Determining move effects without explicit state construction

Construction and Implementation of a GGP Agent

TECHNISCHE UNIVERSITÄT DARMSTADT

To
Prof. Fürnkranz
Data and Knowledge Engineering
TU Darmstadt

from
Christian Wirth
Mat-Nr.: 1274437
Fachbereich 20

Abstract

The UCT algorithm is currently the most successful General Game Playing algorithm available. Its Monte Carlo simulation behaviour is very speed dependant, which is the reason why the next state selection is only depending on the current state and the available moves, but not the next states. A fast method to calculate the difference between those next states and the current state would allow improving UCT by introducing some complex selection rules, the way heuristic agents are using them, as an enhancement for the UCT selection. This thesis is presenting a new method to determine those differences faster than by calculating the next states and comparing them. The main idea is to use the information collected by the legal-moves-calculation together with a precalcuated limitation of the possible effects. The presented method is also implemented into a GGP agent to test the improvements in real scenarios.

Declaration in lieu of oath

I hereby declare in lieu of oath that I composed the following thesis independently and with no additional help other than the literature and means referred to. This thesis has never been made partially available, in this or any other form to any audit board for the purpose of obtaining a degree.

Darmstadt, 28.9.2010

Acknowledgements

Prof. Johannes Fürnkranz, my supervisor for giving me the opportunity to write this thesis as well as helpful input.

Angela Eigenstetter, a fellow student for proof reading the thesis.

Felix Reichert, a long time friend for also proof reading this thesis.

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

Index

1. Introduction

Modern Computers have proven to be very good problem solvers. Many aspects of our daily life as, we know it, would not be possible, without these solving capabilities. Route planning, time management and global communication are only a few examples, but they have one thing in common: They are mathematical problems. As long as the definition of a problem can be mapped to functions and values, it can be solved quickly.

However nowadays, the performance of computers have risen so much, that we can hope for something more: Universal problem solving. This means that the computer can solve a problem that is defined in a non-mathematical representation: A first, major step towards an artificial intelligence as already imagined in literature and movies. An intelligence that is capable of universal reasoning, better then any human. Of course we are not anywhere close to this goal, but progress is being done. One of the approaches to the problem is simplifying it and thus trying to solve this. “General Game Playing” is one of these simplifications. It presents abstract and real (board-) games that will be played by artificial agents without any prior knowledge hence it can be used to determine the efficiency of algorithms to adapt to new problems. To reach the goal of universal problem solving, an algorithm that can be used for any problem is needed. It should be fast and universal but also be able to make use of the domain-dependent information that is coming along with the problem to be able to reason about the problem. Currently two different approaches are used: On the one side are domain independent algorithms that are calculating a high amount of simulations to collect enough data for the given problem to converge to the best solution. On the other side are heuristic algorithms requiring domain knowledge, but if working right they can determine a good solution from this knowledge. Despite the more human-like approach of heuristic agents, the simulation based algorithms are currently in lead when searching for the best approach to General Game Playing. Especially the UCT algorithm, based upon the Monte-Carlo simulation scheme, resulted in a major step forward for GGP. It converges fast enough towards a good solution to beat heuristic algorithms, even in several domains where the heuristic is considering the correct information for its doing. In chapter 1, an overview of the currently available GGP agents and how they work is given. Chapter 2 is presenting the fundamentals of the UCT algorithm and GDL, the language for GGP.

When trying to enhance GGP agents further, a system which makes use of the advantages of both systems would be next logical step but this is hardly possible because UCT is very speed dependent and heuristic approaches tend to be slow. A major problem for those hybrid approaches is the determination of which game states to visit. Simulation based algorithms are selecting those states at random, only guided by the already available data. Heuristic algorithms on the other hand are comparing several states to select the best one. This calculation and comparison of next states is very time consuming and should therefore not be used for the UCT algorithm. To overcome this problem, this master thesis is presenting a scheme that is calculating the difference between the current game state and any following without creating the next state. It is solely based upon the already known game state and the possible moves that can lead to other states, combined with some precalulated restrictions to increasing the speed. How this works, and what has been done to get the system up to speed is explained in chapter 3.

To be able to compare the new approach to the already available GGP agents, it has been implemented into a player. To ease those implementations, frameworks and reasoning systems for GDL are available. Chapter 4 is describing this implementation and which changes have been applied to the base framework, with the results and conclusion of this new agent being discussed in chapter 5.

1.1 General Game Playing

General game playing (called GGP) is a standardized way to represent games that aims at providing rule sets that are computer-readable, but are not limited to specific games. To achieve this, it uses a Datalog dialect called “Game Description Language” (in short: GDL) [28]. Datalog is a powerful language already that is used to describe data and rule sets. It can also be expanded to a more general problem description language [7], therefore it is a good starting point for creating a universal problem description language.
At first, playing games seems to be differing a lot from universal problem solving, but in reality it is only a higher level grammar on the Chomsky hierarchy [8] that could be extended to a lower level.
The properties of the current GDL version are [28]:

- The rules have to be deterministic.
- The games have to be complete information games, meaning that all information is revealed to the player.
- Non zero-sum games are possible.

The current version of GDL is limited, of course, but it already requires a completely different type of programs than specific problem solvers like chess agents: GGP agents can not utilize domain specific knowledge, because the domain of the game is not known before playing. To get an even more general approach, all GGP based contests, known to the author, are submitting the rules only directly before the match is started, meaning there is only a very hort time frame where it is possible to retrieve domain information. General game playing is seen as one of the grand AI challenges of our time. [19]

2. Tree Search

The common solution to general game playing is tree search. Every game can be modelled as a tree, with the nodes representing a game state and the edges are the connection as moves. For most games, this is a graph and not a tree, but by including the depth and the move that has lead to the state, it becomes a tree. An example for such a tree is given with figure (1). In theory, it is also possible to work with a graph, but this method is a fairly new and uncommon approach and will not be considered in this master thesis. [27]
Most games are only defining goal values for terminal states, like real board games. It is not possible to make an analysis of who will win while the game is still running. Of course a trained human can make an educated guess, but that is not a mathematically exact analysis, hence we also have to make such a kind of guess or we need to play several games to see which moves or patterns are beneficial for the current game: We need to find the most promising path within the game tree, that will most likely maximize our goal value. But the game tree is too large to fit into memory or even get computed as a whole in an acceptable time. Hence it is required to decide which parts of the tree to search for those promising paths.
To overcome this problem, there are several possibilities:

1. Deterministic: Exploring the game tree in an ordered fashion and selecting the most promising path.

2. Heuristic: Estimating the value of non terminal nodes by state samples retrieved by the search.
3. Simulation: Getting an average over multiple played games and selecting the best average move.

Those techniques are described in the following chapters. For completeness, it should be mentioned that there also some other, rarely used techniques like evolutionary algorithms, but they are also a kind of a heuristic (6).

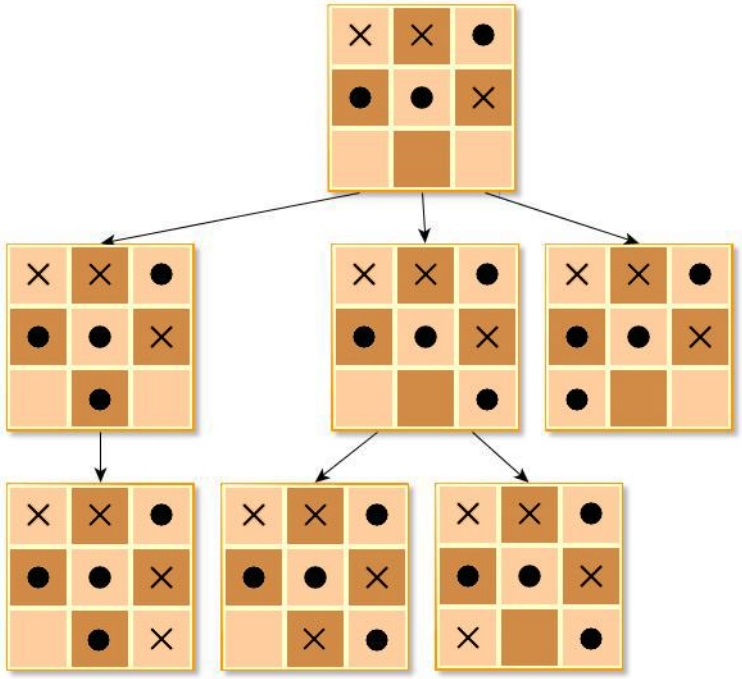


Figure 1: Example game tree from „Tic Tac Toe“

2.1 Deterministic

Deterministic tree search is the oldest approach for solving the game playing problem. It simply traverses the tree in a given order and saves the outcome of every move in every state. This value is calculated with the assumption that the enemy will always select the move that minimizes our goal value and we are always selecting the maximizing move. To reach terminal states early, a depth-first search is used. This is called MiniMax Search [36].
However, because of the size of a game tree, this search is very inefficient, since it's only possible to visit small parts of the tree within the time constraints for a single move. To improve the performance of MiniMax Search, it is possible to cut off parts of the tree that can be guaranteed to not contain a path that is better than one already visited. This is achieved by maintaining upper and lower boundaries of the goal value for every game node. Because of the MiniMax assumption that the enemy will always pick the move that minimizes our goal value, it is possible to detect states that cannot increase the goal value. This is called Alpha-Beta Search or Pruning. [36]
Programs facilitating Alpha-Beta Search are improving the algorithm further, by introducing a heuristic component. This component is used to decide which node (of the unvisited child nodes of a state) will be visited next. This can be done by saving the average move outcome, without

considering the state (History heuristic [33]). But because a depth-first search is used, it is possible that moves in the current state are not getting visited, because the tree is too large.

Deterministic tree search achieves very good results for small game trees, because it can guarantee to return the best possible path of all seen paths. This search can be done in parallel, therefore it is possible to achieve good results even for large game trees when using powerful cluster computer. The well known Deep Blue chess computer was using this approach to reach a quality of game play that can best human grandmasters. [13] For general game playing this technique is rarely used, because it is very time consuming to calculate the next states in an abstract description language. Programs like Deep Blue are a lot faster, because they can exploit game properties to represent the states and apply moves to these states. Some GGP agents are using deterministic tree search components for single player games [14]. These trees tend to be a lot smaller and it is not required to sample as many states as possible, but a path that maximizes the goal value can be played straight away, since the path is not dependent on the moves of an enemy player.

2.2 Heuristic

The second possibility is the heuristic estimation of the goal value for non terminal states. To achieve this, abstract evaluation functions are introduced. These functions are trying to map features of the game states to calculate an estimated goal value for each state. Good features would always have similar values for states that are leading to a victory for the current player.
Some example features [9]:

- Mobility: Number of possible moves
- Payoff: Difference of the number of pieces per player on the board
- Longevity: Number of steps to a terminal state

But it is also possible to retrieve a set of feature candidates from the game description itself [35]. These feature functions are now getting weighed to define which ones are good mappings of the game features for the current game. The heuristic is sampling as many states as possible, by traversing the tree, to estimate this weighing. Several heuristic game players are using stability as an indicator for the relevance of a feature. For Example, if a high mobility value often results in a good goal value, this feature is considered stable and therefore used to evaluate the states. [9]
This kind of search focuses on promising states, instead of promising paths, but the heuristic value of a state is also used to decide which paths to sample first.
Heuristic methods have the general problem that it is impossible to say if the selected features are able to estimate the goal values of a state. As an Example, neither mobility nor payoff would be a good function to evaluate Tic Tac Toe game states. The stability based weighing cannot overcome this drawback, but only reduce it. Another possibility is automatic feature extraction by trying to extract promising features out of the game description. It is already working a bit better, because the features are depending on the domain. But this is difficult and the feature count can become too large for efficient calculation of the evaluation function.
On the other hand, heuristic algorithms are converging much faster towards good solutions, because they only need to sample the game tree and not do exhaustive searching.

2.3 Simulation

Simulation based approaches also do not facilitate domain specific knowledge, like the deterministic approach would. They are a relatively new approach for general game playing, but already used for several specific games, like Go [6]. The basic idea is to just run a high amount of randomly played games for every move possible in the current state. The move with the best average outcome is made.
Even if the basic version looked promising, several improvements and performance enhancements were required to make it competitive [5]. These algorithms are known as Monte-Carlo Tree Search, because they are based upon the Monte-Carlo Simulation that is already used in various domains.

In recent years, a significant extension to Monte-Carlo was made: Upper Confidence Bounds applied to Trees or "UCT" for short. This is a possibility to guide the node selection to a better exploration/exploitation trade-off. Monte-Carlo is totally random and it is therefore possible that only "bad" moves are being selected; Moves that would never be picked in a real game, or moves that already have lots of samples while others have none. UCT ensures that unvisited or promising moves with few samples get selected first (-> Chapter 3.1). But because of the simulation character of the algorithm quite a few simulations are required for it to converge to the best solution. [26]

2.4 Comparison

To compare the performance of general game playing agents, results from several international contests are available. The best known is hosted by the Association for the Advancement of Artificial Intelligence (AAAI, 33) as an annual competition. It was first held in 2005, hence the results of various players are available. Table 1 shows the top three players for all AAAI competitions since 2007.

2010	2009	2008	2007
1. Ary	1. Ary	1. Cadiaplayer	1. Cadiaplayer
2. Maligne	2. Fluxplayer	2. Cluneplayer	2. Fluxplayer
3. Cadiaplayer	3. Maligne	3. Ary	3. Ary

Table 1: AAAI GGP competition results [10,16]

The most successful player till now is Ary, developed by Jean Mehat from the University of Paris. It uses an improved version of UCT that is optimized for speed. Because no knowledge discovery is used, the performance is the only relevant factor for the quality of this player. [29]
The second best player is CadiaPlayer, by H. Finnsson of the University Reykjavík. UCT is also the algorithm of choice here, but in contrast to Ary, several heuristic optimizations are used. The first is the average outcome of the goal value, without considering the state, as an approximation for the value of a move [33]. It is available in two variations that differ in the moves that are used to calculate the average. It is either all moves or only the moves within the UCT tree (->Chapter 3.1). A third heuristic is also using this move heuristic, but the facts valid in the current state are also

considered, meaning the average is not completely independent anymore. Only the fact/move pairs valid for the current state are considered to determine the average outcome of a move. When the simulation count is very low, another assumption is used: The outcome of a move played is also relevant for neighbouring states. Because of this assumption, the reached goal values are not only propagated to the states that have lead to the goal, but also to states that are one move distant. [14]
The newcomer Maligne from a Team of the Univerty of Alberta, is facilitating a hybrid approach. It is learning features from the game, like heuristic approaches, but they are not directly used for selecting moves, but for guiding a UCT algorithm. A feature in Maligne is learned from a state chain by extracting partial state information where a specific move is leading to a explicitly high goal value. An example is the learning of marking the 3rd cell in Tic Tac Toe, when two neighbouring cells are already marked. The features are separated into offensive and defensive features, where an offensive feature is leading to a high goal value for the current player. Defensive features are preventing the enemy from getting a high goal value. [25]
The FluxPlayer is the only real heuristic agent (besides the unsuccessful ClunePlayer) during the recent years AAAI competition winners. It uses an approach for automatic feature extraction that is based upon the fluent calculus, implemented as the “Flux System”. The extracted features are evaluated for every state and then combined by a fuzzy logic calculator to gain a goal value estimate. [35]

Even if UCT is currently “state of the art”, when using the AAAI contest as a quality meter it does not behave better then the other approaches in every game. Every approach has its advantages and disadvantages, like listed in table 2.

	Pro	Con
Deterministic	Domain Knowledge independent Complete Search	Can not facilitate Domain Knowledge Bad Exploration
Heuristic	Can facilitate Domain Knowledge Fast convergence to good solutions	Domain Knowledge dependent
Simulation	Domain Knowledge independent Good Exploration/Exploitation Tradeoff	Can not facilitate Domain Knowledge Randomness Slow convergence

Table 2: Comparison of GGP algorithms

Using domain knowledge is at the same time a big advantage and drawback at. If the right features are considered, it will beat most other approaches, simply because it is possible to estimate the goal values of non-terminal states. This allows to get conclusions over moves much faster, because it does not require several games played to the end and it is also more exact, because it is possible to determine good moves and not only good move chains. But if the wrong features are mapped, the algorithm will tend to use completely wrong paths. Deterministic approaches are a bit out of fashion, because they can only be used efficiently for small game trees or with long time frames for the calculation. Because GGP is not representing games in an easy calculate-able, domain specific representation, but in a universal Datalog dialect, those approaches are even worse. Some GGP agents are using this for playing single player games, because it can be assumed that these game trees are much smaller.

The simulation based approaches are currently leading the field, because the slow convergence drawback can be balanced out with computer power and the randomness with Gibbs sampling. Gibbs sampling uses the history heuristic [33] to select promising moves when no simulation information is available. It is still random, but guided by using a probability distribution. This is the case when selecting unvisited nodes or within the random expansion phase (-> Chapter 3.1). But Gibbs sampling depends on the move command to be a good feature for the games. This is where the approach of this master thesis tries to introduce improvements.

3 Fundamentals

To understand the idea and the problems, mentioned in this master thesis, it is required to understand the details of UCT, Gibbs Sampling and the GDL first. This chapter is dedicated to those fundamentals.

3.1 UCT - Upper Confidence Bound applied to Trees

The basic Monte-Carlo algorithm samples the game tree to learn good state/action pairs. It learns only from the results of simulations and does not require any domain specific knowledge. The actual move done by the MC Player is then the best action in the current state, according to the seen samples. To calculate these values, MC follows a given policy to select an action until it reaches a terminal state. The goal value of this terminal state is then propagated to the selected state action pairs that have lead to the terminal. To have a bias towards early goals, those goal values are decaying along the back propagation. The average value q of the state/action pairs can be implemented incrementally by maintaining a visit counter besides the mc value. This is represented in the formula (1) [15]:

$$Q(s,a)_t = Q(s,a)_{t-1} + \frac{1}{N(s,a)_t} \times (R_t - Q(s,a)_{t-1}) \quad (1)$$

N(s,a) is the visit count for the action a in the state s and R is the reward for the last simulated episode. Of course, N(s,a) can never be zero, because the visit count gets incremented before the goal value is propagated.

MC is guaranteed to converge to the best solution, but only when the tree gets sampled near exhaustively [23], but as mentioned, most game trees are far too large to be sampled exhaustive in a realistic time frame. Hence the policy selecting the next state in the simulation process affects the outcome greatly, because the function decides which parts of the tree are getting sampled. The outcome will only be a good one if the sampled part of the tree is representative for a good game. The simplest policy is just random but this is not guiding. Therefore MC facilitates a greedy policy for selecting the next state. The state/action pair with the currently highest value gets selected, but this leads to a behaviour that is very dependent on the actions selected first. If those actions are already promising, the algorithm will not simulate any other actions and therefore misses large parts of the game tree completely. [14]

UCT adds a second part to this greedy policy that ensures that actions with few samples are simulated first. A trade-off between exploration of unseen state action pairs and exploitation of already good pairs is maintained. UCT itself is the UCB algorithm from Auer, Cesa-Bianchi and Fischer [18] applied to trees. To the greedy selection part (1) a second function is added, where N(s) is the visit count for the given state and N(s,a) is the visit count for the state action pair, as in the MC algorithm. Cp is simply a constant that allows the fine tuning of the UCT bonus. The default value is $\sqrt{2}^{-1}$. [15, 26]

$$a_t = \arg\max_{a \in A_t} \{Q(s,a) + C_p \sqrt{\frac{\ln N(s)}{N(s,a)}}\} \quad (2)$$

As long as unvisited actions are available, UCT defaults to select these actions. It should also be mentioned that states are including the depth in the tree, making them unique.
This extension overcomes the exploration-exploitation problem that arises because the game tree can not be computed in time, but the game tree is still too large to be stored in the RAM. HDD storage is not an option, because the access is far to slow. This second problem is reduced by not storing anything after the UCT border. The UCT border is the first unseen state that is selected by UCT. This state is then used as the start node for the simulation process, which does not save the seen states. Of course, because the states are not getting saved, it is not possible to use the UCT policy for action selection here. The selection function is either totally random or a weighted probability distribution created by Gibbs sampling (->Chapter 3.2)
The success of this approach was shown in the last two GGP Contests by the Cadia-Player [14]. It won the 2007 and 2008 competition.

3.2 Gibbs Sampling

When using UCT, in two tasks the action selection cannot be done by UCT, because the relevant values are not available:

- Selecting one out of several unexplored moves in the UCT phase
- Selecting the next move in the simulation phase

Totally random is a bad policy for these tasks, because the policy cannot reason over any assumptions. Hence the assumption used is: Actions that have been good in other states should also be good in this state. This is called the history heuristic [33], because it is a heuristic over the history of the selected actions. The program maintains a list of all sampled actions and their average outcome, propagated from the terminal states the same way as the state action pairs. To select a new action now, a probability distribution is created out of the available actions. The probability of an action is proportional to the average outcome of the action. Formula (3) is defining this probability distribution [15]. [38]

$$P(play(a)) = \frac{e^{Q_h(a)/\tau}}{\sum_{b=1}^{n} e^{Q_h(a)/\tau}} \quad (3)$$

$Q_h(a)$ is the average outcome for action a, with the parameter τ used to flatten the distribution. A value of zero would lead to a greedy behaviour, because the highest value is selected with probability 1. [15] This should be prevented, because it would lead to a partially deterministic behaviour.
The drawback of this heuristic is that it relies on the assumption that the action itself is a good feature to describe an action. This seems obvious; but it is not. When implementing a GGP agent

an action is only the command that invokes the action. For example, “move 3,3,4,6” would be an action in chess, but it does not describe the piece moved or if an enemy piece is beaten by this move. (->Chapter 3.3 and Chapter 4.1)

3.3 GDL – Game Description Language

The “Game Description Language” is the Datalog dialect used to describe games. It is used to define all information required for a complete game. This is a set of base facts, like successor relations, board cells or piece counts and a set of rules for transitions and game mechanics.
GDL has only few predefined predicates that are known beforehand. These are [28]:

- init(X): The fact X is true in the initial game state.
- role(X): the constant X is a player role
- true(X,S): the fact X is true in the game state S
- legal(P,X,S): the move command X is legal for player P in state S
- does(P,X): player P has selected move command X
- next(X,S): the fact X will be true in the successor state of S
- terminal(S): true when S is a terminal state
- goal(V,P,S): player P has reached goal value V in the (terminal) state S

All other predicates and facts have no common definition, meaning it is not known which predicate defines a cell or a successor. To make sure that it is also not possible to guess the function of a predicate by its name, these predicates are scrambled.
It is not possible to describe *all* games with GDL, since it has some limitations. Those limitations are [28]:

- Determinism: Every legal state/move combination needs to have an explicit successor state.
- Termination: All possible move sequences are leading to a terminal state in a finite number of steps
- Playability: Every role has at least one legal move in every state reachable from the initial state
- Monotonicity: Every (terminal) state has exactly one goal value for every role and goal values do not decrease.
- Winnability: For every role exists a sequence of (joint) moves made up of all roles that are leading to a terminal state. This sequence maximizes the goal value for the role.

Additionally, the current version of GDL only allows *complete information games*. These are games where the player knows every aspect of the current game state. For example, card games like Poker are not complete information games, because the player does not know the hand of the rival players.
Semantically the rules are defined by an inclusion of conditions. The (<= (Result) (Cond1) (Cond2) ..) operator is defined as “If Cond1 and Cond2 are true, the Result is also true. The conditions can also be connected by the OR operator.

3.4 An Example

To understand the idea of this master thesis, it is required to understand how exactly this game playing works. In this chapter, I present as an example, a turn for the game Tic Tac Toe, with the rules given in listing 1. The GDL description [34] is added to the end of this chapter to make it possible to follow the GDL reasoning.
The game starts with the GGP server sending the rules as well as information on which role the agent is playing to every player. It expects the player to return a READY message within a given timeframe. This timeframe is used to parse the rules,to calculate the initial state and for strategy specific precalculations if required. After that, the server sends a PLAY command with the selected move command of all players every turn. If a player has not replied with a legal move within the timeframe of one turn, the server usually selects a random move for this player to prevent the game from crashing. [28]
This example assumes the parsing and precalculations are already done.

1. The initial state is the collection of all *init* facts.
```
State: [(cell 1 1 b),(cell 1 2 b),(cell 1 3 b),(cell 2 1 b),
(cell 2 2 b),(cell 2 3 b),(cell 3 1 b),(cell 3 2 b),
(cell 3 3 b),(control xplayer)]
```

2. Get possible moves by collection all *legal* move commands for the given state
```
Moves for xplayer:[(mark 1 1), (mark 1 2), (mark 1 3), (mark 2 1),
(mark 2 2), (mark 2 3), (mark 3 1), (mark 3 2), (mark 3 3)]
Moves for oplayer: [noop]
```

3. Select move command
```
Selected: (mark 2 2)
```

4. Send reply with move command to server

5. Get all selected moves
```
Commands: [(xplayer, (mark 2 2)), (oplayer, noop)]
```

6. Add a does(role,move) fact to the current state
```
State: [(cell 1 1 b),(cell 1 2 b),(cell 1 3 b),(cell 2 1 b),
(cell 2 2 b),(cell 2 3 b),(cell 3 1 b),(cell 3 2 b),
(cell 3 3 b),(control xplayer),does(xplayer,(mark 2 2),
does(oplayer,noop)]
```

7. The next state is the collection of all *next* facts that are true for the given state.
```
State: [(cell 1 1 b),(cell 1 2 b),(cell 1 3 b),(cell 2 2 x),
(cell 2 2 b),(cell 2 3 b),(cell 3 1 b),(cell 3 2 b),
(cell 3 3 b),(control oplayer)]
```

Of course, the selection of the next move usually requires making plenty of turns to generate a large sample count. A terminal state can be detected by the terminal(State) predicate that is true for terminal states. Under most circumstances, a terminal node has also no legal moves left, but that is not a requirement (-> Chapter 3.3) and can hence not be used for the logic.

```
; Roles
  (role xplayer)
  (role oplayer)

; Initial State
  (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))

; Legal Clauses
  (<= (legal ?w (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control ?w)))
  (<= (legal xplayer noop)
      (true (control oplayer)))
  (<= (legal oplayer noop)
      (true (control xplayer)))

; Goal Clauses
  (<= (goal xplayer 100)
      (line x))
  (<= (goal xplayer 50)
      (not (line x))
      (not (line o))
      (not open))
  (<= (goal xplayer 0)
      (line o))
  (<= (goal oplayer 100)
      (line o))
  (<= (goal oplayer 50)
      (not (line x))
      (not (line o))
      (not open))
  (<= (goal oplayer 0)
      (line x))

; Terminal Clauses
  (<= terminal
      (line x))
  (<= terminal
      (line o))
  (<= terminal
      (not open))

; Dynamic Components
  (<= (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)))
  (<= (next (cell ?m ?n ?w))
      (true (cell ?m ?n ?w))
      (distinct ?w b))
  (<= (next (cell ?m ?n b))
      (does ?w (mark ?j ?k))
      (true (cell ?m ?n b))
      (or (distinct ?m ?j)
      (distinct ?n ?k)))
  (<= (next (control xplayer))
      (true (control oplayer)))
  (<= (next (control oplayer))
      (true (control xplayer)))

  (<= (row ?m ?x)
      (true (cell ?m 1 ?x))
      (true (cell ?m 2 ?x))
      (true (cell ?m 3 ?x)))
  (<= (column ?n ?x)
      (true (cell 1 ?n ?x))
      (true (cell 2 ?n ?x))
      (true (cell 3 ?n ?x)))
  (<= (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))
  (<= (line ?x) (column ?m ?x))
  (<= (line ?x) (diagonal ?x))

  (<= open
      (true (cell ?m ?n b)))
```

Listing 1: Tic Tac Toe rulest

4. Determining move effects

To improve the performance of UCT, several approaches are possible. It is possible to reduce the search space by pruning or just to increase speed by a parallel implementation. [14] But this thesis tries to focus on another approach: A better prediction of which states are useful. Gibbs sampling tries to improve this by using the move history, but for many games this is not a good indicator. For instance, in chess a move command that beats an enemy piece is the same as for a move that ends on an empty cell.
In general, UCT also explores parts of the game tree that will not be played in any real scenarios, because it also plays move that are useless; because they do not improve the situation for the player.
Determining the improvement of the follow-up states is also the approach of heuristic algorithms like the FluxPlayer [35], but these calculations tend to need a lot of time, when compared to only the construction of the next states. UCT greatly depends on a high state visiting count, because it will not return any good results unless it converges. Additionally, there are games where UCT does not converge at all or only very slowly, because the game is more dependent on selecting a single good move in a specific state than playing a good move-chain. As an example, imagine a game where every state has one move that awards a direct reward towards the goal value. Two random search chains that will result in the same goal value will give the same bonus to the UCT and history heuristic values, but it is not possible to determine which moves have resulted in the goal value and which not. If it were possible to detect the moves responsible for the goal rewards, UCT would converge much faster to the best solution, because it could select those moves more often during the random expansion phase.
One way to achieve this is the use of move effects for Gibbs sampling, instead of move commands. The effect of the correct move would result in some change to the game state that is different from all the other moves. This idea is not new but was only applied to games with a known ruleset [27], making the effect determination a lot easier, because it is possible to work with the meaning of the predicates. For example, it is possible to make use of the fact that a “cell” in chess-like games can only hold a single piece, but in GGP this knowledge is not available. GGP also divides the information of what is possible from the information of what will happen: The ***legal*** and the ***next*** terms. (-> Chapter 3.3)
The effect of a move is the change it produces: The creation of a new fact, removing an old one or changing an existing one. Removing an existing fact is a rare case, because it is only a change in most cases. Consider these two possibilities for describing a game with cells and pieces:

1. *Each cell has it’s own fact:* If the cell is empty, it will have a fact like “cell(1 1 blank)”, or like “cell(1 1 piece)” if some kind of playing piece is at this cell. The movement of the piece to another cell would result in two changes: the constant “blank” will be replaced with the constant for the piece and the origin cell will get the “blank” constant, hence no fact was removed.
2. *Each piece has its own fact:* There will also be a fact like “cell(1 1 pawn)” but no “cell(1 2 blank)” Moving the piece will now result in the removal of the fact “cell(1 1 pawn)” and the creation of the fact “cell(1 2 pawn)”, but this can also be seen as a change to the fact “cell(X Y pawn)”, so once again no fact was removed.

Of course, there can be exceptions to this rule, but because of their rarity they will not be considered.

In GGP, effects are the result of the ***next*** inclusion with a single inclusion for every effect. There are three kinds of ***next***-Effects: Changing, Unchanging and Ever-changing.

1. *Changing effects:* The effect is modifying or adding an effect. Those effects will result in a relevant change to the game state, like movement of a piece.
2. *Unchanging effects:* The effect tells us that a fact will stay the same for the next game state, like the position of unmoved pieces.
3. *Ever-changing effects:* An effect that modifies or changes a fact independent of the move the player does. It is always happening, like the player control switch in turn-based games.

Unchanging effects can be detected by their preconditions: The clauses leading to the ***next***-Effect have to include the effect fact itself with another variable binding. If it's the same variable binding, it is unchanging. But the variable binding has to be partially the same, because the predicate itself is not enough to identify a fact. The piece or position information has to be the same.
Ever-changing effects can be detected even simpler. They are not dependent on the move command, meaning there will be no ***does*** clause for the effect. We are only interested in changing effects, because ever-changing effects can not be used to describe the difference between two moves and unchanging effects can be seen as the negation of the changing effects, because there has to be an effect for every fact of a game state.

Definition 1: An effect E is the result of a *next* inclusion, if there is a fact F in the preconditions P of this inclusion with the following characteristics:

- **predicate(E) = predicate(F)**
- **There exists a variable V1 in E at position I and V2 in F at position I with V1 != V2**
- **There exists a variable V3 in E at position I2 and V4 in F at position I3 with V3 = V4**
- **There exists a clause C in P with predicate(C) = does**

Definition 2: A change is the transition of all variable bindings from the precondition fact F, according to Definition 1, to the effect E.

The major problem with these definitions are the preconditions. Those preconditions are normally not part of the ***next*** term, but of the ***legal*** term. The ruleset itself does not define which ***legal*** term(s) are relevant for which ***next*** effect. This information can only be retrieved by a combination of both clauses. They have to be unified and stay solvable. This is a very complicated process, because the number of all possible solutions is quite high and has to be limited as much as possible. This is described in detail in chapter 4.2

Definition 3: The preconditions of an effect E are all clauses of the *next* inclusion of E and all clauses of all *legal* terms that can lead to the effect E.

The complete process is split into two parts: The preparation and the determination process. The preparation is called before the game begins, and creates a list of all changes with their according clause lists. This information is packed into an efficient indexing structure in order to find the relevant changes quickly. This structure is then matched against the state and the legal moves information of a game state to determine all changes the state/move combination will result in. This determination is very fast and simple, because all the relevant calculations are already done beforehand but will nevertheless be described in chapter 4.5. The rest of the chapter 4 is dedicated to the preparation process.

The preparation process consists of 4 steps:

1. Converting all legal and next rules to conditional normal form
2. Creation of all possible legal and next term combinations
3. Determining the effects of the combinations
4. Indexing the changes by the legal terms leading to them

4.1 Conditional Normal Form

The first step when trying to extract the move effects from the ruleset is the conversion into the conditional-normal-form [17]. The conditional-normal-form of a GDL rule is a term that only consists of predicates, which can be bound to ground facts in a single step. The ***legal*** and ***next*** rules in the Tic Tac Toe example in chapter 3.4 are already in this form, but this is an too simple example for most cases. Normally, the rules consist of sub-predicates, defining only parts of the game, like methods in functional programming languages. An example would be the chess-like game Knightwar. On the next page is the ***legal*** part of this game, including the sub-predicates ***dead*** and ***knightmove*** witch can not be bound to a fact.
It is required to get a list of all clauses of every possible solution to retrieve the clauses that are the preconditions for the corresponding ***next*** effect. Because OR handling is more difficult then AND handling, these are then explicitly converted to conditional-normal-form. OR concatenated facts are representing multiple, possible solutions opposing to AND concatenations, which are representing facts for the same solution. We are interested in as few solutions as possible, because when calculating the effect of a move in a concrete state, we have to check all solutions leading to different effects. A single, complex solution can be branched off rather fast, if the solvability check is fast.
Besides this, we are only interested in the concrete fact binding of the predicate if it is creating a solution with a different CNF predicate term, meaning the clause is consisting of different predicates. If the predicates are the same, it can only lead to the same effects with other variable bindings, but the variable bindings are state-dependent anyway. A different CNF term will be created, for example by asymmetric games, because the binding of the ***role*** variable to an atom does change the possible moves.

The complete process is as follows:

1. Expansion into a list of ground predicates with minimal variable binding.
2. Conversion into conditional normal form
3. Simplification for faster solving and solution count reduction

The expansion is done by testing all solutions of all predicates. If the solution is containing a variable, it gets expanded further. Without any variables, the solution is already bound to ground facts and is representing a single, concrete move.

```
(<= (legal ?role (move ?x1 ?y1 ?x2
?y2))
    (true (cell ?x1 ?y1 ?role))
    (not (dead ?role))
    (knightmove ?x1 ?y1 ?x2 ?y2))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus index ?x1 ?x2)
    (plus2 index ?y1 ?y2))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus index ?x1 ?x2)
    (plus2 index ?y2 ?y1))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus index ?x2 ?x1)
    (plus2 index ?y1 ?y2))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus index ?x2 ?x1)
    (plus2 index ?y2 ?y1))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus2 index ?x1 ?x2)
    (plus index ?y1 ?y2))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus2 index ?x1 ?x2)
    (plus index ?y2 ?y1))

(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus2 index ?x2 ?x1)
    (plus index ?y1 ?y2))
```

```
(<= (knightmove ?x1 ?y1 ?x2 ?y2)
    (plus2 index ?x2 ?x1)
    (plus index ?y2 ?y1))
(<= (dead ?role1)
    (true (cell ?x ?y ?role1))
    (true (cell ?x ?y ?role2))
    (distinct ?role1 ?role2))
(<= (dead ?role)
    (true (cell ?x ?y ?role))
    (true (cell ?x ?y hole)))

(plus2 index 1 3)
(plus2 index 2 4)
(plus2 index 3 5)
(plus2 index 4 6)
(plus2 index 5 7)
(plus2 index 6 8)
(plus2 index 7 9)
(plus2 index 8 10)
(plus2 index 9 11)
(plus index 1 2)
(plus index 2 3)
(plus index 3 4)
(plus index 4 5)
(plus index 5 6)
(plus index 6 7)
(plus index 7 8)
(plus index 8 9)
(plus index 9 10)
(plus index 10 11)
(plus score 0 5)
(plus score 5 10)
.
.
```

Listing 2: Part of the Knightware rulest [34]

When considering the given Knightwar rules (figure 3), the solution `(plus2 index ?x1 ?x2) (plus index ?y1 ?y2)` for the ***knightmove*** predicate is expanded, because it includes variables, but the first argument of the ***plus*** predicate is already bound to ***index***, limiting the solution count by turning the `(plus score ?x1 ?x2)` facts irrelevant. The ***plus*** predicate is not getting expanded further, because it can only be bound to ground facts. A single expansion solution to the ***legal*** predicate would be a term like this:

```
(true (cell ?x1 ?y1 ?role)),(not ((true (cell ?x ?y ?role1)),(true (cell ?x ?y ?role2)),
(distinct ?role1 ?role2))),(plus index ?x1 ?x2),(plus2 index ?y1 ?y2))
```

The conversion into conditional normal form can now be done with Boolean algebra. This is done with Boolean conversions like the De-Morgan rules [17] and should be known. The Prolog version of this algorithm is a slightly modified version of the CNF resolver from the book “Simply Logic” by Peter Flach.[17] The difference is that the original version is working with a textual representation of the terms, with *OR* and *AND* predicates. The modified version works with the predefined Prolog predicates “,” and “;” to remove useless conversations of the term.
The resulting term now looks like this:

```
(true (cell ?x1 ?y1 ?role)),(not((true (cell ?x ?y ?role1)));not((true (cell ?x ?y
?role2)));not(distinct ?role1 ?role2))),(plus index ?x1 ?x2),(plus2 index ?y1 ?y2))
```

The term now gets simplified by different approaches. At first, constructs like *(A and (not(A) or B))* are turned into *A and B*. This is already limiting the solution count, because Prolog does not try to solve *not(A),* and unsuccessful predicate calls are much more time consuming then successful

calls, since it is required to test every theoretically possible variable binding. A successful call is returning a solution as soon as the first legal binding is found.
In the next step, negative state-independent conditions are being removed. A state-dependent condition is a ***true*** call, because this predicate tests if the given fact (in this case `(cell ?x1 ?y1 ?role)))` is part of the current game state. Predicate calls like ***distinct*** are not state-dependent, because they will have the same solutions in every state. Therefore every state-independent predicate call will have a solution, as long as the arguments are not bound, and as long as it has a solution at all. If the arguments are not bound to a predicate, it means that these variables are only relevant for a concrete solution, not for the general possibility to solve the term.

Definition 4: A term is negative state-independent under the following conditions:

- **The outer predicate of the term is the negation**
- **The negated predicate is not the *true* predicate**
- **The negated predicate has at least one solution**

Negative state-independent calls like `not(distinct ?role1 ?role2)` are consequently never solvable, hence terms like *(A or not(B))* are being turned into *(A and B)* if B is state-independent. Completely removing B is not possible, since it would increase the solution count, because B can limit the solutions of A. This is also the case in the given example, because ***distinct*** is limiting the solution of the variable ***?role2*** to not be the solution of ***?role1***. Now we have this term:

```
(true (cell ?x1 ?y1 ?role)),(not((true (cell ?x ?y ?role1)));not((true (cell ?x ?y ?role2))),(distinct ?role1 ?role2)),(plus index ?x1 ?x2),(plus2 index ?y1 ?y2))
```

If the resulting term has at least one solution, it is considered valid, but the count of legal expansions is much higher than the count of solvable, legal expansions. As already mentioned, these unsuccessful calls are very time consuming and should therefore be accelerated. This can be achieved by sorting the predicates in an efficient way, as well as by improved solving mechanisms.
Because fast solvability checks are required several times in this program, they are designed to be as efficient as possible, but making the process complex. Hence these algorithms are discussed to their own chapter. (-> Chapter 4.6)

4.2 Constant-less unification

After expanding all ***next*** and ***legal*** terms, it is required to investigate which combinations of the terms are valid. Depending on which ***legal*** term has lead to the move, it is possible to restrict the number of next terms relevant. Not all combinations will be solvable.
It should be noted that variables that have the same string identifier (as example “?x”) will not be considered as the same variable in Prolog. Variables are only considered the same if they are in the same term or are being bound to another variable by a predicate call. This problem makes it difficult to find out which clauses of both terms will be bound to the same fact, but this is required for quick solving. Solving a list of every combination of a ***next*** term to a ***legal*** term is very time consuming, because of the high clause amount. Besides that, information of which variables are the same is required later on for the move effect determination.

```
(<= (next (cell ?x ?y b))
      (does ?player (move ?piece ?x1 ?y1 ?x2 ?y2))
      (single_jump_capture ?player ?x1 ?y1 ?x ?y ?x2 ?y2))

(<= (legal ?player (move ?piece ?u ?v ?x ?y))
      (true (control ?player))
      (true (cell ?u ?v ?piece))
      (piece_owner_type ?piece ?player pawn)
      (pawnjump ?player ?u ?v ?x ?y)
      (true (cell ?x ?y b))
      (single_jump_capture ?player ?u ?v ?c ?d ?x ?y))


(<= (single_jump_capture ?player ?u ?v ?c ?d ?x ?y)
      (kingjump ?player ?u ?v ?x ?y)
      (kingmove ?player ?u ?v ?c ?d)
      (kingmove ?player ?c ?d ?x ?y)
      (true (cell ?c ?d ?piece))
      (opponent ?player ?opponent)
      (piece_owner_type ?piece ?opponent ?type))

(<= (kingjump ?player ?u ?v ?x ?y)
      (role ?player)
      (role ?player2)

(<= (pawnjump white ?u ?v ?x ?y)
       (next_rank ?v ?v1)
       (next_rank ?v1 ?y)
       (next_file ?u ?x1)
       (next_file ?x1 ?x))


(<= (kingmove ?player ?u ?v ?x ?y)
      (role ?player)
      (role ?player2)
      (pawnmove ?player2 ?u ?v ?x ?y))

(<= (pawnmove white ?u ?v ?x ?y)
      (next_rank ?v ?y)
      (or (next_file ?u ?x) (next_file ?x ?u)))

(next_rank 1 2)
(next_rank 2 3)
.
(next_file a b)
(next_file b c)
.
(opponent white black)
.
(piece_owner_type wn white pawn)
(piece_owner_type bn black pawn)
.
.
```

Listing 3: Part of the Checkers rulest [34]

To emphasize the details and problems of the process, it is demonstrated with an example of the game Checkers. Listing 3 shows the parts of the GDL game description relevant for this example. It shows the terms relevant for a capturing jump for ***legal*** and ***next***.
After the expansion into CNF (one of) the resulting terms will look like this:

```
legal: (<= (legal ?player (move ?piece ?u ?v         ?x ?y))
       (true (control ?player))
       (true (cell ?u ?v ?piece))
       (piece_owner_type ?piece ?player pawn)
       //pawnjump
       (next_rank ?v ?v1)(next_rank ?v1 ?y)(next_file ?u ?x1)(next_file ?x1 ?x)
       (true (cell ?x ?y b))
       //kingjump
       (role ?player) (role ?player2)   (next_rank ?v ?v2)(next_rank ?v2 ?y)
```

```
       (next_file ?u ?x2)(next_file ?x2 ?x)
       //kingmove
       (role ?player) (role ?player2)
       (next_rank ?v ?d) (next_file ?u ?c)
       //kingmove
       (role ?player) (role ?player2)
       (next_rank ?d ?y)(next_file ?c ?x)
       //base clauses from single_jump_capture
       (true (cell ?c ?d ?piece))
       (opponent ?player ?opponent)
       (piece_owner_type ?piece ?opponent ?type))

next:  (<= (next (cell ?x ?y b))
       (does ?player (move ?piece ?x1 ?y1 ?x2 ?y2))
       //kingjump
       (role ?player)(role ?player2)(next_rank ?y1 ?v3)(next_rank ?v3 ?y2)
       (next_file ?x1 ?v4)(next_file ?v4 ?x2)
       //kingmove
       (role ?player)(role ?player2)(next_rank ?y1 ?y)(next_file ?x1 ?x)
       //kingmove
       (role ?player) (role ?player2)(next_rank ?y ?y2)(next_file ?x ?x2)
       //base clauses from single_jump_capture
       (true (cell ?x ?y ?piece))
       (opponent ?player ?opponent)
       (piece_owner_type ?piece ?opponent ?type))
```

The OR alternative from ***kingmove/pawnmove*** is removed, because it is not a valid solution. ***x1/y1*** and ***x2/y2*** are already bound by the previous ***kingjump*** clause. The first step now is the unification of the move commands. This results in binding the move command of the ***does*** clause in the ***next*** term with the move command from the ***legal*** head. The predicate call is now bound to `(does ?player (move ?piece ?u ?v ?x ?y))`. All occurrences of ?x1, ?y1, ?x2, ?y2 are now also bound to their companion piece, as well as the player variable.

In the next step, all clauses in ***next*** are now being matched to all corresponding clauses in ***legal*** until a match is found that is not limiting the solution. A match not limiting the solution will not bind any unbound variables to an atom and will not reduce the variable count. Algorithm 1 is used for this unification. Besides the mentioned variable count verification, it is limiting match testings to the same functor (predicate). This does not guarantee only correct matches are being made, but the solution is tested for solvability after the complete matching process.

```
For every E1 in next do
        For every E2 in legal do
                If E1==E2 || constantlessUnifiable(E1,E2) then
                        Legal.remove(E2)
                        Break;
                End
        End
End

constantlessUnifiable(E1,E2)
        functor(F,E1)
        functor(F,E2)
        variable_count(E1,C)
        variable_count(E2,C)
        E1=E2
        variable_count(E1,C)
```

Algorithm 1: Constant less term unification

To improve the process, the clauses of both terms are sorted the same way before matching, resulting in the first possible match to be the correct match in most cases.
Let us now focus on the clauses from the ***kingjump*** predicate in ***next*** that should be matched to the ***pawnjump*** clauses in ***legal.*** The `(role ?player) clause` can be directly matched against it is counterpart, because they are equal. The `(next_rank ?v ?v3)` clause, where ?y1 got bound to ?v by the move command, should now be matched against `(next_rank ?v ?v1)` or `(next_rank ?v1 ?y)`.
It can be directly matched to `(next_rank ?v ?v1)` , because of the sorting, which is also the correct match. With a wrong sorting, it could be also matched to `(next_rank ?v1 ?y)`, resulting in `?v1=?v` and `?v3=?y`. The remaining ***kingjump*** clause is now `(next_rank ?y ?y2)` and the ***pawnjump*** clause `(next_rank ?v ?v)`. Besides that the clause is not solvable anymore, it can not be matched against any other clause, because this would result in two variables getting bound to ?v. The result is now exactly the ***legal*** term, all clauses of ***next*** can be matched to a ***legal*** clause. The head of the ***next*** term is bound to `(cell ?c ?d b)` and the unmatched rest of the ***legal*** term is concatenated to the term.

4.3 Effects

After creating these combinations, it is now possible to retrieve the change effects and their types. The type of an effect is the ownership of the arguments from the ***next*** effect. An argument can be one of the following types:

- *Variable:* The argument is a variable that is unbound after the combination process. Those arguments are usually describing positions.
- *Neutral:* The argument is occurring in state independent clauses. This type is rarely encountered, because neutral arguments are often also occurring in state dependent terms, resulting in the All type.
- *All:* The constant is part of state dependent clauses, but for both players. Player independent piece constants are assigned this type. For instance, we often have a “knight” piece that describes any knight in a chess-like game.
- *Role:* If the constant is only occurring in state dependent clauses, but merely for one player, then the role name of the player is used as type. This is usually only the case in asymmetric games, because for other games the player variable is only being bound for concrete states.

Besides the change of the type, the value of the variables and the predicate is also used as effect.

[[[[cell(a, 2, wn), cell(c, 1, b)], [[cell, [b, all], [wn, variable]], [cell, [wn, variable], [b, all]]]]]]

is an example effects term from a chess-like game. It describes the movement of a piece *wn* from position a,2 to a the blank cell c,1. This results in the start cell being marked as blank (constant *b*) and the target cell now holding the piece *wn*. The first part describes the bound effects and the second part is the type change mentioned before. *Wn* is not having a role type, because the effect is the same for all pieces and players.
The extraction of these types is pretty simple. After converting all terms into CNF and combining them, the program is parsing all clauses for all arguments. When encountering a constant, it is

tested if the player variable is already bound and the type is set accordingly. For investigating the *change* of the type, it is still required to find the according precondition for the effect clause (corresponding Definition 1.2), but they can be found by matching the effect clause against all precondition clauses, that are now available by the combination process.

4.4 Indexing Structure

The indexing structure should sort all terms and their according effects. It is used to retrieve all relevant effects of an explicit move/state combination by unifying these clauses with the given terms of the effects. If it is possible to unify the state with a term in such a way that the term becomes solvable, the effects are relevant for the given move. To be efficient, the structure should minimize the amount of unifications required to retrieve those effects. An unoptimized version would require testing the unification of all ground clauses in the current state, and all move independent ground clauses in the GDL rules, with all clauses of the extracted terms. Prolog itself already limits the unifications to clauses with the same predicate and variable count, but that is still a very high amount. Consider the example from chapter 4.2: There is a ***next_rank*** fact for every rank of the game board and all unbound ***next_rank*** clauses in the extracted term have to be tested for every possible combination. But by remembering the unified clauses that have lead to the move, we can already reduce the possibilities a lot. When calculating the legal moves for a game-state, all clauses of the ***legal*** predicate are being matched against the current state. We can save these unifications and reuse them for the effect terms. It is only required to have a logic association between the effects term and the ***legal*** term that has lead to the effects term. Besides the merged legal and next term for the effects, the ***legal*** term is stored with the same variables as the merged term. This is achieved by saving the legal term in a logical container before merging it with the ***next*** term. When a unification for the ***legal*** term is triggered, it get's also executed in the logical container.
The ***legal*** predicate itself is saving all facts that have lead to a solution, besides the solution itself. The solution `(legal white (move wn a 2 c 1))` from the Checkers example will also return this:

```
(true (control white))
(true (cell a 2 wn))
(piece_owner_type wn white pawn)
(next_rank 2 3)(next_rank 3 4)(next_file a b)(next_file b c)
(true (cell c 1 b))
(role white)(next_rank 2 3)(next_rank 3 4)
(next_file a b)(next_file b c)
(role white)(next_rank 2 3)(next_file a b)
(role white)(next_rank 3 4)(next_file b c)
(true (cell b 3 bn))
(opponent white black)
(piece_owner_type bn black pawn))
```

The copy of the ***legal*** term for the logic container will remain in the same order as it was created by the ***legal*** predicate. It is not sorted like the effects term. This way it is possible to unify the facts from the legal solution described above with the stored term by only matching the clauses at the same position. Because it was possible to unify all clauses of the ***next*** term with the ***legal*** term, no unbound variables are left for the effect. We already have a unique solution, but there are exceptions. For example, a piece counter effect will not be bound, because the piece counter is not relevant for the ***legal*** move, only for ***next***. This unbound rest still has to be matched against the complete state, but this can be done quickly, because the clause amount of the rest term is very low.

However, there can be more than one ***legal/next*** combination which is valid for the current move. To reduce the amount of terms that need to be tested, similar terms are also being unified. The terms for effects that are based upon the same ***legal*** solution are being unified into a single solution. Only the rest terms from the ***next*** predicate can differ.

The indexing structure is now saving:

- The unified legal term
- The possible rest of the next term
- The effect

The possible rests of the ***next*** term with the according effects are being grouped and indexed by the unified ***legal*** term. This way all effects relevant for a specific ***legal*** term are grouped. Because a move can only be created by a single ***legal*** solution, it is only required to consider all effects and rest terms of the ***legal*** term that have lead to the solution. In theory, it could be possible to get several legal solutions leading to the same move, but this could then be handled like a different move. After aquiring all effects for all moves, all effects for the same move command are grouped, regardless of the ***legal*** solution; hence all effects for the move command are collected.
Let us assume Checkers has a piece counter, then we would get a structure like this:

Legal term:

```
(legal ?player (move ?piece ?u ?v ?x ?y))
(true (control ?player))
(true (cell ?u ?v ?piece))
(piece_owner_type ?piece ?player pawn)
(next_rank ?v ?v1)(next_rank ?v1 ?y)
.
.
```

Effect 1: `(cell ?u ?v b)`, Rest term: `[]`
Effect 2: `(cell ?x ?y ?piece)`, Rest term: `[]`
Effect 3: `(piece_count ?opponent ?z)`, Rest term: `(piece_count ?opponent ?z1)(succ ?z ?z1)`

The first two effects are already completely bound by the legal term, and for the last effect, ***?opponent*** is already bound. The state will only contain a single fact with `(piece_count black ?z1)`, directly binding **?z1** and leaving only a single solution for `(succ ?z ?z1)`. Now all effects are determined.

4.4.1 Substitution Tree Indexing

We still have the problem of the possibly high amount of legal solutions. Even after grouping all relevant effects for a single legal term, it remains time consuming to find this legal term. An efficient indexing structure for this problem would be a substitution tree. Substitution trees are using the term itself as an index. It was shown that these trees are efficient both in search time and memory consumption. [20]
Each edge is containing a clause of the term and the tree is traversed by unifying the tree entries with facts. If the unifying was successful, the traversing progresses to the next deeper node. The tree's leaves are containing the effects and the rest terms. Because the legal terms are all ordered

in the same way, the tree is easy to build. It is simply a chain of clauses as long as the legal term is the same. When the legal term starts to differ, the node will get new edges, representing other clauses.

The calls of predicates not resolving to a fact (like ***knightmove***) are also added, because they can be used to differentiate some moves early. For example, the chess-like games that also include pawns besides knights have a call to ***pawnmove*** instead of ***knightmove***.

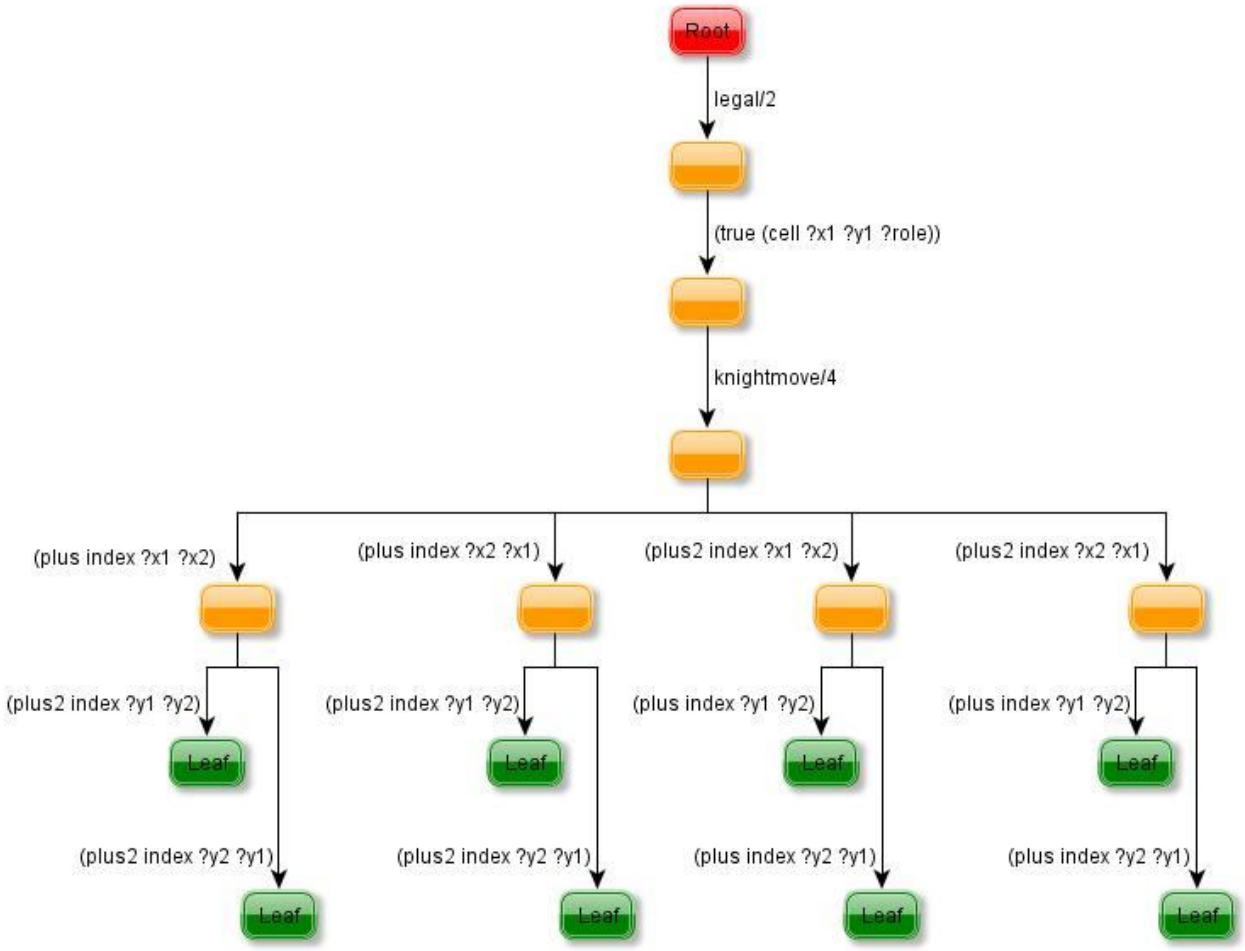


Figure 2: Substition tree for knightwar

This is the substation tree for the Knightwar example from chapter 4.1. It is differs from the result of the real program, because it serializes the legal term in another order, this is however not relevant and has only programmatic reasons. The `(not (dead ?role))` clause is not part of the tree, because we are only interested in clauses that are binding new facts. Negated clauses are not binding any facts. If it was possible to bind the facts in a negation, clause would be solvable and therefore the negation would fail.

This tree now only has a single, valid solution for every level. After binding the `(true (cell ?x1 ?y1 ?role))` clause to a fact, all other occurrences of ***?x1*** and ***?y1*** are also bound. The choice between ***plus2*** and ***plus*** as the next predicate is removed by the legal term. Only one of these will be possible as 2nd clause. Because of the binding of ***?x1***, the correct possibility of the two remaining ones is defined as well. If there is no possible unification for the current edge, the search is backtracking to the last node and tries other unifications. If this leads back to the root node, then no solution is valid.

4.4.2 Compression

The amount of required unification can still be further reduced. We are only interested in the quick determination of effects. No other information is relevant to us. Hence we can remove everything that does not contribute to the determination of the effect. A substitution tree entry is contributing to this determination if it's binding a new variable that is part of the effect, or if it is creating a new branch of the tree.

Definition 5: A clause C is relevant to the effect E under the following conditions:

- **It exists a variable A in C and A exists in any clause D of E**
- **A does not exists in any clause C2 that comes before C in the term L**

Definition 6: A clause C is relevant to the substation tree E under the following conditions:

- **C is the i-th element of the term L**
- **It exists a term L2 with the i-th element not equal C**

legal/2 and ***knightmove/2*** do not fulfil any of these conditions and are therefore removed. The other clauses represent a choice and are binding relevant variables. Removed clauses are replaced with a number representing the number of irrelevant, consecutive clauses.

Clauses of the rest term which are saved in the leaves also have to fulfil definition 5 or they will br removed. The rest term is required only to bind variables for effects that have not been bound yet.

4.5 Determination

The determination itself can now be done fairly simple. The determination function is called with the ***legal*** term from the legal moves predicate, the root node from the indexing tree, and zero as last argument, marking the current position in the ***legal*** term. The ***legal*** term itself will be a AND concatenated list of clauses, without any OR predicates. If two clauses are OR concatenated and both clauses are solvable, two solutions for the ***legal*** predicate will be returned, but they will not both occur in a single solution.
A solution to the ***legal*** predicate of the game Knightwar (->Chapter 4.1 and Image 5) could look like this:

Move: `move(3 3 4 1)`
Term: `[legal/2,true(cell 3 3 black) ,knightmove/4 ,plus(index 3 4) ,plus2(1 3)]`

When applying the algorithm 2 to the index structure from chapter 4.4.1 the encountered clause matches are:

```
legal/2 = legal/2 -> true(cell 3 3 black) = true(cell ?x1 ?y1 ?role)
-> knightmove/4 = knightmove/4 -> plus(index 3 4) = plus(index 3 ?x2)
-> plus2(index 1 3) != plus2(index 3 ?y2) -> plus2(index 1 3) = plus2(index ?y2 3)
-> Leaf
```

The leaf contains effect data that will now be returned without being null, therefore closing all recursive determine calls by breaking the "For" loop. If no current edge can be unified with the

given legal clause, all remaining edges of the parent node are tested. By performing a solving check with the effect data, that includes the relevant rest term (->Chapter 4.4.2), all variables are now bound. The given example will now return the effect:

```
[[[cell(3 3 hole), cell(4 1 black)], [[cell, [black, variable], [hole, all]], [create]]
```

This is the final result for the Prolog part of the program; all further use of this information (in the history heuristic) is done in Java as part of the Gorgon player example implementation.

```
For(Edge : current.Edges)
        If(Edge.Data != Null &&    solvable(Edge.Data)) Then
                Return Edge.Data
        End
        If(Edge.Clause instanceOf Number) Then
                I+=Edge.Clause
        Else If(Term[I] = Edge.Clause) Then
                I++
        End
        Effect = Determine(Term,Edge.Node,I)
        If(Effect != Null)
                Break
        End
End
Return Null
```

Algorithm 2: Determination

4.6 Solving

There are several points in the preparation and determination process requiring fast solving checks. While preparing, we want to check often if the current term even has any solution at all. If unification renders a term unsolvable it cannot become solvable again at any later stage the process, hence we would like to detect unsolvable terms as early as possible. Solving checks themselves are unifications as well, because they have to test if every clause of the current term can be unified with a fact. If all variables are simultaneously bound, the term is solved. Depending on the length of the term, it is possible that more unifications are required to perform a solving check than for the combination process itself. To still be able to branch off invalid combinations early, several different solvers are used, depending on the required quality and the complexity of the given term.

Besides the preparation solving checks, the solving of the rest terms when determining the effects must also be as fast as possible. In fact, this is more important than quick preparation solving checks, because the move effect determination needs to be done for every move in every state. The preparation is performed before the real match begins and only has to fit within the given precalculation time frame. (->Chapter 1.1)

There are four different solvers in total:

- A fast, but incomplete solver for regular checks.
- A complete solver for short terms
- A complete solver for long terms
- A complete solver for the determination process

All of these solvers have their strengths and weaknesses and are optimized for a specific situation. The details of these are described in the following chapters.

4.6.1 List solver

The constant-less unification process can create illegal argument configurations that cannot be bound to any fact anymore. (->Chapter 4.2) This is not limited to the clauses that get unified, but can also affect other clauses with the same variables. To detect these unifications early, it is necessary to often test the term for solvability. Because the term is already in CNF, it can be considered a list of clauses. The list solver is simply testing the solvability of every element of this list. The difference to a normal predicate call is, that the combination of the clauses is not solved, only the clauses themselves. Each clause is copied and solved, but the variable bindings are not forwarded to other variable occurrences. However, variables already bound by the combination process are staying bound, of course.
By not forwarding the variable bindings to other clauses, the solvability of a clause cannot affect any other clauses. This way backtracking is not required, and the first fact that matches the current clause predicate is often unify-able, because the combination process only binds very few variables. Hence it usually takes only one unification per clause for this solvability check.
Terms which are not solvable because there is no valid combination of single clause solutions cannot be detected this way, but the test is very fast. The solver is called for unification in the combination process, after the constant-less unification is finished for the clause.
This solving check is very effective, because most move terms are based on a similar clause chain, like in the Knightwar example in chapter 4.1, therefore often allowing a theoretical combination.

4.6.2 Term order solver

After the complete combination process is done the solvability of the term is tested again, but this time with a complete check that includes the variable binding forwarding required for a solving check of multiple clauses, that share variables. The build in Prolog solver for terms (algorithm 3) is very fast, because it optimizes the unifications themselves [11], it is however an exhaustive process. The solver is binding every clause to the first possible fact. If a clause is not bindable with any fact, all remaining solutions for the last called predicate are revised.
Let us assume we want to solve the following term:

```
plus(A,B),plus(B,C),plus(C,5)
```

with the fact base:

```
plus(1,2), plus(2,3),
plus(3,4), plus(4,5)
```

The solving now would take 7 unifications, because the first two clauses are being unified twice with a fact which creates an illegal solution for the 3rd clause. A reordering by moving the clause plus(C,5) to the beginning of the term would reduce the required unifications to 3, because there is only one possible solution for plus(C,5) even without binding the other variables.

Sorting the clauses by solution count seems to be a good approach. However there is a better way, because retrieving the solution count already requires binding every clause to every fact in order to test if the unification is possible. Because this has to be done in addition to the solving itself, the total number of unifications rises, but there is an approximation available: Standard term ordering

```
Clause = Term[0]
If(Clause.Predicate = AND) Then
        Call(Clause.Argument[0])
        Call(Clause.Argument[1])
        Return True
Else If(Clause.Predicate = OR) Then
        If(Call(Clause.Argument[0])) Then
                Return True
        Else
                Call(Clause.Argument[1])
                Return True
        End
Else
        For(Fact : Facts)
                If(Fact = Clause) Then
                        Return True
                End
        End
End
Return False
```

Algorithm 3: Prolog solver (simplified)

The standard term ordering in Eclipse CLP is defined as follows (greater to smaller) [11]:

1. Term Arity
2. Functor Name
3. Arguments from left to right
4. Variables
5. Atoms

The atoms are also having a sub-order, but this is not relevant in this case. In fact, we are only interested in two parts of the ordering: Terms with a higher arity have a higher ordering value. In most cases, more arguments results in more possible solutions; hence variables have a higher value then atoms. An atom is already a single solution, a variable can have several.
Comparing plus(A,B) with plus(C,5) will return the 2nd clause as smaller, because the arity is the same, as well as the first argument. But the atom “5” is considered smaller then the variable “B”.
Sorting the term by standard term ordering (smaller to greater) is an approximation of the solution count ordering accordingly. This can be done without a single unification because the term ordering value is defined by the clause itself. The sorting process is done with the Quicksort algorithm [21].

4.6.3 Enhanced backtracking solver

Even with presorting, the exhaustive Prolog solver is far from optimal. When testing all solutions with a single-step backtracking, the solver is going to test new solutions that do not have any effect on the currently unsolvable clause.

```
For(i=0;i<ClauseList.length;i++)
        Variables = getTermVariables(ClauseList[i])
        For(Variable : Variables)
                If(not(VarMap.contains(Variable)) Then
                        VarMap.add(Variable,i)
                End
        End
End
For(Clause : ClauseList)
        Variables = getTermVariables(ClauseList[i])
        For(Variable : Variables)
                Indexes.add(VarMap.get(Variable))
        End
        MappedClauses.add(Clause,Indexes)
End
```

Algorithm 4: Enhanced Backtracking (Map creation)

When solving the term

```
plus(A,B),plus(C,D),plus(B,5)
```

with the fact base

```
plus(1,2), plus(2,3),
plus(3,4), plus(4,5)
```

the 2nd clause is not affecting the last one because they are not sharing variables. The Prolog solver is not testing variable dependencies and will therefore test all remaining solutions for `plus(C,D)` before testing a new solution for `plus(A,B)`. The last clause will only be solved when binding the first clause to the 4th possible fact. Because the solver assumes a new solution for the 1st clause could lead to a new solutions for all other clauses, the 2nd clause is bound to every fact for every solution of the first clause. Hence 4*(1+4)+1=21 unifications are required to solve the problem with the standard solver. The enhanced backtracking solver overcomes the problem by also considering the link between the variables.

The solver creates a map that contains a pair for every clause of the term, with the clause as first entry and the first occurrences of the clause's variables as second entry. The map preserves the order of the clauses. Algorithm 4 shows how this map creation is implemented. It simply collects all variables of all clauses and saves their first occurrences. In a second step, the first occurrences map is applied to the clause list for creating the MappedClauses list which contains the clauses with all first occurrences. The solving loop (Algorithm 5) now tests all unifications with a fact, and if the binding was successful the next clause is called as usual; but if no bindable fact is available, the solver is returning the errormap of the first occurrences of the clause's variables. It should be noted that all variables of the clause are getting bound at once; therefore it is not possible to determine which variable could not be bound. Every recursive solving call then tests if the errormap contains the index of its clause. If it does, the clause has bound a variable that is part of the unsolvable clause. Creating a new variable binding for this clause could potentially enable the unsolvable clause to get bound to a new fact. When considering the given example, the 2nd clause will now only be bound once for every solution of the first clause, because the failing of the 3rd clause solving will backtrack over this clause to the first one. The unification count is now reduced to 4*(1+1)+1=9.

```
If(MappedClauses.length <= i) Then
        Return []
End
Current  = MappedClauses.get(i)
For(Fact : Fact)
        If(Current.Clause = Fact) Then
                j = i+1;
                errormap = solvingLoop(MappedClauses,j)
                If(not(errormap.contains(i)) Then
                        Return errormap
                Else
                        errormap.remove(i)
        End
End
errormap.add(Current.Indexes)
Return errormap
```

Algorithm 5: Enhanced Backtracking (Solving loop)

Though this algorithm is much more efficient then the standard Prolog solver, it is slower in most cases, of its higher complexity. Besides the additional information that has to be considered, as well as the map creation, it is also slower because the built in solver is optimized on a much lower abstraction level than this meta-solver written in Prolog. But for very complex terms, this solver is a lot faster and also less memory consuming, because it needs not to save the backtracking information for every step, but only where it is required. This improves the solvability up to a point where some complex terms are only solvable with the enhanced solver, because the standard solver will run out of memory. For GGP in particular, memory consumption is a relevant point, because as much memory as possible should be used for storing the game tree. The decision this solver should be used is depending on the term complexity, as mentioned, but determining the complexity is itself a difficult problem. It would be required to create a dependency graph of the term to find out which clauses are important (having variables that are linked with many other clauses). To simplify this process, an approximation of the complexity was used: The term length. It can be calculated very fast by simply counting the clauses, and it is a good enough feature in all tested cases.

Because the solver is only used for terms that need a high amount of unifications to be solved, it is also possible to improve the presorting a bit by using a better but more time consuming presorting than the simple term order sorter. In fact, a good dependency sorting will often decrease the overall solving time, because the enhanced backtracking solver has to test considerably less unifications.

The presorting algorithm 6 for enhanced backtracking solving uses the variable occurrence amount as its most important sorting feature, and an approximation of the solution count as its second level feature. Binding the most often occurring variable first will result in the greatest reduction of unbound variables. A lesser amount of unbound variables reduces the choice points, meaning the amount of facts bindable to a clause. After retrieving all clauses containing the most-common variable, they get sorted by the solution count of the predicate. The lesser the amount of solutions for the first clauses, the lesser the overall amount of solutions that have to be considered:

The term `pred1(A,B), pred2(B,C)` with the fact base `pred1(1,2),pred1(2,3),pred2(3,4)` would be resorted to `pred2(B,C), pred1(A,B)`. Both contain the most-common variable B, but `pred2` has only one possible solution and is also reducing the possible solutions of `pred1(A,B)` to one by binding B. The original term would bind `pred1(A,B)` twice before solving it.

The solution count is only an approximation because it is precalculated for every predicate and arity without considering any possible bound variables. This way, it is only required to count the amount of facts for every predicate, not the solutions of every clause.
After adding the clause with the most-common variable and the least solutions, the list of the rest clauses currently having the most-common variable is sorted the same way until all elements are added. Only when all most-common variable clauses are added, a new most-common variable is determined out of all remaining clauses. The reason for this approach is simple: A clause with partially bound variables will have less possible solutions than a clause with no bound variables (when having the same arity and predicate) and adding clauses with less solutions early results in less solutions overall, as shown earlier.
Additionally, when a clause is added to the result, all variables of this clause are removed from the count list, since they will be bound for every following clause when solving the term. Clauses having no shared variables with the count list only have variables that are already bound. Therefore these clauses are added right away, for testing if the variable binding is valid for the given predicate.

```
For(Clause : ClauseList)
        Variables = getTermVariables(Clause)
        For(Variable : Variables)
                Count[Variable]++
        End
End
Sort(Count)
For(Variable : Count)
        For(Clause : ClauseList)
                Variables = getTermVariables(Clause)
                If(not(Count.contains(Variables)) Then
                        currentClauses.add(Clause)
                        ClauseList.remove(Clause)
                Else If(Variables.contains(Variable)) Then
                        currentClauses.add(Clause)
                        ClauseList.remove(Clause)
                        Count.remove(Variables)
                End
        End
        SortbySolutionCount(currentClauses)
        resultList.add(currentClauses.remove(0))
        If(currentClauses.length > 0) Then
                Presort(currentClauses,Count,resultList)
        End
End
```

Algorithm 6: Enhanced presorting

4.6.4 Determination preorder solving

The determination process is the time critical part of this approach because it needs to be done for every move in every step of the game. The time the determination takes directly reduces the amount of states that can be sampled for UCT. Apart from a good indexing method (->Chapter 4.4), it is required to solve the rest terms fast. If the substitution-tree indexing process was successful, the validity of the effect is still not guaranteed. Also, it has also to be determind which sub effects are relevant, hence it is required to solve the remaining effect terms. After the compression the remaining terms are very short in most cases; therefore we are going to use something like the term order solver. The main drawback of the term order solver is the requirement that we need to sort the clauses, but for the determination process we can split the solving from the sorting. The rest terms are not changing after being added to the index tree. Because of this fact we can move the sorting into the index tree creation that is done in the preparation phase so it will not influence the determination time. To reach a solution as fast as possible, a real solution count sorting is used. The sorting is not done by the fact amount of the predicate, but with the real clauses and all possible bound variables. To reach a result as fast as possible, the player and the move variable are getting bound before sorting, since this information will also be available before the determination. It is created by the calculated legal moves. The sorter binds the player and move variables to the first valid solution for the term and then counts all remaining legal fact bindings for the clause. The sorting itself is again done with Quicksort.

4.7 Special Cases

During the preparation phase, some special cases can occur, which prevent the successful creation of the move determination structure. For example, recursions will lead to an infinity loop in some cases, because the game description language itself is not limiting the recursion depth. It is also possible that the term is unsolvable when encountering not-equal clauses. To make games playable that contain these problematic features, a special case handling is required.

4.7.1 Recursion handling

Recursions are very common in GGP, especially in chess-like games. The rook move is often defined recursively because this way it can be easily defined, like shown in listing 4. When encountering a recursive predicate, several questions occur:

- How to detect recursive predicates?
- Should the clause be expanded?
- Is the recursion solvable?

```
(<= (rookmove ?u ?v ?u ?y)
    (clear_column ?u ?v ?y))

(<= (clear_column ?u ?v ?y)
    (next_rank ?v ?y)
    (file ?u))

(<= (clear_column ?u ?v ?y)
    (next_rank ?v ?w)
    (true (cell ?u ?w b))
    (clear_column ?u ?w ?y))
```

Listing 4: Rookmove recursion example

The detection of a recursion is fairly simple because the predicate needs to occur in the body of the predicate's rule. It is possible that the predicate is not itself calling directly but via another predicate. However, this scenario

was not encountered in any real game description and is therefore not supported, because for runtime reasons. To detect such sub-recursions, it would be required to verify all predicates that could be called by the predicate before expanding the clause.

If a recursive predicate is encountered, it is added to a list of recursive predicates. The list saves the recursive version of the predicate as well as the terminal version. For the creation of the indexing structure, the call to the recursive predicate is replaced with a new call `solveRecursion(Clause,State)` which triggers the new recursion handler. This replacement is also done for the occurrence of the recursive predicate in the recursive version of the predicate's body. The clause in this call is the old recursive predicate clause. The `solveRecursion` predicate takes care of two difficulties: It keeps track of the recursion depth to limit it for the solvability checks. If a solvability check fails because of a clause that occurs after the recursive clause, the solver will backtrack to the recursive clause and try another solution. But if the failing clause cannot be solved at all it will lead to the creation of an infinite amount of recursive solutions. The depth limit is preventing this. Of course there can be no depth limit guaranteeing correct solvability for all games, but a value of 30 seems to be acceptable, because it is larger than the side of nearly all game boards. Algorithm 7 shows this depth limiting `solveRecursion` predicate.

```
If(i<depthLimit) Then
        Functor(F,Clause)
        Arity(A,Clause)
        testClause = getClause(F/A/term)
        If(not(call(testClause))) Then
                testClause = getClause(F/A/rec)
                i++
                call(testClause)
        End
End
```

Algorithm 7: Recursion handler

The other difficulty is the uniqueness of moves. If adding the expansion of the recursive predicate calls to the move effect structure, there would be a new move term for every depth the recursion can be solved at because it would contain more clauses. With this replacement, there will be only one term that doesn't depend on the recursion depth. Limiting the amount of move terms results in a smaller substitution tree, making the determination process faster.

The `solveRecursion` clause is not added to the index part of the substitution tree, but to the effect leaves because the recursive predicate calls are removed from the term used for parsing the index. Recursions are mostly defining the connection between two facts, like the current piece position and the target position of the move. The facts themselves or the information contained in them will also be part of other clauses of the legal move terms. For example, the start and end position of a move is also saved in the move command, hence it can be removed because is not required to unify the recursive clause. It is still added to the rest term in the leaf to verify if the clause is already bound.

To further improve the speed of the recursion handler all visited solutions, both successful and unsuccessful, are added to a solution list. When the recursion handler is called, it is first tested if the given clause can be bound to one of the correct solutions to skip the remaining recursive solving process. If the recursive clause is already completely bound, it can also be matched against the failing solutions, because the clause cannot be modified in any way to make it solvable. This acceleration normally only affects the preparation process because in most cases when determining moves, the recursions have already been solved by the ***legal*** predicate. But it is possible that the recursion occurs in a sub effects term (see next chapter).

4.7.2 Sub effects

Sub effects are effects that can but don't have to occur in combination with another effect. We have two valid ***next*** terms for the same ***legal*** term, where the one combined term is an extension of the other. An example for this behaviour are chess-like games. Every move will have the effect that the target cell contains the moved piece. If there was an enemy piece at the target cell, the amount of pieces would be reduced meaning a fact describing the piece count will change.
When creating the substitution tree the largest part of the combined term will become part of the index structure or be removed by the compression. The remaining clauses are compared and all clauses that are contained within the master effect are getting removed from the sub effect. All sub effects with the remaining terms are added to the effect as an additional list. If an effect is determined, the process also tries to solve all sub effects and if it was possible adds them to the effects list. Listing 5 is showing the aforementioned example from chess-like games, where `(pieces opponent ?m)` is the sub effect.

```
(<= (next (cell ?x ?y ?p))
    (does ?player (move ?p ?u ?v ?x ?y)))

(<= (next (pieces ?opponent ?m))
    (does ?player (move ?p ?u ?v ?x ?y))
    (true (cell ?x ?y ?q))
    (distinct ?q b)
    (opponent ?player ?opponent)
    (piece_owner_type ?q ?opponent ?kind)
    (distinct ?kind pawn)
    (true (pieces ?opponent ?n))
    (succ ?m ?n))
```

Listing 5: Sub effects example

4.7.3 Not equal

Not-equal clauses are often part of the move terms because they are required to test if two variables or facts are distinct. Every game that has a component in which it is possible to beat an enemy piece requires this predicate to define the difference between the players and the opponents' pieces. The example from chapter 4.7.2 is also includes a distinct relation to distinguish between a blank constant and a players' piece constant. These clauses become problematic when testing for solvability. A not-equal clause is solvable when both arguments of the predicate cannot be unified, but as long as the arguments are unbound variables, they can always be unified. This problem is solved by two additions: Not-equal clauses are moved to the end of the terms. A not-equal clause is only useful if the arguments are also occurring in other clauses. If this were not the case, the game would test the inequality of something that is not relevant for the move.
This way we can cuarantee that the arguments of the inequality clause are bound. By making them last, their variables will already be bound by other clauses. This is also reduces the amount of clauses containing unbound variables when encountered, hence it improvs the solving speed.
But when testing the solvability of a term, we are not always binding all clauses at once (->chapter 4.6.1). By replacing not-equal with non-identical for those test, we will also get a valid solution, because the Prolog identity test is only successful if both arguments are exactly the same. For example, VarA != VarB is unify-able because they are unbound, but they are not identical because they are different variables.

5. The Gorgon Player

There are multiple possibilities to make use of the effect determination for a GGP player. The Gorgon Player is an example implementation making use of the new information for the Gibbs sampling directly (-> chapter 3.2). Other examples for facilitating this data are given in the Further Work chapter at the end of this thesis (-> chapter 6.3)
The following chapters describe the implementation details of the Gorgon Player. The UCT and communication part of the player was implemented in Java [37]. For the reasoning part, Eclipse CLP [11], a Prolog environment, was employed. The complete program is based upon the "Basic Player" [24] by Stephan Schiffler, but was heavily modified and extended. The Basic Player offers a Prolog interpreter for the GDL descriptions and the reasoning, which is required for calculating next states, legal moves etc.
Additionally it offers a framework for the implementation of new strategies. The server communication is also taken care of. Besides the implementation of the UCT strategy and the move effect reasoning, it was required to modify the program, because it was not intended for parallel computing. But because current computers have multiple cores and/or CPUs, a parallel implementation is required to make use of all the computational power modern systems are offering.

5.1 UCT Strategy

The implementation of the UCT strategy is pretty straight forward. The searcher starts with the current state of the game and traverses the game tree as described in Chapter 3.1.

```
While(true)
        If(state.depth > startNode.depth + depthLimit) Then
                randomExpand(state,currentGame)
                state = startNode
        EndIf
        checkExpansionResults()
        nextmove = uctSelect(state)
        currentGame.add(state,nextmove)
        If(notInTree(state,nextmove)) Then
                randomExpand(state,currentGame)
                state = startNode
        Else
                state=getNext(state,nextmove)
                If(isTerminal(state)) Then
                        goalValues = getGoalValues(state)
                        propagateGoalValues(currentGame,goalValues)
                        state = startNode
                End
        End
EndWhile
```

Algorithm 8: UCT search

It facilitates some minor improvements, like iterative deepening [32] or perfect choices. Perfect choices are moves that will guarantee a maximal payoff for a player. If a state has a perfect choice, it can be considered a terminal because the player should always select the maximizing move. Perfect choices are not handled by the UCT searcher itself, but by the underlying model. When a terminal state is encountered, which is a prefect choice, the prestate is also marked as terminal. This results in the UCT algorithm ignoring the real terminal because the search ends when a terminal is encountered. This can be considered a simple pruning mechanism. Algorithm shows the implementation of the UCT search, including the iterative deepening. The random expansion function (algorithm 9) is required for decoupling the simulation runs from the UCT search. It calls a new thread playing a single game to the end by selecting the next move by the Gibbs Sampling function and not via the UCT values. The details of the decoupling are mentioned in chapter 5.2.1.

```
While(not(isTerminal(state))
        moves = getMoves(state)
        nextmove = selectMove(moves)
        state = getNext(state,nextmove)
EndWhile
goalValues = getGoalValues(state)
depth = state.depth-startNode.depth
goalValues = decayValues(goalValues,depth)
addToResults(goalValues,currentGame)
```

Algorithm 9: Random expansion

UCTSelect (algorithm 10) selects the next move according to the UCT formula. If an unsampled action is encountered, its value is assumed to be 100. This results in unselected actions getting selected first.
When multiple actions are having the same UCT value, Gibbs sampling is used to determine the action that should be played. To increase the effect of the Gibbs sampling, all moves are considered best, that are having an UCT value of at least 0.9*bestValue. This was added because of the assumption, that moves can be described by their effect (Chapter 4.3) and that this description is more relevant then small differences in UCT values.

```
moves = getMoves(state)
bestValue=0
For(move : moves)
        If(state.isVisited(move)) Then
                value = calculateUCT(state,move)
        Else
                value = 100
        EndIf
                If(uctValue > bestValue)
                        bestValue = value
                End

        valueList.add(Move,Value)
End
```

```
For(i=0 to valueList.size())
        If(valueList.get(i).value  < 0.9*bestValue) Then
                valueList.remove(i)
        EndIf
End
        Return gibbsSampling(valueList)
```

Algorithm 10: UCT Select

The choice of the acceptable value range is very important. Too big a value will render the UCT useless, because too many values could be close together. However, a small value would ignore the heuristic effect values, because only one move is selected any case.

The Gibbs sampling uses three kinds of information for its heuristic:

1. The move history, as used in the classic Gibbs sampling. This is a relevant description for games, where a specific move or position is always better. The edges in Othello are an example for this, because they cannot be flipped back.
2. All effects of the move. This often includes the same information as the move history, because the effect is dependent on the move, but they are not equal. The effect will also include predicates like piece counter that or often directly connected to the goal values.
3. All changes of the move. This is the information of what constant is getting replaced by another constant and of which type they are. In contrast to the effect, this also includes parts of the precondition, but not the constant parts of the effect, like the position. The type of a constant is a role or neutral, in order to define who "owns" the fact.
   This feature can describe a piece beating the enemy queen in chess, as example.

The idea behind this is that different features can be beneficial for different games. If a feature is not able to capture the properties of a game, it will have similar values for every entry, hence other features that can describe a good move will have a higher value. Combining everything into a single feature would limit the heuristics, because the information is too fine-granular. The heuristic value of the effect "cell(1 1 enemyQueen)->cell(1 1 myPawn)" would not profit from the detection that "cell(2 2 enemyQueen)->cell(2 2 myPawn)" is a good move. However, the position information is irrelevant for this feature, hence this information is split.
The Gibbs temperature τ (->Chapter 3.2) is selected in such a way, that even small differences in the heuristic value of a feature will result in higher selection probability. This way the good features have a much greater influence on the probability value than bad ones, although all three features are considered. The move selection for which move should be played in the real game is also uses the 10% tolerance value with Gibbs sampling, but it's modified to not work with probabilities but by selecting the highest heuristic value. If the highest heuristic value is the same for multiple moves, the next lower values are considered.

One of the problems of this UCT implementation is single player games. The algorithm is build to find the best possible average values for the next few game states, meaning it does sample as many of the possible moves, as often as possible. But single player games are not played with the best average move, but by playing the first move of the best seen path to a terminal, hence we should try to simulate as many *different* paths as possible. To achieve this, a larger part of the

game tree is stored into the RAM, which is possible because single player games tend to take up less memory anyway, since more players increase the count of possible moves. The early random expansion of the game tree, when a node is encountered that is not part of the game tree, also is removed. By always UCT-selecting up to the iterative deepening threshold, the count of moves played multiple times is also reduced, because UCT favours unselected moves.

5.2 Changes to Basic Player

The Basic Player is used as the base for the complete program. It is a Java program that is based upon the Palamedes IDE plugin for Eclipse [12]. The Palamedes IDE provides a GDL editor interface with the possibility to test if those game descriptions are working properly. For this reason, a rudimentary GGP player with the required reasoning engine is also implemented. The player includes a model for the game tree as well as the sub objects like nodes plus edges and a possibility to add new strategies by creating new implementations for the given strategy java interface. A HTTP based connection interface to a dedicated GGP server is also available.
The reasoning is completely done in Prolog, including the parsing of the GDL description files required to create the Prolog rules. [24]
The program was heavily extended to enable the use of the move effect information and to increase the overall speed.

5.2.1 Changes to the Java program and parallelism

Most of the methods required to facilitate the move effect information are included in the UCT strategy, as explained in chapter 5.1. The only required change to the given core program was an extension of the game tree model to include the Gorgon information together with the move information. The move information is saved per player and state to enable simultaneous move games. A real move is always a combination of one move command for each player. More complex changes have been introduced to accelerate the program, namely by implementing parallelism. The original program was only running in a single thread, not making use of the multi core architecture of modern computers. To enable the random searcher decoupling, as explained in chapter 5.1, several changes where required, because decoupling the search affects several other parts of the program. The search is moved to a new thread, which is a permanently running child thread that is only updated when a new random search is called. Of course, there are multiple of these searcher threads, depending on the number of cores the machine has. When a searcher is finished, the thread enters a sleep loop that runs as long as no new search data is set. This prevents the thread from being terminated. It is required to reduce the thread management to a minimum, because thread creation and termination is a computationally heavy problem [4] and can affect scheduling and wakeup of threads. Every one of these searcher threads has its own Eclipse CLP instance for calculating the next states, moves and gorgon information. The main program has also its own instance for required calculations within the UCT part of the game tree. Concurrency queues that implement the required synchronisation blocks are used to communicate with the main program.

The first problem that arises by implementing these parallel searcher threads is the UCT selection mechanism. The UCT selector, picking the next states for random expansion, is continuing to run when a random search was started. But as long as no random search was finished, there is no UCT information available, hence starting a high amount of searcher threads is not a good idea. For a good trade-off between processor usage and availability of the UCT data, (2*CoreCount)-1 searcher threads are used. Together with the main thread every core is now running two threads, resulting in a near 100% core load, but with the currently average amount of 2-4 cores per CPU it's still a low number of 3-7 searches that are started before UCT information is returned. This is a good value because most games usually have more than 7 possible child states anyway, and UCT is always selecting unvisited states first.

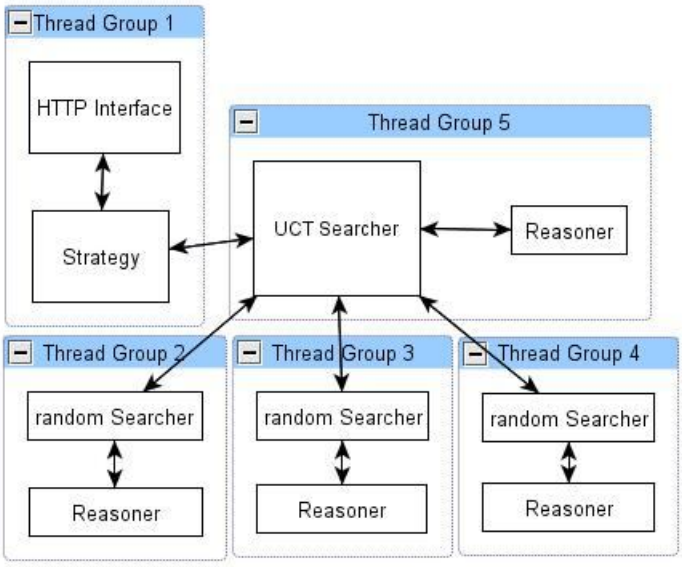


Figure 3: Thread distribution

For selecting the unvisited states, it is only required to save which states have been called for search, but not the results of the random expansion.

The real amount of threads is higher, as shown in figure 3, but the sub threads are coupled with blocking calls, and can therefore be considered a single thread and the interface/strategy thread is idling as long as no new data is passed from the server.

The next change now is required because of the random searcher threads blocking the main UCT searcher when all searchers are working because the UCT searcher waits for a chance to pass a new random search node to a free searcher thread. The UCT searcher is also calculating the best concrete move that should be passed back to the server, but as long as the thread is blocked, this calculation cannot be done. To overcome this problem, the best move calculation is called every second, starting 10 seconds before the current move timer runs out. This calculation can also be called by any of the sleep loops of the random searcher threads and the UCT searcher to guarantee that the best move is set, no matter at which point the program execution is at. The strategy thread will wakeup 750ms before the move timer runs out to respect any delay within the communication and for reading the best move from the UCT search. By only reading a variable, the strategy is not required to wait for method synchronisation with the UCT searcher. This is needed, because it is possible that it takes more then 750ms for a synchronisation point to come up, which would result in the strategy not being able to return a valid move in time. The Java sleep method is also not very reliable, because it depends on the internal OS scheduler, which is not giving any guarantees for time boundaries of thread switches, like wakeup. Therefore the Quartz [30] scheduler was used for polling the strategy thread, giving a much more reliable result.

The Basic Player is also offering a possibility to delete unused nodes, by marking them accordingly and the program will delete them later on. To have a more controlled method for removing nodes, this build in function was not used but replaced with a direct deletion within the UCT strategy. When the program is stepping forward to the next node, after the server transmitted the new moves, all branches of the game tree that are not starting with the new node are removed. By

deleting these irrelevant nodes immediately, more memory is available for new nodes and the more useful nodes are part of the game tree, the faster the tree exploration is, because the already saved nodes do not need to be calculated again. It is still required to verify the state of the game node can be deleted as well to prevent the deletion of states that are also part of other game nodes. But this mechanism is already provided by the Basic Player itself.

5.2.2 Changes to the Prolog program

The Prolog program, reliable for the parsing GDL rules and reasoning was modified to include the complete solution of the predicate. Usually Prolog does not save the clauses that have lead to the solution, but only the variable bindings of the called predicate. Everything else is hidden from the predicate call and is only available when debugging. To have the clauses ready without activating time consuming debugging functions, the GDL parsing was modified to extend all predicates with a variable returning the clauses of the predicate. In fact, two variables have been used, because of the way Prolog works. The first variable is use as an input bound to a list of the already seen clauses within the rule. The second variable concatenates the current clause with this list. It is not required to add predicate calls to the rule itself for creating the solution list which makes the process faster and easier, because it is not necessary to differentiate between “real” clauses and those list creation clauses. Any rule looks like this:

```
RuleHead(AVar,…,InVar,[[RuleHead(AVar,…),OutVar]|InVar])                    :-
APredicate(AVar,…,[],TmpOutVar1),BPredicate(AVar,…,TmpOutVar1,TmpOutVar2),…,NPredicate(A
Var,…,TmpOutVarN,OutVar).
```

The result is a nested list with a new level for each rule:

```
[RuleHead(AVar,…),[APredicate(AVar,…),BPredicate(AVar,…),…,NPredicate(AVar,…)]]
```

It can easily be flattened into a normal list, representing the complete clause list. The flattening is done afterwards to be able to detect recursive clauses for deleting them together with all sub clauses, which is required because the recursive clauses are not part of the indexing tree
(-> Chapter 4.7.1). The overhead created by this extension is near to none, because all information is available and has just to be added to the result list. This parser extension is pretty simple, but has to include a few special cases:

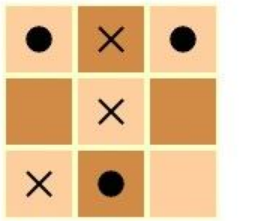
- *Infix predicates:* Predicates like `A \= B` are having a fixed artiy of two and are therefore not extended. They are just ignored.
- *NOT predicate:* Those predicates are only successful when no binding is available. This results in the clause list not being bound for successful calls. The predicate is also ignored.
- *OR Predicate:* The predicate has two sub clauses and is an infix predicate, but the solution of at least one sub clause will be relevant. In fact, it is exactly one sub clause, because the solver will not test the second predicate if the first one is already solvable. The parser is extending both sub clauses directly, ignoring the OR predicate.

For the solvability checks, it was also required to change the Prolog member predicate that is used to test if a fact is the member of the current state within the ***true*** clauses. To be exact, the member predicate tries to unify any entry of the list with the given fact. If a variable is included in the list, every fact can be unified with this variable. This was removed because the state list will be unbound when calling a solvability check.

Another change is the marking of all GDL predicates as dynamic. It is required to differentiate between GDL predicates and build-in predicates, to be able to decide which clauses should be expanded within the CNF creation phase (-> Chapter 4.1). Dynamic predicates can be detected with a single call, making it fast and simple.

5.3 Analysis Tool

It is very difficult to analyse the behaviour of a GGP player, because it is dependent on many different aspects of the algorithm. Additionally, the performance of a player can only be determined after a high amount of matches, because there is a high amount of randomness in the process that can lead to bad results for single games even if the overall performance is good.



REMAINING STATE:
(CONTROL XPLAYER)

PLAY CLOCK:
REMAINING: Inactive (Not the current step)
PLAYERS:

XPLAYER*	GORGONPLAYER
OPLAYER	FLUXPLAYER_TEST

HISTORY:

	XPLAYER	OPLAYER
6.	NOOP	(MARK 2 1)
5.	(MARK 2 3)	NOOP
4.	NOOP	(MARK 3 3)
3.	(MARK 1 1)	NOOP
2.	NOOP	(MARK 1 3)
1.	(MARK 2 2)	NOOP

Figure 4: TicTacToe visualization form Dresden GGP. [34]

General mistakes in the behaviour can be found by analysing the choices the player made in concrete games. The Dresden GGP Server is offering visualization for most games because of this reason [34]. But this is only a possibility to find larger bugs that are resulting in a wrong convergence that will have a visible effect in nearly every game, or for every match of a specific game. It is also still problematic to find the mistake in the source code, because of the multiple dependencies. For example, the move selection is dependent on the UCT values, the heuristic values and the result of the probability selection.
To be able to find those complex bugs, as well as to be able to retrieve quantifiable values easily, an analysis tool was implemented. It is a wrapper for a core component of the model: The game class that is representing a single game with its game tree and the reasoner connections. The analyser is able to work in two different modes: a slim, text based and a full visual analysis. The text based mode is used for measurements mainly, because it requires much less resources then the full visual analyser. The performance drawback is small enough for the analyser to be used in real matches.

The information available is:

- Runtime in MS: The time the program used in a single step.
- Next node calls (per minute): Amount of next states that got reasoned by the Prolog program. Next node calls to nodes that are already part of the tree are not included.
- Node count: Amount of nodes in the game tree.
- Node hash count: Amount of key entries in the node hash table. A difference from the node count is marking a wrong deletion.
- State count: The amount of states in the game tree. Should not be higher then the node count.
- State hash count: Amount of hash mappings between the states and the nodes. Can differ from the state and the node count, but not be higher then either.
- Terminals seen: Amount of finished simulation runs.

The visual version of the analyser is also including a view of the game tree. Besides the ability to view the concrete state and the moves that have lead to it, it is also adding parts of the UCT and creation information. The additional information shown by this is (as node label, from left to right):

- Node ID: Number of the node, sorted by the time of first visit.
- Goal value: For terminal and perfect choice nodes.
- Action visit count: Number of times the action has been selected. A move is an action for each player.
- UCT value: The average outcome of the state/action pair for each player.
- Node visit count: Number of times the node has been visited.
- Node shape: Terminal nodes are shown as circles.
- Node color: Terminal or perfect choice nodes are red.
- Node border: In alternating turn games, the current players turn have a green border.

The visit count and UCT value data is only available within the UCT part of the game tree, because the random expansion is not collecting this data. The visualiser also allows the viewing of game tree from older states to view differences between the trees. The drawback is that the game tree has to be stored for every step, resulting in high memory consumption; therefore it is only used for debugging and not for real matches. The visualisation itself was created with the Jung2 Framework for Java [22].

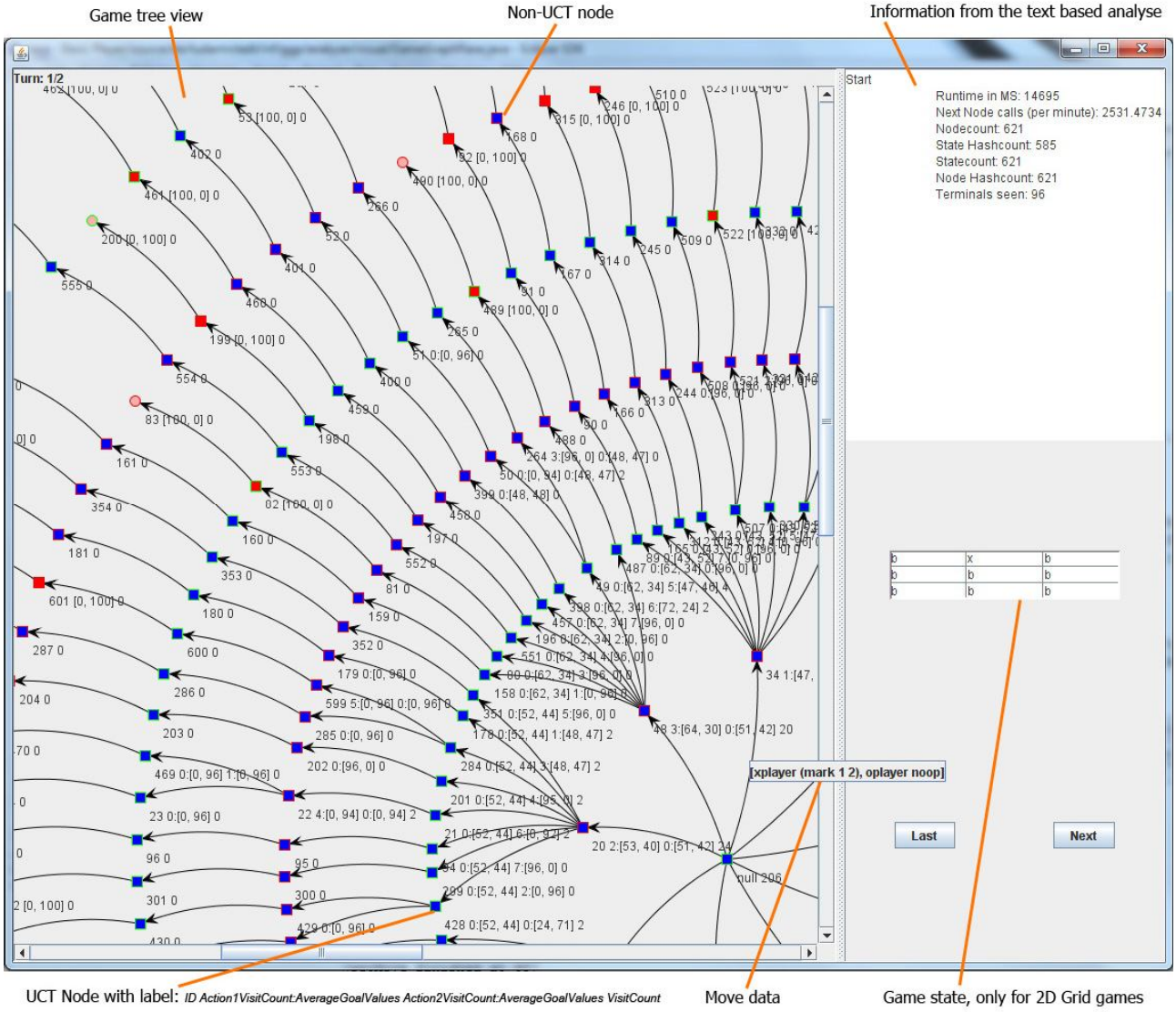


Figure 5: Visual analyser

6. Results and Conclusion

To examine the direct effect determination, several tests have been performed. The first set of tests is showing a time comparison between the normal state calculation and the Gorgon enhanced version. It verifies that the Gorgon effect determination is faster then a theoretical calculation of all next states which would be required to determine the move effects the classic way. The time for comparing both states is not considered, because the Gorgon version has to be a lot faster than the classic version to not slow down the UCT convergence too much. A state comparison requires much less time than the state calculation itself, hence the move determination has to be faster than the next state calculation anyway.
In a real GGP agent, the reasoning for next states and legal moves takes the biggest chunk of the available calculation time, but not all of it of course. Besides model management and the UCT algorithm itself that are identical for both the classic and the Gorgon version, the Gibbs affects runtime behaviour. Therefore also a direct time comparison was performed, testing the amount of nodes visited in both implementations. Those results are more important but less transferable to other theoretical GGP agents, because they are dependent on the algorithm and the concrete implementation.
Besides those pure runtime checks, it was also tested how the agent is performing in a real GGP environment. Several games from the Dresden GGP Server [34] have been played in a setup with the Gorgon enhanced agent against the base version. It shows how the performance drawback is affecting the result, together with the better move heuristic. The games have been picked with one topic in mind: They need to have move effects that are not already part of the move command. The Gorgon agent will not be able improve over the classic version when playing games where the move command and the effect are identical. No additional information is made available by the move effect determination, but the time consumption of this will result in a worse convergence. Examples for this problem are games like TicTacToe and Nim4 because they are only adding a single fact to the game state for every move, and this fact is the move command. The used games and why they could show a better solvability with the Gorgon enhancements will be explained later on. This is required for an in depth analysis of the pro's and con's of the Gorgon player. To achieve balanced results, games of different types have been selected: From single to 4 player games. From asymmetric, alternating turn games to parallel, simultaneous games.
All games that have been tested are part of the Dresden GGP Server games library [34] and have been written by different authors. All games are also played in a continuous tournament, hosted on the server, used for determining the performance of various GGP agents. Hence they are a good basis for the comparison.

3pffa: A three player game that allows spawning a new piece on specific positions, relative to the already placed pieces, or the capturing of an enemy piece by moving an already existing one. The goal is to capture as many pieces as possible and the Gorgon player is able to detect those capturing moves.

Blockerparallel: Asymmetric game where one player has to create a bridge of touching pieces that is connecting both sides of the game board. The other player has to prevent that. The game is played on two boards simultaneous and the Gorgon player can differentiate the effects for both boards.

Breakthrough: Both players have to reach the other side of the board or capture all enemy pieces with diagonal moves only. The differentiation between capturing and non-capturing moves is possible.

Gridgame: The players have to reach the other side of the board, but only one piece each is available and the move command is the direction only. The Gorgon effect determination returns the next cell that will be moved to.

Guess: The first player is choosing a number and the second player is replying with “higher” or “lower” to define which part of the number series should be removed, relative to the returned number. The number chooser wins if numbers are still left after 12 turns. The determined effects are the numbers that are getting removed.

Mummymaze2p-comp2007: An explorer has to reach the exit in a maze while a mummy, with double the step amount, tries to capture him. The move commands are directional only and it is possible to retrieve the resulting position of the players.

Pancakes6: A singleplayer game in which a list of 6 constants have to be sorted into the right order. A flip command for each position is available, reversing the order of all constants from the first to the given position. The determined effect is the new order.

Pointgrab_state: The player has to select the right move out of six in 30 different steps. To increase the difficulty, the game is played as a simultaneous move two player game. The Gorgon player can detect the goal increasing effect of the right moves.

Skrimish2: A simplified chess game with only 1 rook,2 knights, 4 pawns and a king. The Gorgon move effect is differentiating capturing from non capturing moves.

Smallest_4player: Each player has to select a number and the lowest number that has only been selected by one player wins. The effect determination is returning 1 as a move differentiating from all others, because it will win as long as the other moves are not known.

Games with two or more players have been evaluated with switching roles to guarantee that role based differences in the goal values are not affecting the result. This is the case within asymmetric games and most games have also a slightly higher win chance for the starting player.

6.1 Results

The tests have been performed on different systems, because of the high amount of matches required for a statistical relevant result. For the time and node expansion comparison, as well as the preparation time determination, the following machine was used:

- Intel Core2 Duo P9500 2,53Ghz
- 3 GB Memory
- 32Bit Windows XP Professional SP3

The real matches have been performed on two identical computers with lower specifications:

- Intel Celeron 3Ghz
- 1 GB Memory
- 32Bit Debian Linux, 2.6.30.3 Kernel

The following table 3 is showing the results of the time comparison. The legal column contains the time used for calling the ***legal*** predicate, which returns all legal moves. The second column presents a value for calling the ***legal*** predicate, including the Gorgon effect determination. The difference is the major factor for the worse node expansion count of the Gorgon system, but it is very game dependent. Especially complex ***next*** clauses will increase the time required for the effects determination. The third column considers a complete state calculation. This value has to be higher than the gorgon value for the determination to be effective. If this is not the case, it is possibly faster to calculate the next states and compare them. The last columns are showing the difference between the gorgon effect determination and the complete next state calculation, resp. the difference between the legal move calculation and the added gorgon effects. The Gorgon/State difference is the most important value, because it is showing the real speedup of the proposed system. All values are the sum of 10000 runs.

Gamename	legal	legal+gorgon	legal+states	Difference legal/gorgon	Difference gorgon/states
3pffa	2.4s	6.0s	15.5s	+ 150%	- 61.3%
Blockerparallel	23.4s	66.5s	---**	+ 184%	- 99.9%
Breakthrough	5.4s	9.0s	600s*	+ 67%	- 98.5%
Gridgame	1.9s	3.6s	79.5s	+ 89%	- 95.5%
Guess	3.7s	9.4s	424.9s	+ 154%	- 97.8%
Mummymaze2p	1.3s	2.7s	4.3s	+ 108%	- 37.2%
Pancakes6	0.5s	1.5s	0.5s	+ 200%	+ 200%
Pointgrab_state	0.7s	3.7s	27.0s	+ 229%	- 86.3%
Skrimish2	3.2s	8.8s	250s*	+ 175%	- 96.3%
Smallest_4player	2.7s	7.1s	2300s*	+ 163%	- 99.7%

Table 3: Time comparison results (*approximated with 1000 runs, **several hours)

It is easy to see that the Gorgon effect determination is faster than a next state calculation in nearly every game, but the runtime increase compared to the legal only calculations is still high. To get a more practical result, the node expansion count in real scenarios was also tested. Table 4 is showing that the node expansion count is only loosely related to the legal and gorgon calculations.

Gamename	Expansion count base	Expansion count Gorgon	Difference
3pffa	11,9 k/min	4,1 k/min	- 66%
Blockerparallel	2,6 k/min	1,1 k/min	- 58%
Breakthrough	6,9 k/min	2,9 k/min	- 58%
Gridgame	12,1 k/min	5,0 k/min	- 59%
Guess	3,4 k/min	3,2 k/min	- 6%
Mummymaze2p	11 k/min	4,4 k/min	- 60%
Pancakes6	19,4 k/min	15,6 k/min	- 20%

Pointgrab_state	15 k/min	10,1 k/min	- 33%
Skrimish2	4,1 k/min	1,8 k/min	- 56%
Smallest_4player	9,8 k/min	2,7 k/min	- 72%

Table 4: Node expansion count (in thousand per minute)

It should be noted that the node expansion count is varying greatly, depending on the states selected for sampling, the amount of nodes in the game tree and more. The penalty for using the Gorgon effect determination is especially decreasing when the node expansion count is already high. More expansions are resulting in a larger game tree which requires more management. Pointgrab_state and Pancakes6 are good examples for this effect.
Besides the question of how fast the determination works, it is also relevant if the precalculations can be done in time. All tested games are having a very small preparation timeframe, as pointed out by table 5. The results are showing the preparation for a single reasoner, not including the time required to copy the data to other reasoner.

Gamename	**3pffa**	**Blockerparallel**	**Breaktrough**	**Gridgame**	**Guess**
Preparation time	0.83s	0.02s	0.03s	0.02s	0.01s
Gamename	**Mummymaze**	**Pointgrab_state**	**Skrimish2**	**Smallest**	**Pancakes6**
Preparation time	2.1s	0.02s	1.0s	0.02s	0.02s

Table 5: Preparation times

Of course the preparation is not working perfectly for all games, but for most. The main reason for a failing preparation is ususally not the time, but explicit ruleset problems. A too high preparation time was only encountered with highly recursive games, like racer [34]. If it is required to solve recursions to depth of 50-100 for multiple times, it will take several minutes on the used system, even with recursion solution caching. In some cases it was possible to speed up the precalculations by resorting the facts manually, but this is of course no acceptable solution.
Most games that are not working are failing the preparation totally or are creating an illegal effect determination tree. Some special cases, like the already mentioned indirect recursions (-> chapter 4.7.1), are not implemented explicitly because they are encountered rarely.

The last part of this chapter is presenting the results of the implementation of the proposed system into a real GGP agent, the Gorgon Player. The agent competed against a base version, with the same UCT core, but with only normal Gibbs sampling. The usefulness of Gibbs sampling itself was already shown [14]. Each game was played 100 times with a 15, 30 and 45sec turn clock, resulting in 300 matches' total. As start time clock, 60sec has been selected. All results have been collected in table 6, together with the average goal value difference. The first value is always the Gorgon enhanced player and the unoptimized player is used as base value, meaning if the Gorgon player is better, the average difference will be positive. An expectation value for the games can not be defined, because most of them are not zero-sum games.

Gamename	15sec turns	30sec turns	45sec turns	Average diff.
3pffa	30 : 20	33 : 20	33 : 18	+ 66%
Blockerparallel	25 : 39	22 : 39	23 : 37	- 39%
Breakthrough	5 : 95	7 : 93	4 : 96	- 94%
Gridgame	77 : 5	83 : 11	83 : 17	+ 636%
Guess	7 : 93	5 : 95	6 : 94	- 94%
Mummymaze2p	35 : 66	32 : 69	32 : 69	- 52%
Pancakes6	51 : 65	57 : 74	59 : 77	- 32%
Pointgrab_state	100 : 25	100 : 26	100 : 26	+ 290%
Skrimish2	42 : 42	47 : 49	52 : 49	+- 0%
Smallest_4player	17 : 75	13 : 63	12 : 52	- 78%

Table 6: Real machtes results

6.2 Conclusion

When regarding at the preparation time table (table 4), it shows clearly that the proposed system can be used with a very short start clock. In GGP, it is uncommon to have start clocks below 10sec to allow a limited amount of precalculations at least. The preparations are taking only a negligible amount of time in most cases, but even games with long preparation times, like Mummymaze2p-comp2007, will fit within the given time frame easily.

Considering the other results, they are showing a mixed picture. The performance gain compared to a complete states calculation is enormous. In most cases, the effect determination takes less than a tenth of the time required to calculate all successor states. This is especially noticeable for parallel turn games, like Blockerparallel and Smallest_4player. Parallel move games are having an extremely high branching factor in most cases, because all possible moves for both players can be combined. The Gorgon effect determination does not need to calculate the combination of all moves, but only for the single moves alone. The drawback is that the effect determination does not consider the interaction of moves in parallel turn games. Effects that are depending on both moves at once will not be determined correctly. This is especially problematic for Smallest_4player because without the other players move, choosing 1 will always be the best choice. UCT is also converging to this choice, which is a mistake and is resulting in the declining goal values for the game, as shown in table 5. For a better playing of Smallest_4player, an efficient opponent model is required to be able to predict the enemies' choice.

The effect determination is not faster than a complete state calculation for all games; of course, there are exceptions, like Pancakes6. This is always the case if the next terms are very simple and only consisting of few clauses and the legal CNF term is not limiting the effects. In Pancakes6, the next effect is only a resorting of atomic facts which can be done in a single unification step. Fact creating games, like TicTacToe or Nim4, would show the same behaviour. Additionally, single player games are only profiting little from the effect determination, because they only have to find a single, good path. This can also happen when sampling totally random. Those two properties combined are the reason why Pancakes6 is showing a worse behaviour with the Gorgon effect determination compared to the base version.

The biggest problem when using the Gorgon system is that the determination requires more than double of the legal moves calculation in nearly all cases, which is also resulting in a much lower node expansion count, as shown in table 4.
Hence the positive effect of the additional data made available by the effect determination has to compensate the time penalty. This is only possible in few games, namely 3pffa, Gridgame, Pointgrab_state and partially for Skrimish2. In 3pffa and Pointgrab_state, there can be a move available that is directly increasing a fact that is used to determine the goal value, which will be detected by the Gorgon determination. In Gridgame it is possible to detect if the player is getting closer to the target cell, and in Skrimish2 Gorgon is determining capture moves which are foten resulting in a higher victory probability.
All other games are showing a worse behaviour when using Gorgon, because the improved Gibbs sampling heuristic is not affecting the UCT convergence enough. It can be concluded that the Gorgon effect determination is working fast and good, but it is only suited for the use as Gibbs sampling information if the effect facts are playing a major role in the goal value calculation.

6.3 Further Work

In the current version, the Gorgon move effect determination is not very useful, but there are several possibilities to enhance the available information or make more complex use of it. For a more universal approach, the effect determination should not be activated for all games. This decision should be performed by several different parameters. The most obvious one is the difference between the gorgon extracted move data and the move command itself. When creating a mapping between those two, it would be possible to detect if the extracted data has a higher information value then the move command, by comparing the information. For the Gorgon determination to be activated for the whole game there should be a relevant difference from a 1:1 mapping, meaning that there are moves that have multiple or different gorgon commands in different states. The second possibility to determine when the Gorgon enhancement should be used is the goal value. If a running simulation with activated Gorgon sampling is returning a higher average goal value then the base version, it should be used all the time for the current game. It would also be possible to measure the difference of the node expansion count, but the threshold for the difference would be varying every game, making it difficult to determine if the Gorgon enhancement should be used without running several games forehand.
But to make really good use of the given determination scheme, it is required to not only improve the current system, but increase the information value. The results have shown that the currently extracted data is not worth the time penalty in most scenarios when used for Gibbs sampling. It is possible to improve the given data is itself by extending the CNF resolution to the goal predicate. This would result in a list of all facts required to gain a specific goal value. By creating a list of all state dependent facts and the goal values they can result in, it is possible to determine which facts should be instantiated to reach a higher goal value. The goal value for each fact is the highest value where the fact is element of the goal predicate CNF term. There is an exception for facts that are also occurring negated in another goal solution. If the negation has a lower maximal goal value than the fact itself it is limiting the goal value, hence the associated value should be the lowest goal value it is participating in. It should be noted that it is also possible for the negation to have the higher maximal value which would require the fact to be remove for improving the goal. A direct determination, whether the Gorgon move effect is creating or removing a fact that is helping the goal value, would already improve the result a bit, but most games are only having

very few facts that are directly involved in the goal predicate. A better way is calculating the editing distance of the new fact to the goal value facts, as already used in some heuristic agents [35]. Of course they are not comparing to the determined effects but to all facts of a state.

The best idea is possibly testing the presented move effect determination with heuristic players, because they are not as speed dependent but they are calculating every next state for selection reasons often. As long as the difference is enough for the heuristic algorithm to function it is an interesting possibility, but it will be difficult to extend the method to a complete next state calculation algorithm, because it would have to be a 100% robust. A small error in an early state would be cascading down to the terminal state, rendering the complete simulation useless, but it is not possible to detect those errors without creating the real next states as comparison.

The most promising, but also most complex usage of the effect determination would result in an abstract state representation scheme. The connection from a precondition fact to the effect can be seen as an edge in a state which can have different types. The simplest type is the change of a fact that can be further differentiate by the ownership of the fact, as presented in chapter 4.3. If the ownership is not changing, it is a “move”. An ownership change should be marked as “capture” or “loss”, depending on the direction of the ownership change. By also including negated conditions as fulfilled, it is also possible to have “blocking” edges. This state representation graph can be further extended by also including the edges of the following states to get a more detailed picture of the state. An example state graph is given in figure 6. The next state (opponent moves) edges are created by simply calculating legal moves with a changed current role.

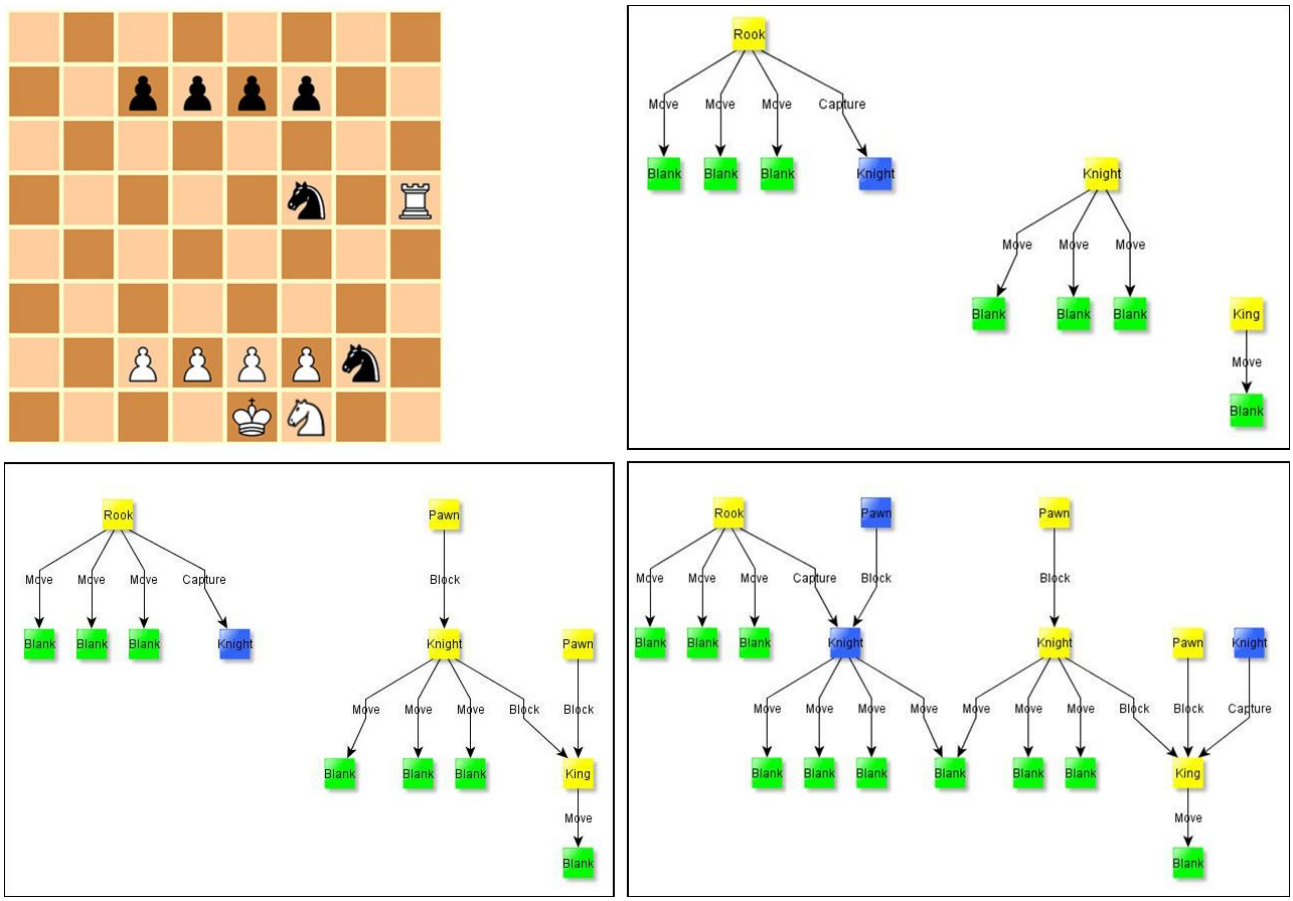


Figure 6: State graph example (white turn), clockwise: 1. Skrimish2 example state, 2. Base graph, 3. Base graph with blocking Edges, 4. Base graph with blocking Edges and opponent moves.

It is very interesting to have an abstract graph representation for states in general, because this allows the calculation of patterns. An idea that has already been facilitated but with much less abstract game representations [25]. It also enables a knowledge transfer from already sampled states to new seen states, by determining a degree of similarity. The similarity of a fact base, which can be calculated easily, is not a good concept of similarity because they are position dependent and hence not considering symmetries or reoccurring patterns at other positions. An abstract graph representation is much better suited for similarity calculations.
While using something like the Gorgon effect determination in a production quality system, it would also be required to implement all special cases. As example is the high recursion depth problem, mentioned in chapter 6.1, which could be solved with by automated term ordering. When sampling a reasonable amount of recursive solutions (out of real game states), it is possible to sort the facts by the number of successful unifications. It is even possible to determine relevant factors (like node depth) to dynamically sort the facts for each state.

7. Summary

UCT is currently the most successful algorithm for GGP agents, but it is dependent on a high state sample count for convergence. Heuristic players that are facilitating domain knowledge or abstract features for determining good states are not as powerful but can best UCT in some domains. An efficient move effect determination scheme would enable GGP algorithms to use abstract features more easily, because it is not required to calculate the successor states for comparing state features. The Gorgon effects determination method is showing that it is possible to calculate move effects fast, compared to the complete state calculation, when using an optimized system that is also including advanced indexing schemes and efficient solving algorithms. It is especially interesting that the Prolog solver is fast but inefficient. Several games are also requiring special case handling, for problems like recursions. Implementing a parallel agent is also not trivial, because of the unreliable scheduling used in modern operating systems. The Gorgon system is presenting solutions to nearly all of these difficulties but it is still slowing down UCT too much for a universal use. Only games with special properties, like move effects that directly influencing the goal value are profiting enough for showing an improved result in real agents. The proposed system should be tested in an algorithm dependent on comparisons to all successor states, like some heuristic agents are doing it. Those algorithms should benefit greatly from the fast effect determination. It is also possible to improve the extracted Gorgon information or using it for creating advanced state representation schemes.

8. Appendix A – Algorithms, Figures, Listings and Tables

9. Appendix B – Symbols and GDL primitives

$Q(s,a)$: Average reached goal value by of the action a in state s
$N(s,a)$: Visit count of the action a in state s
$P(play(a))$: Gibbs propability for selection playing action a
$Q_h(a)$: Average reached goal value for action a
τ : Gibbs temperature for flattening the probability distribution

init(X): The fact X is true in the initial game state.
role(X): the constant X is a player role
true(X,S): the fact X is true in the game state S
legal(P,X,S): the move command X is legal for player P in state S
does(P,X): player P has selected move command X
next(X,S): the fact X is will be true in the successor state of S
terminal(S): true when S is a terminal state
goal(V,P,S): player P has reached goal value V in the (terminal) state S
distinct(X,Y): fact X is not unify able or equal with fact Y

10. Bibliography

1 V. Allis, 1994, *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, ISBN 9090074880

2 Association for the Advancement of Artificial Intelligence, http://www.aaai.org, visited 22.9.2010

3 P. Auer, N. Cesa-Bianchi, P. Fischer, 2002. *Finite-time analysis of the multiarmed bandit problem*. Machine Learning, vol. 47.2 ,pp. 235–256

4 J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, J. Weglarz, 2001, *Scheduling Computer and Manufacturing Processes*, Springer, Berlin, Germany, ISBN 3540419314

5 B. Bouzy, B. Helmstetter, 2003, *Monte-Carlo Go Developments*, Advances in Information and Communication Technology of the International Federation for Information Processing (ACG-IFIP), vol. 263, pp. 159-174

6 B. Bruegmann, 1993*, Monte Carlo Go*, ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z

7 S. Ceri, G. Gottlob, L. Tanca, 1989, *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*, Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering (IEEE-TKDE), vol. 1, no. 1, pp. 146-166

8 N. Chomsky, 1956, *Three models for the description of language*, Transactions on Information Theory, vol. 2, pp. 113–124, Institute of Radio Engineers (IRE)

9 J. Clune, 2007, *Heuristic Evaluation Functions for General Game Playing*, Proceedings of the 22nd national conference on Artificial intelligence (AAAI), vol. 2, pp. 1134-1139, Vancouver, Canada, ISBN 9781577353232

10 E. Cox, M. Genereretht*, http://games.stanford.edu/*, visited 22.9.2010, Stanford University

11 Eclipse CLP Prolog, *http://eclipseclp.org/,* visited 9.8.2010, Cisco-style Mozilla Public License 1.1

12 http://www.eclipse.org/, visited 22.9.2010, The Eclipse Foundation

13 H. Feng-Hsiung, 2002, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*, Princeton University Press, USA, ISBN 0691090653

14 H. Finnsson, Y. Björnsson, 2009, *CADIA-Player: A General Game Playing Agent*, Institute of Electrical and Electronics Engineers Transactions on In Computational Intelligence and AI in Games (IEEE-CIAIG), vol. 1, pp. 4-15

15 H. Finnsson, Y. Björnsson, 2008, *Simulation-Based Approach to General Game Playing,* Proceedings of the 23rd national conference on Artificial intelligence (AAAI), vol. 1, pp. 259-264, Chicago, Illinois, ISBN 9781577353683

16 H. Finnsson, G. Guðmundsson, *http://cadia.ru.is/wiki/public:cadiaplayer:main*, visited 22.9.2010, School of Computer Science, Reykjavik University, Iceland

17 P. Flach, 1994, *Simply logical: intelligent reasoning by example,* pp 202-205, John Wiley & Sons Inc., New York, USA, ISBN 0471941522

18 S. Gelly, D. Silver, 2007, *Combining online and offline knowledge in UCT*. Proceedings of the 24th international conference on Machine learning (ICML), vol. 227, pp 273-280, Corvalis, Oregon, ISBN 9781595937933

19 M. Genesereth, N. Love, 2005, General *Game Playing: Overview of the Association for the Advancement of Artificial Intelligence Competition* (AAAI), AI Magazine, vol. 26, pp. 62-72

20 P. Graf, 1994, *Substitution Tree Indexing*, Max-Planck-Institut der Informatik, Saarbrücken

21 C. A. R. Hoare, 1962, *Quicksort,* Computer Journal, vol. 5, pp. 10–15

22 http://jung.sourceforge.net/, Jung2 Graph Framework, visited 22.9.2010

23 M. Kearns, Y. Mansour, A.Y. Ng. 1999, *A sparse sampling algorithm for near-optimal planning in large Markovian decision processes*, Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI), pp. 1324-1331

24 I. Keller, S. Schiffel, *Basic Player Framework, http://palamedes-ide.sourceforge.net/start.html,* visited 9.8.2010

25 M. Kirci, J. Schaeffler, N. Sturtevant, 2010, *Feature Learning Using State Differences,* Department of Computung Science, University of Alberta, Edomonton, Canada

26 L. Kocsis, C. Szepesvári, 2006, *Bandit based Monte-Carlo Planning,* European Conference on Machine Learning in Lecture Notes in Computer Science (ECML-LNCS), vol. 4212, pp. 282-293

27 R. Levinson, 1995, *General Game-Playing and Reinforcement Learning*, University of California, Santa Cruz (UCSC-CRL), Computational Intelligence, vol. 12, pp 1-221, Santa Cruz, USA

28 N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth, 2008, *General Game Playing: Game Description Language Specification,* Stanford University, Stanford

29 J. Mehat, T. Cazenave, 2008, *Monte-Carlo Tree Search for General Game Playing,* Université Paris 8, France

30 http://www.quartz-scheduler.org/, visited 22.9.2010, Terracotta Inc.

31 J. Reisinger, E. Bahceci, I. Karpov, R. Miikulainen, 2007, *Coevolving Strategies for General Game Playing*, Proceedings of the Institute of Electrical and Electronics Engineers Symposium on Computational Intelligence and Games (IEEE-CIG), pp. 320-327, Piscataway

32 S. J. Russell, P. Norvig, 2003, *Artificial Intelligence: A Modern Approach*, 2nd ed., Upper Saddle River, New Jersey: Prentice Hall, ISBN 0137903952

33 J. Schaeffer, 1989, *The History Heuristic and Alpha-Beta Search Enhancements in Practice*, Institute of Electrical and Electronics Engineers Transactions on Pattern Analysis and Machine Intelligence (IEEE-TPAMI), vol. 11, pp. 1203-1212

34 S. Schiffel, M. Günther, *http://euklid.inf.tu-dresden.de:8180/ggpserver/index.jsp*, visited 9.8.2010, TU Dresden

35 S. Schiffel, M. Thielscher, 2007, *Fluxplayer: A Successful General Game Player,* Proceedings of the 22nd national conference on Artificial intelligence (AAAI), vol. 2, pp. 1191-1196, Vancouver, Canada, ISBN 9781577353232

36 J. Slagle and J. Dixon, 1969, *Experiments with some Programs that Search Game Trees*, Journal of the Association for Computing Machinery (ACM), vol. 2, pp. 189-207.

37 *The Java Language: An Overview,* 1995, http://java.sun.com/docs/overviews/java/java-overview-1.html, Sun Whitepaper

38 B. Walsh, 2004, *Markov Chain Monte Carlo and Gibbs Sampling*, Lecture Notes for Ecology and Evolutionary Biology (EEB) 581, version 26