicates and actions */
    level++
end while
```

Figure 7: The learning algorithm.

```
createFeatureUsingStateFacts(state middle,
vector middlePredicates, vector leafPredicates,
leafAction, action, player,
vector rootPredicates, rootAction)
    temp = middle
    middleStateVector ← predicates of middle
    for all predicates in temp
        remove one by one,
        new state is reducedTemp
        if isLegal(reducedTemp,action)
            reducedTemp.performMove(action)
            stateVector ← predicates of reducedTemp
            if (not contains(getStateVector(middle),
                        middlePredicates))
            ||
               canPreventReachLeaf(middle,
                  middleAction,leafPredicates,
                  leafAction,winner)
                        middlePredicates.add(predicate)
        else
            middlePredicates.add(predicate)
```

Figure 8: The function to find feature predicates.

```
DoMonteCarloSimulation(state currentState)
if features are not used
    do random move selection
if features are used
    vector statePredicates ← predicates of state
    for each predicate in the statePredicates
        features = knowledbase[predicate];
        for each feature in features
            if contains(statePredicates,feature(predicates))
                if isLegal(feature(move))
                    moveValues[feature(move)] $= 100 * C^{level-1}$
    if there are moves with value 100
        select move between them
    else
        gibbsSampling(probabilities,moveValues)
        selectMove(probabilities)
```

Figure 9: The algorithm to use the features in the UCT search.