Common Fate Graph Patterns in Monte Carlo Tree Search for Computer Go

Tobias Graf
University of Paderborn
International Graduate School "Dynamic Intelligent Systems"
Email: tobiasg@mail.upb.de

Marco Platzner
University of Paderborn
Email: platzner@upb.de

***Abstract*—In Monte Carlo Tree Search often extra knowledge in form of patterns is used to guide the search and improve the playouts. Shape patterns, which are frequently used in Computer Go, do not describe tactical situations well, so that this knowledge has to be added manually. This is a tedious process which cannot be avoided as it leads to big improvements in playing strength. The common fate graph, which is a special graphical representation of the board, provides an alternative which handles tactical situations much better. In this paper we use the results of linear time graph kernels to extract features from the common fate graph and use them in a Bradley-Terry model to predict expert moves. We include this prediction model into the tree search and the playout part of a Go program using Monte Carlo Tree Search. This leads to better prediction rates and an improvement in playing strength of about 190 ELO.**

I. Introduction

While Monte Carlo Tree Search (MCTS) [1] can theoretically be used without any knowledge of the problem, it is of high benefit to additionally guide the search by expert knowledge and to bias the playouts towards logical moves.

This knowledge is condensed in prediction models [2] [3] [4] which predict good moves for a given board situation. They can be learned off-line from a large set of expert games and then be applied in MCTS.

In Computer Go knowledge is provided by shape patterns [2] which match the local neighborhood around a possible move. These patterns are popular as they are not only efficiently computable but also capable of representing Go-knowledge quite well. Especially in playouts where speed is of utmost importance shape patterns with a very small size are used.

A disadvantage of shape patterns is the lack of tactical knowledge. Often the information if a group of stones can be, e.g., captured is outside the pattern resulting in bad predictions. In Figure 1 two atari situations are shown, i.e. black can capture the white group in each of them by placing his stone on the marked intersection.

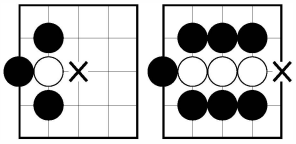


Fig. 1. Atari situations

A pattern would need to include all the stones to represent this simple situation. While small shape patterns might handle the left situation they cannot deal with the right one which is very similar. To recognize an atari situation in the right position the shape pattern needs to be very large which raises problems of generalization to new situations. Moreover, a 3x3 pattern of rectangular size centered on the move to play is a common type of pattern used in playouts for computer go [5] [6]. This type of pattern cannot represent the shown situations at all. The consequence is that tactical knowledge like the one shown above has to be added by hand. This is a very tedious and time consuming developing process limiting the amount of knowledge usable in MCTS.

In 2001 Graepel [7] introduced a new representation called the *common fate graph* (CFG) which uses a graph representation that merges connected stones of the same color into one vertex while empty intersections on the graph are not merged. Although the graph representation looses some information about the shape and size of groups it is much better suited to deal with situations like the atari shown above. The CFG of the atari situations are shown in Figure 2 with empty intersections represented as squares. As you can see all the information about the atari situation for both positions is included in a neighborhood of size two (i.e. only two edges in the graph away) which makes it easier to define local patterns.

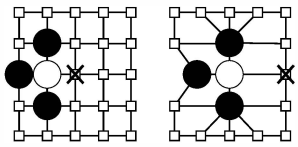


Fig. 2. Common Fate Graph of atari situations

The challenging point for an extensive usage in MCTS programs for Computer Go is the computational complexity of using a graph representation. Graepel [7] used a representation based on enumerated paths and Support Vector Machines as well as Kernel Perceptrons to classify good and bad moves of expert games. A later comparison [8] of different learning algorithms showed that complex non-linear learners (e.g. SVMs or Random Forests) are necessary to get the desired prediction accuracies which makes it difficult to use the CFG-representation in MCTS from a performance point of view. Moreover, SVMs or Random Forests are not easy to integrate into common move prediction models in Computer Go making an integration into a full MCTS program challenging.

978-1-4799-3547-5/14/1.00

While there are several publications which use the common fate graph for subproblems in the game of Go (e.g. [8] [9] [10]) to our knowledge there is no complete integration into a MCTS Go program. We therefore developed common fate graph patterns which address the above issues and integrated them into a Go program.

The contributions of our paper are:

- We show how to extract patterns from the common fate graph and how to integrate them into a move prediction model.
- We explain how to compute these patterns efficiently in an incremental way.
- We compare the prediction strength of the common fate graph patterns to the usual shape patterns applied in the playout and tree search part of MCTS.
- We conduct experiments to measure the overall gain in strength by playing against another Go program.

The remainder of this paper is structured as follows:

In section II we give a brief introduction into background information and related work on the common fate graph.

In section III we present the procedure of pattern-extraction from the common fate graph and how to use them in a Bradley-Terry model. To achieve a sufficient performance we show how to update the patterns incrementally.

In section IV we conduct several experiments for the common fate graph patterns. We quantify the prediction strength on expert games, evaluate the playing strength against another Go program and determine the efficiency of the incremental updates.

Finally, in section V we give an outlook and point to further directions

II. Background and Related Work

Shape patterns are the most common way to represent Go knowledge in MCTS programs. For a possible move on an empty intersection the whole local neighborhood up to a specific distance is included into a pattern. Figure 3 shows an example of all shape patterns from size two to ten as defined by Coulom [3]. A shape pattern of size three then means that the information (black, white, empty) of all the marked intersections with a number three or lower are included. One disadvantage of these patterns is that they only match the exact stone-configuration and thus do not generalize well to unseen situations as a pattern does not match if only a single stone changes. To address this issue one does not use only patterns of a fixed size but includes all sizes 2,3,4,... up to 10 for example. Thereby, often occurring situations can be handled by large patterns while a slightly different situation might still be matched by some smaller pattern.

Patterns can be used in MCTS in two different ways. First, the exploration in MCTS is guided by the prediction model to lessen the effect of the large branching factor of the game of Go. To focus the search either progressive widening [3] [11], which only searches a limited amount of moves, or progressive bias [11] [12], which artificially increases the value of good

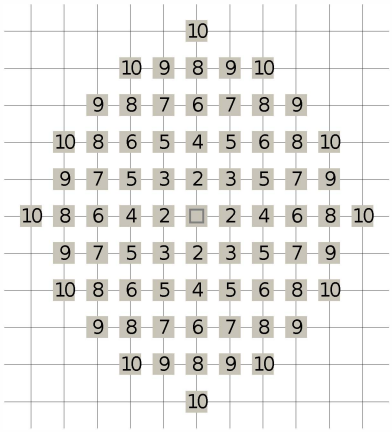


Fig. 3. Shape patterns from size 2 to 10 as defined in [3]

moves, can be used. Experiments in [11] show that this type of pruning the search tree is very efficient to achieve good results in Computer Go. The second way of using the prediction model is in the playout part of MCTS. Instead of randomly selecting moves the probability distribution of the prediction model is followed. This way of knowledge-use in playouts was e.g. examined in [13] where a new kind of learning algorithm was tested. For the 19x19 Go board it was shown that following expert moves in the playouts is an efficient way of achieving strong performance.

A game in Go which is played out until the board is full has about 450 moves. This means that the tree search part of MCTS in contrast to the playout part uses only a fraction of the time. This often leads to an asymmetrical use of patterns in MCTS when applied to Go. In the tree search part large shape patterns with costly extra features are used while in the playout part only small shape patterns with some fast local tactical features are applied.

An often used move prediction model in Computer Go is the generalized *Bradley-Terry model* [14]. In a Bradley-Terry model the probability to perform action a in state s (i.e. a move in a given position) is modeled by:

$$\pi(s,a) = \frac{e^{\phi(s,a)^t \theta}}{\sum_b e^{\phi(s,b)^t \theta}}$$

The patterns (also called features) for a move a are model by a binary vector $\phi(s,a) \in \{0,1\}^n$ where features can be active (1) or inactive (0) in a given position s. The weight vector $\theta \in \mathbb{R}^n$ is used to fit the model with the values $\gamma_i := e^{\theta_i}$ are also known as the gamma values of a certain feature. They are related to the ELO-Formula which is used in e.g. chess to describe the strength of players.

The same model is also known in reinforcement learning with the name *softmax-policy*. The difference is that in the Bradley-Terry model usually only binary features are allowed while there is no restriction for the features in the softmax-policy.

The common fate graph was first introduced in [7]. They used a SVM and Kernel-Perceptron with Gauss-Kernels to classify good/bad moves of tsumego problems and expert games. The author concluded that the feature representation

is so rich that the learning algorithm is not critical as he got similar results for SVMs and Perceptrons. In [8] this was tested for several learning algorithms. It was shown that only non-linear machine learning algorithms like SVMs achieve good results.

In [9] the *Enriched Common Fate Graph (ECFG)* was proposed. In the ECFG the labels of empty intersections are enriched by manually designed patterns while black and white groups get a weight assigned depending on their stability. They derived a kernel by enumerating all paths in the graph of a position with a maximum length and then trained a support vector machine to classify a position as winning or loosing.

The CFG was also used to determine subgoals in the game of Go. In [10] the author used path-features to determine if groups are connected (i.e. the opponent can or cannot cut the groups).

Our main work is based on the linear time graph kernel of [15] which, like the ECFG, enriches the labels. Starting on a graph where each vertex is labeled with one of a few labels they proposed a universal procedure to incorporate neighborhood information into each label. This is done by hashing the neighborhood labels and the own label of each vertex into a new label and then replace the label of the vertex by this new one. This can be done several times for the whole graph so that after each iteration the labels of the vertices contain more and more information of the local neighborhood. Based on these enriched labels they derived a kernel on graphs which can be computed in linear time.

III. Common Fate Graph Patterns

This section provides the details on how to create features from the common fate graph, how to update these features incrementally during a game and how to incorporate the features into a Bradley-Terry model.

The main idea is to enrich the labels of the common fate graph, like in the ECFG, but with the neighborhood hashing procedure proposed in the linear time graph kernel [15]. Then paths are enumerated and as they have an enriched representation they can be used as features directly without using a kernel. This speeds up computation and eases the integration into the Bradley-Terry model. The neighborhood hash has the advantage that it provides a universal way to enrich the labels of all vertices without any manual design.

A. Graph-Features

The feature extraction for a move is separated into three steps. First, the board is transformed into its corresponding CFG representation. The resulting graph has only three different labels which allows us to enrich them by a neighborhood hash in a second step. Finally, all paths starting from the empty intersection of the move are enumerated and treated as patterns.

1) CFG-Transformation: The board position is first transferred to a CFG representation as described by [7]. To achieve this each stone and empty intersection on the board is represented by a vertex and connected to its four neighboring vertices. This one-to-one mapping is also termed *Full-Graph-Representation*. To simplify this representation every black or white vertex is merged into a single vertex with all neighboring vertices of the same color. Vertices representing an empty intersection are not changed. The idea of this representation stems from the fact that according to the rules of the game connected stones on the board are either captured or stay on the board together. They thus share a *common fate*, the reason why this simplified graph representation is called *common fate graph*. A visualization of this process is shown in Figure 4(a) and 4(b) with a board position and its corresponding common fate graph.

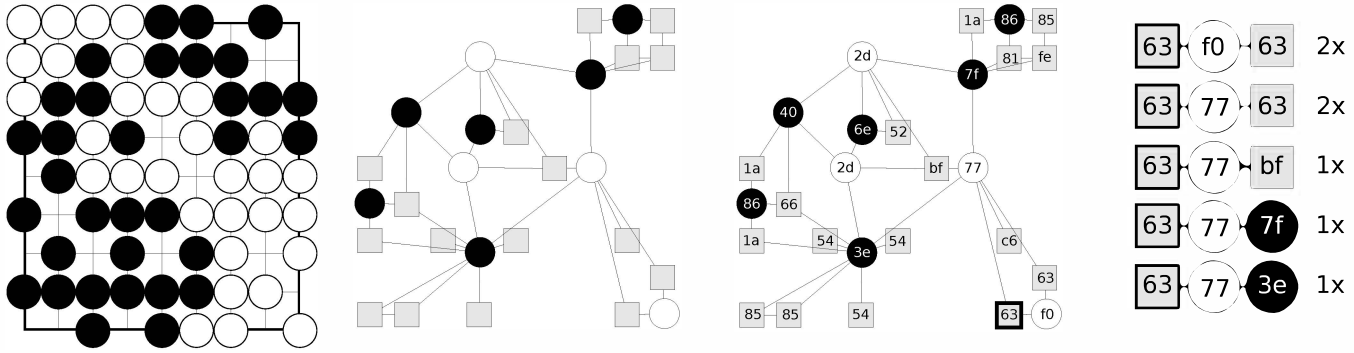
2) Label-Enrichment: In the basic CFG the vertices are just labeled with their corresponding type on the board: black group, white group or empty intersection. To increase the representational power the next step is to enrich the labels of the graph by a neighborhood-hash [15] so that the labels also incorporate information of their local neighborhood. In the neighborhood-hash in the first iteration each vertex-label gets a random (but fixed) value assigned. In the following iterations for each vertex the own label and the label-values of all neighbors are hashed and thus combined into a new label. This label replaces the vertex-label and a new iteration can follow which is based on the new labels of the graph. In this way the neighborhood of a vertex is hashed into the labels of each vertex. The number of iterations determine how much of the neighborhood is included in the hash value. The hashing algorithm for a vertex and its neighborhood is shown in algorithm 1.

Algorithm 1 Neighborhood-Hash

```
function ITERATE(v, iteration)
    if iteration = 0 then
        if v.color = black then
            v.label ← BLACK
        else if v.color = white then
            v.label ← WHITE
        else
            v.label ← EMPTY
        end if
    else
        label ← BIT-ROTATE(v.label)
        if v.liberties ≥ 4 then
            label ← label + CONST_4_LIBERTIES
        else
            for all n ∈ neighborhood(v) do
                label ← label + n.label
            end for
        end if
        v.label ← mix32Bit(label)
    end if
end function

function MIX32BIT(a)
    a ← a + (a << 15)
    a ← a ⊕ (a >> 10)
    a ← a + (a << 3)
    a ← a ⊕ (a >> 6)
    a ← a + (a << 11)
    a ← a ⊕ (a >> 16)
    return a
end function
```

In this algorithm we closely follow the procedure of the



(a) Board position (b) Common Fate Graph (c) Neighborhood Hash (d) Path Features

Fig. 4. Extraction of common fate graph patterns from a board position

linear time graph kernel [15] but with a slightly different hashing algorithm which uses summation of labels instead of the XOR operation. The reason for this change is that in our graphs a vertex has many neighbors with equal labels. If the XOR operation is used this leads to many obvious collisions (i.e. a group with one black neighbor and a group with three black neighbors get the same hash value). The original hashing algorithm of the linear time graph kernel [15] requires counting of equal labels to deal with this but this is too expensive to do in MCTS. By using a summation instead of the XOR operation these obvious collisions are avoided. To improve the quality of the hashes we use large random initial labels with 32 bit and mix all the bits after an iteration by an integer mixing-function proposed by Thomas Wang[1]. The mixing function assures that a change of some bits results in all bits being changed.

Additionally, we incorporate some domain knowledge and do not include the neighborhood in the hash value if a vertex has at least four empty intersections in its neighborhood (i.e. more than four liberties). This helps to reduce the number of different hash labels but still offers enough expressiveness to get good prediction results. An example of how the CFG looks like after 1 iteration of the neighborhood hash is illustrated (with 8 bit labels) in Figure 4(c). See e.g. the top white groups which both have the label "2d" as they have the same direct neighborhood of three black groups and one empty intersection.

3) Path-Enumeration: The last step consists of enumerating all paths of a certain length. Paths incorporate structural information into the patterns and were already used by Graepel to classify expert moves [7]. A pattern is centered on an empty intersection so each path starts there and is followed up to a maximum length. To simplify an incremental implementation we also allow cycles in this paths[2]. Paths offer a good representation as they naturally contain Go information, e.g. "play on an empty intersection next to an opponent stone in atari which is a neighbor of our own stone in atari". An example of a path-enumeration is shown in Figure 4(d). All paths starting from the marked intersection up to length two are enumerated. Please note that due to cycles also all paths of length one are included implicitly.

If we now want to get all features $\phi(s, a)$ for a move on intersection a we enumerate all paths starting from a in the label enriched CFG. The label of a path is the concatenation of all the vertex-labels the path consists of. For our features we only use paths up to length two so that the path labels consist of three vertex labels. Paths with length greater than two got better prediction results but were too slow to be useful for MCTS. From the vertex-labels only the lower 8 bits are used (which is a trade-off between memory consumption and accuracy) so that we have 2^{24} different types of paths as features.

After the enumeration of paths features are not binary as a path can be present multiple times. To include the path features into a Bradley-Terry model we need to transform them into binary form. This can be achieved by only using the existence-information, i.e. a feature is active if there is at least one path present and inactive if there is no path. The resulting feature vector in a given state s for action a is then:

$$\phi(s,a) = \begin{pmatrix} 0 \\ \dots \\ 1 \\ \dots \\ 1 \\ \dots \\ 0 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow \text{at least one path with index } i \\ \\ \leftarrow \text{at least one path with index } j \\ \\ \end{matrix} \begin{matrix} \leftarrow \text{no path with index } 0 \\ \\ \\ \\ \\ \\ \leftarrow \text{no path with index } 2^{24}-1 \end{matrix}$$

The feature vector is large but very sparse. Usually only 10-20 features are active.

B. Incremental Updates

If we want to use the CFG patterns in the playouts of a MCTS speed is one of the most important factors. To increase the performance of the tactical patterns we reduced the work to update them by exploiting their incremental nature.

For a new board position the path-features have to be enumerated for all empty intersections. We store the paths and their count in a hash table. Once we have this data we have to update it whenever a new stone is placed on the board. Due to the rules of the game this change is only locally. So one idea is to do a full update of all intersections in the neighborhood

[1]The original website of Thomas Wang is not online anymore but can be accessed by http://web.archive.org/web/20110807030012/http://www.cris.com / Ttwang/tech/inthash.htm

[2]In literature on graph kernels usually called "walks".

of the last move that could be affected. This saves a lot of work as only few intersections have to be updated and not the whole board.

Still we can save even more time as is illustrated in Figure 5. The black stone is placed on the board and the neighborhood-hash is updated. This results in all neighborhood-hashes to be changed on the marked intersections (gray dots). Looking at the intersection marked with a big X this results in only one path-change. It would be wasteful to enumerate all paths again for this empty intersection.

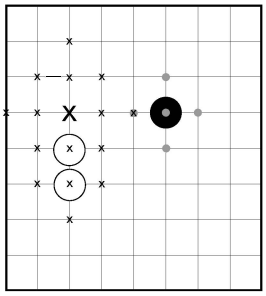


Fig. 5. Incremental Updates: Only few path-features are affected by a move

As we have a hash-table with all paths for this intersection we can just remove the old path and add the new one. So all that has to be done when a new stone is placed on the board is:

1) Update the neighborhood-hash
2) For each intersection:
   a) Remove all paths that have a changed label
   b) Add all new paths

This is the basic idea of the incremental path-features. It can be efficiently implemented as shown in algorithm 2 (we leave out how to update the neighborhood hash and the CFG).

Whenever a stone is placed on the board the steps in the function "PlaceStone" are executed. The key is to first determine which vertex-labels will be changed by a move without changing the graph. These vertices are remembered for later usage. As some vertices might disappear due to a group-merge or a capture this is split into a "remove" and an "add" set. This is done in the function "PredictLabelChanges". For example if a group is captured the group-vertex is added to the remove-set while all newly created empty intersections are added to the add-set. If all vertices remain the same and only the neighborhood-hash-label changes the vertex will just be added in both the remove-set and the add-set. Then we will first remove all paths which have a vertex in the remove-set. After that the changes to the CFG will be done and only in the finals step we will add all paths which have a vertex in the add-set.

The function "UpdatePath" will enumerate and update all possible paths with a vertex in the set s. The first block enumerates paths which have the vertex in the middle while the last block enumerates paths with the vertex at the beginning or the end. In this way all paths are updated which have a vertex in the given set. To avoid that vertices get updated several times only neighbors which are not part of the given set s are followed during some of the enumerations.

Algorithm 2 Incremental Update

```
function PLACESTONE(location)
    R = ∅
    A = ∅
    predictLabelChanges(location,R,A)
    updatePaths(R, remove)
    applyChangesToCfg(l)
    updatePaths(A, add)
end function

function UPDATEPATHS(S, operation)
    for all g ∈ S do
        for all a ∈ neighbors(g) \ S do
            for all b ∈ neighbors(g) \ S do
                update(a, g, b, operation)
            end for
        end for
        for all a ∈ neighbors(g) do
            for all b ∈ neighbors(a) do
                update(g, a, b, operation)
            end for
            for all b ∈ neighbors(a) \ S do
                update(b, a, g, operation)
            end for
        end for
    end for
end function
```

IV. Experimental Results

In this section we measure the prediction-strength of the models, how strong the resulting Go-program plays and how fast the CFG patterns can be updated.

A. Framework

In our experiments we used a Monte Carlo Tree Search program with the RAVE-Heuristic [16] and Progressive-Bias [12]. The selection-formula for the tree-search is representative for current Go-programs and was as used by Shih-Chieh Huang in his program ERICA [6]. He did extensive experiments on playout policies which was the reason to follow his work as closely as possible so that the results can be compared. At each selection step in MCTS the move which maximizes

$$(1-\beta)\cdot Q_{\mathrm{Uct}}(s,a)+\beta\cdot Q_{\mathrm{Rave}}(s,a)+PB\frac{\pi(s,a)}{\mathrm{visits}(s,a)}$$

is chosen. The terms in the formula are defined as:

$$Q_{\mathrm{Uct}}(s,a):=\frac{\mathrm{wins}(s,a)}{\mathrm{visits}(s,a)}+C\sqrt{\frac{\ln(\mathrm{visits}(s))}{\mathrm{visits}(s,a)}}$$

$$Q_{\mathrm{Rave}}(s,a):=\frac{\mathrm{wins}_{\mathrm{rave}}(a)}{\mathrm{visits}_{\mathrm{rave}}(a)}$$

$$\beta:=\frac{\mathrm{visits}_{\mathrm{rave}}}{\mathrm{visits}+\mathrm{visits}_{\mathrm{rave}}+4\cdot B^2\cdot\mathrm{visits}\cdot\mathrm{visits}_{\mathrm{rave}}}$$

Our constants in the selection-formula were set as

$$PB = 50$$
$$C = 0.6$$
$$B = 0.1$$

In the tree search large patterns as described by Coulom [3] are used with all the tactical patterns described there but without ladder- and territory-features. In the playouts small patterns of size 3x3 with atari information, locality and tactics as described in [6] are used in a Bradley-Terry model. The moves in the playouts are chosen randomly according to the probability distribution of this model. This version is our baseline when we add CFG patterns into the playout part and in the tree search.

B. Learning

We fit each policy $\pi(s, a)$ for move prediction by maximizing the log-likelihood of a training set with positions and corresponding move decisions of experts.

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \pi(s_i, a_i)$$

For this purpose we used the game-collection from u-go.net[3]. It contains games on the 19x19 board from strong amateur players (at least one player has a rank of 4d or higher) and we filtered it for non-handicap games which have at least 300 moves. We randomly assigned each game to a training set with 20.000 games and a validation and testing set with 2000 games each.

We used the Minorization-Maximization algorithm as proposed by R. Coulom in [3] to maximize the log-likelihood. In his paper batch learning was used but as our training set is large we adopted the algorithm to the Stochastic-Majorization-Minimization[4] framework of [17].

Similar to [3] we added two virtual games, i.e. one win and one loss, against an opponent of strength $\gamma = 1$ as a prior for each weight to regularize the model and avoid over-fitting. Additionally, only features that occur more than 10 times in moves chosen by experts are learned.

C. Prediction Strength

We compared the prediction strength of the trained Bradley-Terry models on the test set of 2000 games on the 19x19 board.

In Figure 6 the cumulative distribution function of the probability that the expert move is within the top n ranked moves is shown. For comparison we included the rankings of the tree-search- and the playout-patterns. We have comparable results to what is published in literature (e.g. [3], [4]), but achieved a slightly better ranking for the tree-search patterns. This is due to the fact that our training set is larger than those used in the papers.

[3]http://u-go.net/gamerecords-4d/

[4]Minorization-Maximization and Majorization-Minimization refer to the same principle depending on whether one wants to minimize or maximize

You can see that the CFG patterns increase the ranking quality of both the tree-search as well as the playout-patterns. The CFG patterns increase the prediction accuracy of the tree-search-patterns to 42.1% which is even larger than the results published in [18] who use latent factors to model interactions between features and whose results are among the best published so far.

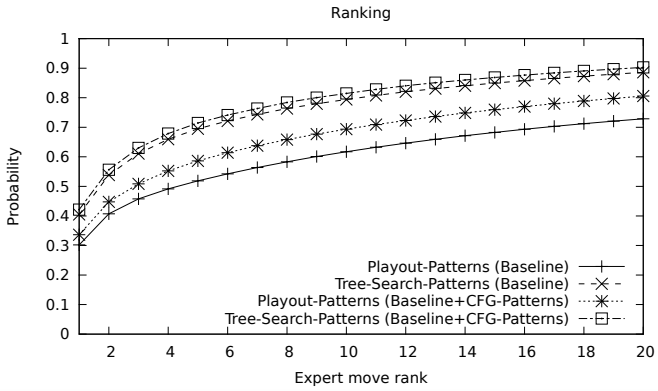


Fig. 6. Ranking: probability that the expert move is within the top n ranked moves of the model

In Figure 7 the log-likelihood of selecting the expert move is plotted. The log-likelihood is given for 12 game-phases each consisting of 30 moves. In contrast to the previous ranking plot this gives a better discrimination of the strength of each prediction model. You can see that the playout-patterns have low predictive strength in the beginning of a game and get better at the end. In contrast, the big patterns in the tree-search-model are very good in the opening as they can learn opening lines by rote but decline towards the end of the game. The CFG patterns are similar to the playout patterns as they are unable to learn opening sequences but get similar prediction strength than the big patterns towards the end of the game.

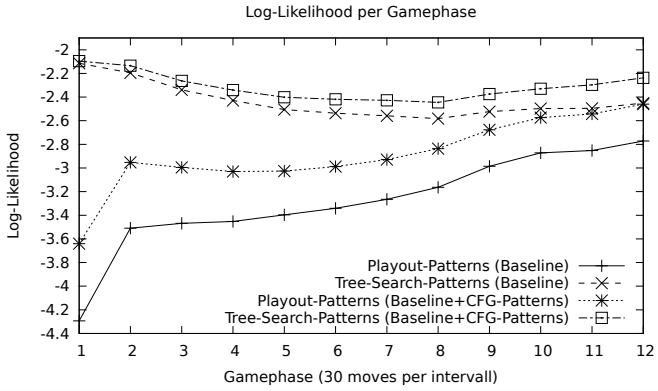


Fig. 7. Mean log likelihood of selecting the expert move per game-phase

D. Playing Strength

We ran 1024 games against GNUGo 3.8. [19] with Level 0 on the 19x19 board to measure the winning rate (table I). We use as a baseline a version with large patterns in the tree-search and small patterns and some local tactics in the playouts as described in section IV-A. This is typical for current computer

Go programs. Then we added a version with CFG patterns in the playouts and one with CFG patterns in the playout and tree-search. All patterns and local tactics of the baseline version are still included in these versions. Each configuration runs 1000 playouts/move. The settings and the baseline-version are similar to what is published about the go-program ERICA [6] so that the results can be compared.

TABLE I. WINNING-RATE AGAINST GNUGO 3.8. LEVEL 0 WITH 1024 GAMES, 95 % CONFIDENCE INTERVALS

Type of program	Winning Rate	ΔELO vs. GNUGo
Baseline	46.6% ± 3.1%	-23.5 [-45, -2]
+ CFG Patterns (Playout) *	53.9% ± 3.1%	+27 [+5, +48]
+ CFG Patterns (Tree-Search)	65.4% ± 2.9%	+111 [+89, +134]
+ CFG Patterns (Playout & Tree-Search) *	72.5% ± 2.7%	+168 [+144,+192]

Programs which cannot be compared on a time-basis are marked with a star (when using CFG patterns in the playouts the speed for playouts is considerable slower than the baseline, see IV-E). The version with CFG patterns in the playouts is about 50 ELO, in the tree-search 135 ELO and in both about 190 ELO stronger than the baseline. It seems that the CFG patterns in the tree-search part are of much more value than in the playout part, despite the fact that the CFG patterns increase the ranking quality of the playouts more than in the tree-search.

E. Incremental Updates

To show the efficiency of the incremental updates we measured the average number of path-updates per move for 1000 playouts on an empty 19x19 board. The results are shown in table II. You can see that the number of updates per move is very low especially at the beginning of a game. In later game-phases the number of updates gets larger because more stones are on the board which are more likely connected resulting in an increasing number of neighbors of each vertex. Nevertheless, the work is still only a fraction of what had to be done by a non-incremental algorithm.

TABLE II. PATH-UPDATES PER MOVE

Game-Stage (Move)	Path-Removals/Move	Path-Additions/Move
0-100	27.5	24.7
100-200	26.6	24.2
200-300	78.4	77.3
300-400	80.2	79.9

We measured the number of playouts per second on an Intel Xeon E5-1620 CPU with 3.6 GHz on a single core. On an empty board we get 1797 ± 18 playouts/s for the baseline playouts and 196 ± 2 playouts/s with CFG patterns, which is considerably slower.

A possible way to remain competitive with these slower playouts is to use the CFG patterns only in the tree-search which does not slow down the program and still achieves an increase of playing strength of about 135 ELO. The results for the CFG patterns in the playouts are interesting as they improve the playing strength of the MCTS. This is a non trivial result as stronger policies are not guaranteed to improve the performance of MCTS [20]. Therefore, CFG patterns in the playouts might be useful as part of a cluster version (as we e.g. have in our Go program gomorra [21]) where the raw speed of the playouts is harmed by the communication overhead anyway.

V. CONCLUSIONS AND FURTHER WORK

The common fate graph allows us to represent a Go board in a very natural form leading to good prediction algorithms. Unfortunately, most prediction models on graphs have high costs.

In this paper we showed how to incorporate the neighborhood hash of a linear time graph kernel to construct explicit features. This allowed us to use patterns of the common fate graph in the Bradley-Terry model to predict expert moves and use it in the tree search as well as in the playout part of a Go program. Updating the representation in an incremental way allowed us to maintain good performance. The prediction model got better prediction rates than typical predictors used in Computer Go, which are based on shape patterns. We conducted experiments by playing against another Go program and achieved an increase in playing strength of about 190 ELO.

Future work includes to investigate ways to do a better neighborhood hashing. One possible way is to do more iterations to include more information on a vertex's neighborhood. This information could prove viable to get better prediction rates in the opening phase of the game.

Furthermore, we expect to increase the number of playouts per second by improving the code used to calculate the CFG patterns. While the incremental updating procedure helped to reduce the amount of work per move the code itself is still unoptimized. This could help to lessen the performance gap between current playouts and those using CFG patterns in the future.

The most interesting aspect of CFG patterns is their capability of representing tactical knowledge in the playouts. We currently work on adapting the playouts based on the knowledge obtained in the tree-search. This is limited by the representation-capability of the features used in the playouts. By using CFG patterns we hope to be able to develop playouts which can adapt to local tactics.

REFERENCES

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, March 2012.

[2] D. Stern, R. Herbrich, and T. Graepel, "Bayesian Pattern Ranking for Move Prediction in the Game of Go," in *Proceedings of the 23rd International Conference on Machine Learning*, 2006, pp. 873–880.

[3] R. Coulom, "Computing Elo Ratings of Move Patterns in the Game of Go." *ICGA Journal*, vol. 30, no. 4, pp. 198–208, 2007.

[4] M. Wistuba, L. Schaefers, and M. Platzner, "Comparison of Bayesian move prediction systems for Computer Go," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, Sept 2012, pp. 91–99.

[5] Y. Wang and S. Gelly, "Modifications of uct and sequence-like simulations for monte-carlo go," in *IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007.*, April 2007, pp. 175–182.

[6] S.-C. Huang, "New Heuristics for Monte Carlo Tree Search Applied to the Game of Go," Ph.D. dissertation, 2011.

[7] T. Graepel, M. Goutrié, M. Krüger, and R. Herbrich, "Learning on Graphs in the Game of Go," in *Proceedings of the International Conference on Artificial Neural Networks*, ser. ICANN '01. London, UK: Springer-Verlag, 2001, pp. 347–352.

[8] J. Foulds, "Learning to Play the Game of Go," Honours Thesis, University of Waikato, 2006.

[9] L. Ralaivola, L. Wu, and P. Baldi, "SVM and Pattern-Enriched Common Fate Graphs for the game of Go," *ESANN*, vol. 2005, pp. 27–29, 2005.

[10] E. H. Nijhuis, "Learning Patterns in the Game of Go," Master's thesis, University of Amsterdam, 2006.

[11] K. Ikeda and S. Viennot, "Efficiency of Static Knowledge Bias in Monte-Carlo Tree Search," in *Computers and Games 2013*, 2013.

[12] G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy, "Progressive strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[13] S.-C. Huang, R. Coulom, and S.-S. Lin, "Monte-Carlo Simulation Balancing in Practice," in *Proceedings of the 7th International Conference on Computers and Games*, ser. CG'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 81–92.

[14] T.-K. Huang, R. C. Weng, and C.-J. Lin, "Generalized Bradley-Terry Models and Multi-Class Probability Estimates," *J. Mach. Learn. Res.*, vol. 7, pp. 85–115, Dec. 2006.

[15] S. Hido and H. Kashima, "A Linear-Time Graph Kernel," in *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, Dec 2009, pp. 179–188.

[16] D. Silver, "Reinforcement Learning and Simulation-based Search in Computer Go," Ph.D. dissertation, 2009.

[17] J. Mairal, "Stochastic Majorization-Minimization Algorithms for Large-Scale Optimization," in *Advances in Neural Information Processing Systems 26*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., 2013, pp. 2283–2291.

[18] M. Wistuba and L. Schmidt-Thieme, "Move Prediction in Go Modelling Feature Interactions Using Latent Factors," in *KI 2013: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, I. Timm and M. Thimm, Eds. Springer Berlin Heidelberg, 2013, vol. 8077, pp. 260–271.

[19] "GNU Go," Website, available online at http://www.gnu.org/software/gnugo/; visited on 31.03.2014.

[20] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *International Conference of Machine Learning*, Corvallis, United States, Jun. 2007.

[21] T. Graf, U. Lorenz, M. Platzner, and L. Schaefers, "Parallel Monte-Carlo Tree Search for HPC Systems," in *Proceedings of the 17th International Conference, Euro-Par, volume 6853 of LNCS*. Springer, 2011, pp. 365–376.