Decision Trees for Computer Go Features

Francois van Niekerk and Steve Kroon

Computer Science Division, Stellenbosch University, Stellenbosch, South Africa
francoisvn@ml.sun.ac.za, kroon@sun.ac.za

Abstract. Monte-Carlo Tree Search (MCTS) is currently the dominant algorithm in Computer Go. MCTS is an asymmetric tree search technique employing stochastic simulations to evaluate leaves and guide the search. Using features to further guide MCTS is a powerful approach to improving performance. In Computer Go, these features are typically comprised of a number of hand-crafted heuristics and a collection of patterns, with weights for these features usually trained using data from high-level Go games. This paper investigates the feasibility of using decision trees to generate features for Computer Go. Our experiments show that while this approach exhibits potential, our initial prototype is not as powerful as using traditional pattern features.

1 Introduction

In Computer Go, Monte-Carlo Tree Search (MCTS) is currently the dominant algorithm [1, 2]. While the standard MCTS algorithm requires limited domain knowledge for a moderate level of strength [1], it has been shown that the inclusion of more domain knowledge can greatly increase the playing strength of Computer Go engines using MCTS [1, 3, 4]. One successful approach to incorporating such domain knowledge is using features [5]. This paper reports on a prototype implementation using decision trees as MCTS features in order to extract domain knowledge for Go.

After giving some background in Section 2, Section 3 describes our proposed method of using decision trees as features. Section 4 presents experimental results for the proposed approach.

2 Background

2.1 The Game of Go

Go is a combinatorial game played on a board consisting of a rectangular grid of intersections (a 19x19 grid is the most popular board size) [6]. Two players, black and white, alternate placing stones of their respective color on empty board intersections. Orthogonally contiguous stones of the same color form chains. If a chain of stones has zero adjacent empty intersections, also known as liberties, then the entire chain is removed from the board. The game ends after two successive passes — the winner is the player controlling the largest portion of the board.

2.2 Go Features for Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is the current dominant algorithm in Computer Go and all the top engines make use of MCTS variants [1, 2]. While hand-coded domain knowledge can quickly improve performance of MCTS engines, it is highly preferable to use automated methods of incorporating domain knowledge. One technique that successfully incorporates a large amount of domain knowledge in an automated manner is the use of Go features [5].

Go features are traditionally divided into pattern and tactical features [5]. Pattern features are simple encodings of the state of the surrounding board intersections. Tactical features encode simple domain knowledge not present in the pattern features, such as capturing a chain in atari (i.e. with only one remaining liberty). Each feature takes on one of a number of mutually exclusive levels. A comprehensive list of the tactical features used in this work is given in Table 1. Each potential move can then be described by a feature vector, with each vector component specifying which level a feature assumes for the candidate move. Patterns are typically represented by a single feature with many levels, with each level corresponding to a different pattern.

The intersections included in a pattern are typically all those within a certain distance from the center of the pattern. A popular distance measure used for large patterns in Go, and in this paper, is *circular distance* [5, 7]: $\delta x + \delta y + \max(\delta x, \delta y)$, where $\delta x (\delta y)$ is the difference between the $x(y)$-coordinates of the pattern center and another intersection.

Feature levels for patterns should be invariant to changes in rotation, reflection, and whose turn it is to play. Invariance to player turns is usually achieved by swapping stone colors as necessary, while the invariance requirements for rotation and reflection are met by considering the eight combinations of rotation and reflection and using the pattern with the lowest hash value.

In order to make practical use of features, each level of each feature is assigned a trained weight, as discussed in Section 2.3. Feature weights corresponding to the levels in the potential move's feature vector are combined to form a compound weight for the move. These move weights can then be used in the MCTS tree to order moves for exploration, and in playouts for move selection.

2.3 The Generalized Bradley-Terry Model and Training Weights

In order to train weights for each feature level, features can be modeled using the generalized Bradley-Terry model for predicting the outcome of competitions between multiple teams of individuals [5]. In this model, the skill of each individual i is represented by a positive value γ_i, with a larger γ corresponding to a more skilled individual [5]. For training feature level weights, each individual represents a feature level and a team represents the feature vector for a potential move. The following example shows how the model predicts the outcome of a competition between teams of individuals [5]:

$$P(\text{1-2-3 wins against 2-4 and 1-5-6-7}) = \frac{\gamma_1\gamma_2\gamma_3}{\gamma_1\gamma_2\gamma_3 + \gamma_2\gamma_4 + \gamma_1\gamma_5\gamma_6\gamma_7}$$

A collection of competition results harvested from game records can be analyzed using this model — the resultant optimization problem (to determine the γ values, representing weights) can be approximately solved using minorization-maximization (MM), which has been shown to have good performance [5, 8].

Alternative techniques for training weights, not considered in this work, include Loopy Bayesian Ranking, Bayesian Approximation Ranking, Laplace$_q$ Marginal Propagation, and Simulation Balancing [8, 9, 10].

2.4 Graphs for Go

While simple Go patterns are useful, an alternative representation of the Go board is the Common Fate Graph (CFG) [11]. In the CFG of a Go board, each chain of stones and each empty intersection is represented by a single graph node. This causes certain functionally equivalent patterns to become equal. Due to computational concerns, their practical use in Computer Go has been limited — one notable concept arising from this representation is the *CFG distance* [2]. The decision tree approach in this work makes use of another graph representation for Go positions.

2.5 Decision Trees

Decision trees are tree structures with queries at internal nodes and values at leaves [12]. The queries evaluate the attributes of an input data point. In order to use a decision tree, the tree is descended, with the evaluated queries at internal nodes determining the descent path. The value stored at the resultant leaf is then typically used as a predicted outcome for the input.

Decision trees can be particularly sensitive to queries near the root of the tree. Decision forests, also known as random forests, can be used to construct a more robust model: this approach uses multiple decision trees that are grown from subsets of the input data to create an ensemble of decision trees. Such an ensemble of decision trees has been shown to yield more accurate classification than a single decision tree in many cases [13].

3 Decision Trees as Features

3.1 Overview

This section presents an approach to using decision trees as features for Go. An alternative way of viewing decision trees is that they partition the input space in a hierarchical fashion, and assign a predicted value to each element of the final partition, represented by the leaves. In our method, each query in a decision tree provides additional information about the surrounding board position — each node can be thought of as representing a pattern, with leaves representing the most complex patterns.

For this paper, we utilize a graph representation of the board: a *discovered graph* of the board area surrounding a candidate move is grown during each tree descent, such that the patterns represented by this graph at decision tree nodes grow in size and specificity as the tree is descended. If the discovered graph were grown to its maximum size and detail, it would yield the graph representation of the whole board.

A decision forest, an ensemble of such decision trees, is used to improve robustness. Each decision tree in the forest is treated as a separate feature with each leaf node corresponding to a unique feature level.

Section 3.2 presents the structure of these decision trees, by specifying our stone graph board representation and the form of the queries. In Section 3.3, a method for growing and training weights for the leaf nodes of such trees is described.

3.2 Structure

Queries in the decision tree are phrased in terms of a graph representation of the Go board and various representations are possible. In this work, we elected to use the following *stone graph* to represent the board position:[1]

- There is a node corresponding to each stone on the board and each of the four board sides.
- There is an edge between every pair of nodes.
- The weight of each edge is the Manhattan distance between the two stones (or the stone and board side) represented by the edge's end nodes on the board.
- Each node has, as applicable, attributes for the status (black, white or side), size (number of stones) and number of liberties[2] of its respective chain.

An additional node, that represents the empty intersection for the potential move under consideration, is then added to the stone graph to form the *augmented stone graph* (ASG). This node has edges to every other node in the ASG — these edges have weights allocated as in the stone graph. This node is labeled as node zero and referred to as the center. At the root of the decision tree the discovered graph contains only node zero. As the tree is descended, each query encountered either adds a node from the ASG to the discovered graph (as well as its edges to nodes in the discovered graph), or refines information about the attributes of a node or the weight of an edge already in the discovered graph.

Each decision tree query has multiple possible outcomes, one per child tree node. The queries were designed to be invariant to rotation and reflection as far as possible.

We specified three possible parametrized queries for expanding decision tree nodes — parameters are shown like [**this**]:

[1] This graph representation was chosen in the view that it may improve the opening of Oakfoam, the MCTS implementation being extended [14].

[2] A variation of pseudo-liberties [15] (where chains in atari have their number of pseudo-liberties set to one) was used to simplify implementation.

NEW: Is there a new [**black? white? side?**][3] node with an edge weight to the center less than or equal to [**distance**]?
Look for a new node to add to the discovered graph from the ASG, and number the new node incrementally, if one is found. If multiple matching nodes in the ASG are found, attempt to select a unique node according to the rules found in Appendix A. Separate children are added to the decision tree for each allowed status, and none.

DIST: Is the edge weight between node [**x**] and node [**y**] [**less than**|**equal to**][4] [**val**]?
Query an edge of the discovered graph. Children are added to the decision tree for yes and no.

ATTR: Is the [**size**|**number of liberties**] of node [**x**] [**less than**|**equal to**] [**val**]?
Query a node of the discovered graph. Children are added to the decision tree for yes and no.

Figure 1 shows a portion of an example decision tree with a highlighted descent path. The leaf at the end of the descent path corresponds to a move on the fourth line with no stones within a Manhattan distance of eight. Note that node zero is the node representing the candidate move and, in this case, node one is the closest border. Also note that the distance between an intersection on the fourth line and the border is three.

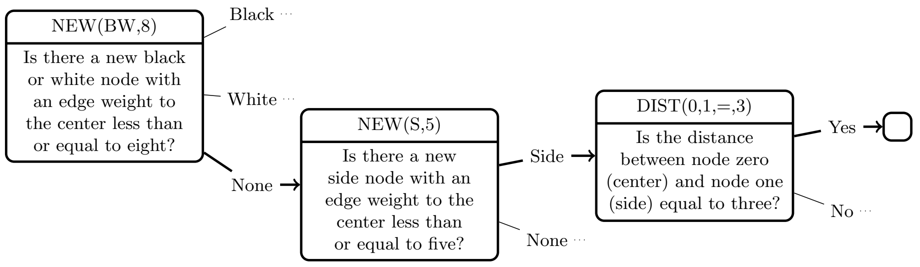
NEW(BW,8)
Is there a new black or white node with an edge weight to the center less than or equal to eight?
Black ···
White ···
None
NEW(S,5)
Is there a new side node with an edge weight to the center less than or equal to five?
Side
None ···
DIST(0,1,=,3)
Is the distance between node zero (center) and node one (side) equal to three?
Yes
No ···

Fig. 1. A portion of an example decision tree showing a descent path. The leaf at the end of the descent path corresponds to a move on the fourth line with no stones within a Manhattan distance of eight.

3.3 Learning

Traditional decision trees attempt to partition the input space such that labels for points in the input space are homogeneous within partition elements. In our case, we do not have labeled points, so conventional decision tree training techniques are not applicable. Instead, we aim to construct our decision trees such that the portion of input space corresponding to each of the final partition elements is roughly equal in size.

To achieve this, we choose queries that divide visits to children nodes roughly evenly. Statistics are gathered for candidate queries, and the query with the best

[3] This parameter can be any combination of black, white and side, such as: black or side.

[4] [**x**|**y**] means either x or y.

split quality q is chosen when a certain number of descents to the respective node have occurred and the relevant quality is above a certain threshold. To divide tree descents evenly, we defined $q = 1 - 2|0.5 - s|$, where s is the proportion of visits to the last outcome for the candidate query.[5]

Once a decision tree has been grown, weights for the leaf nodes can be trained along with any other features using MM [5]. When a decision forest is used, each tree is independently grown and then the weights of all the trees are trained together (again with any other features). MM was chosen for training because it has been shown to have good performance, and there is a freely available tool that has been used for previous work, allowing us to verify our implementation for tactical and pattern features [5, 8].[6]

4 Experiments and Results

4.1 Overview

Our decision tree features will be used for move ordering, so their performance will first be tested on a move prediction task. We will then use the best configuration for a limited playing strength comparison.

Oakfoam [14] is an open source MCTS Go engine, used for the implementation and testing of these decision tree modifications. A collection of high-level 19x19 games played on KGS from 2001 to 2009 was used for training and testing. This data set is available from [16].

The following approach was used for extracting features and training their weights:

1. If enabled, harvest popular patterns from the collection. The number of games used is varied to adjust the number of patterns. Patterns with intersections within circular distances of 3–15 from their center, which occur at least 20 times in the considered games, are harvested.
2. If enabled, grow a decision forest by collecting statistics from games in the collection. A query is added to a node after at least 1000 descents to the node have occurred, and the quality is sufficient ($q \geq 0.4$ was chosen). The number of games used is varied to adjust the size of the trees.
3. Train weights for all the features using MM.

Only 10% of the games' moves were used in order to sample from a large number of games. For decision forests, the 10% of moves were independently sampled for each tree. Once the weights were trained, the appropriate test (move prediction or strength comparison) was performed. More details on these tests follow in Sections 4.2 and 4.3.

The sections that follow, the notation of x/y for decision forests is used to signify a decision forest comprising y trees with x leaves in total.

[5] This is designed to deal with NEW queries that have more than two children.
[6] However, we later discovered that the MM tool was unable to deal with large training data sets for our tests.

Table 1 lists the tactical features used in this work. The table also includes example weights for two configurations: one using tactical and pattern features, and another using tactical and decision tree features.

4.2 Move Prediction

For move prediction, features were used to compute move weights for all legal moves in a Go board position, and an ordering of the moves according to these weights was formed. The rank of the actual move played in this ordering was then used as a measure of move prediction accuracy. This process was repeated for every position in a collection of games. Each point on the resultant move prediction graphs show the proportion of positions where the actual move was within the top x ranked moves.

These move prediction tests were each performed on 100 19x19 games that are disjoint from the training data set. The 95% confidence interval width, w.r.t. different testing data, for a single data point assuming 100 moves in each of the 100 games (typical for 19x19 games) is smaller than 0.02. These confidence intervals are therefore not shown on the graphs. Slight changes in the curves are observed for different training data sets, but more time would be needed to quantify this variance.

In the legends of the graphs that follow, tactical, pattern and decision tree features are indicated with T, P and DT respectively. The number of games used for training weights is indicated in square brackets; e.g.: "T + DT(10000/1) [1000]" indicates that the configuration used tactical and decision tree features, there was one decision tree with 10000 leaves, and that the weights were trained with 1000 games.

We first evaluated the effect of increasing the number of decision tree leaves, while keeping the number of trees fixed. This was done for a single decision tree and a decision forest with eight trees. The results are shown in Figure 2. We found that move prediction accuracy improved as the number of leaves increased, but only up to a certain point.

We then investigated the impact of increasing the number of trees in the decision forest for a roughly fixed total number of leaves. We began with the strongest configuration from the previous series of tests: T + DT(17761/8) [4000]. The results are shown in Figure 3. We found that increasing the decision forest from one to eight trees improved the move prediction accuracy. We also found that increasing the number of trees in the decision forest from eight to sixteen trees decreased the number of games we could use for training to 2000. This increase in decision forest size did not yield an improvement for move prediction, but it did require more processing time, so the previous configuration was kept for use in the next step.

Finally, we compared various combinations of tactical, pattern and decision tree features. The results are shown in Figure 4. We found that tactical and decision tree features did not perform as well as tactical and pattern features, but that they showed a substantial improvement over tactical features alone. We

Feature	Level	γ^P	γ^{DT}	Description
Pass	1	7.68	1.00	Pass after a normal move
	2	408.71	54.13	Pass after another pass
Capture	1	0.40	1.00	Capture a chain
	2	37.51	3.93	Capture a chain in a ladder
	3	2.21	8.71	Capture, preventing an extension
	4	3.04	11.80	Re-capture the last move
	5	14.59	42.03	Capture a chain adjacent to a chain in atari
	6	33.74	17.12	Capture a chain as above of 10 or more stones
Extension	1	5.06	10.07	Extend a chain in atari
	2	0.63	1.13	Extend a chain in a ladder
Self-atari	1	0.59	0.44	Self-atari of 5 or fewer stones
	2	0.21	0.16	Self-atari of more than 5 stones
Atari	1	1.92	1.84	Atari a chain
	2	0.84	0.60	Atari a chain and there is a ko
	3	2.03	2.19	Atari a chain in a ladder
Distance to border	1	0.43	0.84	
	2	1.12	1.16	
	3	1.51	1.16	
	4	1.20	1.13	
Circular distance to last move	2	9.18	13.36	
	3	6.44	7.75	
	4	3.57	4.37	
	5	3.27	3.79	
	...	...	...	
	10	1.53	1.68	
Circular distance to second-last move	2	1.41	1.67	
	3	1.54	1.77	
	4	1.23	1.14	
	...	...	...	
	10	1.08	1.10	
CFG distance to last move	1	2.98	2.63	
	2	3.38	2.96	
	3	3.42	3.26	
	4	2.13	1.97	
	...	...	...	
	10	1.06	1.02	
CFG distance to second-last move	1	4.33	3.00	
	2	3.39	2.51	
	3	2.50	1.70	
	4	2.08	1.66	
	...	...	...	
	10	1.16	1.08	

Table 1. List of tactical features, with example weights from two configurations: γ^P for tactical and pattern features with 83644 patterns, and γ^{DT} for tactical and decision tree features with a 17761/8 decision forest. The weights for level zero of each feature are fixed at 1.0. When multiple feature levels are applicable, the highest level is selected.

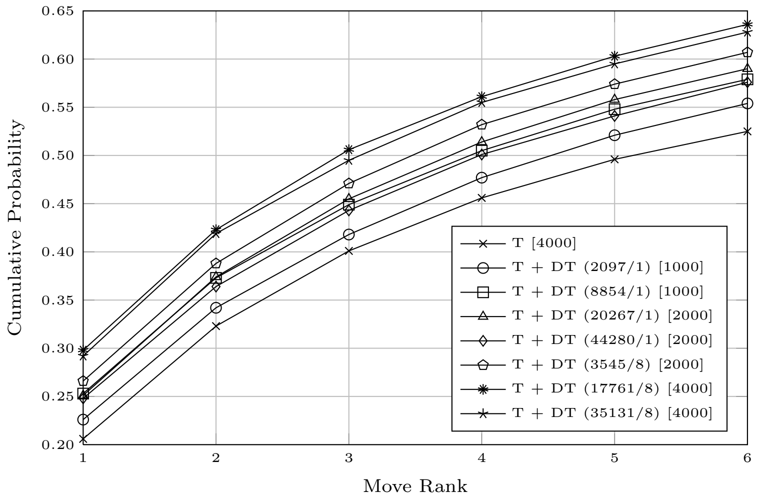

Fig. 2. Move prediction of tactical and decision tree features with different numbers of decision tree leaves.

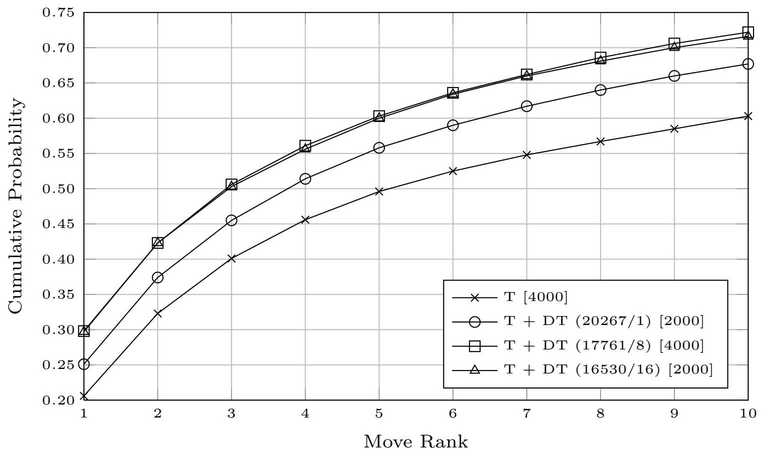

Fig. 3. Move prediction of tactical and decision tree features with different decision forest sizes.

also found that the inclusion of decision trees along with tactical and pattern features made no significant difference to move prediction accuracy.

4.3 Playing Strength

We used move prediction performance to select configurations for strength comparison tests — the configurations used in Figure 4 were compared in terms of playing strength by playing a series of games against GNU Go [17]. All games were played on 19x19 against GNU Go (version 3.8, level 10) with 7.5 komi and alternating colors starting on consecutive games. Playing strength was compared

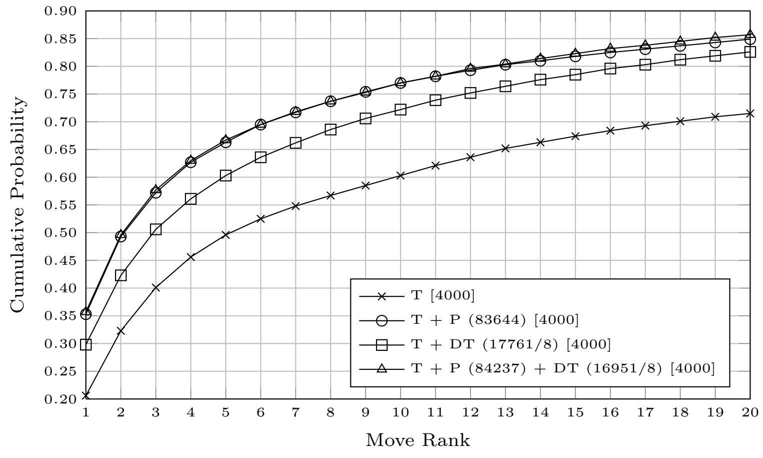

Fig. 4. Move prediction of the best configurations of each combination of pattern and/or decision tree features.

using 10000 playouts per move. This restriction was used because the aim was to investigate the feasibility of the decision tree features, and the prototype decision tree implementation was not optimized. For these tests, features were used for move selection during progressive widening in the MCTS tree [1, 5]. Results of the strength comparison are shown in Table 2.

Tactical	Pattern	Decision forest	MP Accuracy	Games	Winrate
X	-	-	20.6%	200	5.0%
X	83644	-	35.3%	300	49.0%
X	-	17761/8	29.8%	200	30.0%
X	84237	16951/8	35.7%	200	50.5%

Table 2. Comparison of playing strength of 10000 playouts per move vs GNU Go with various configurations. MP accuracy proportion of moves ranked best by the features used.

We found that the inclusion of decision tree features with tactical features resulted in a large increase in playing strength. We also found that the inclusion of decision tree features with tactical and pattern features did not result in a significant change to playing strength. These results are as expected from the move prediction results in Section 4.2.

5 Conclusions and Future Work

We have presented an approach using decision trees as features for extracting domain knowledge from game records for Computer Go. Our approach employs queries that refine knowledge of the current board position as the tree is descended. Our prototype implementation showed reasonable results in terms of move prediction and playing strength, although it did not perform as well as traditional pattern features. However, we believe there is significant potential for our method due to the general applicability of our method: many other board representations, query structures, and query selection criteria can be considered, and the general approach should be easily transferable to other domains.

One can specify tactical and pattern features as decision trees — from this perspective, our approach benefits from the structure of the tree being learned from training data, and not just the weights.

Our current work was not able to investigate larger decision trees, since the MM tool employed was not able to handle sufficient training data for these situations. To address this, we intend to explore other methods for training weights, such as Laplace$_q$ Marginal Propagation [9].

6 Acknowledgments

The first author would like to thank the MIH Media Lab at Stellenbosch University for the use of their facilities and support of the work presented here.

References

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. 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. 1–49, 2012.

[2] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, "Current Frontiers in Computer Go," *IEEE Symposium on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 229–238, 2010.

[3] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *24th International Conference on Machine Learning*, pp. 273–280, ACM Press, 2007.

[4] G. M. J.-B. Chaslot, L. Chatriot, C. Fiter, S. Gelly, J. Perez, A. Rimmel, and O. Teytaud, "Combining expert, offline, transient and online knowledge in Monte-Carlo exploration," *IEEE Transactions on Computational Intelligence and AI in Games*, 2008.

[5] R. Coulom, "Computing Elo Ratings of Move Patterns in the Game of Go," *ICGA Journal*, vol. 30, 2007.

[6] K. Baker, *The Way to Go.* American Go Foundation, 1986.

[7] F. de Groot, "Moyo Go Studio." `http://www.moyogo.com`, 2004.

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

[9] L. Lew, *Modeling Go Game as a Large Decomposable Decision Process.* PhD thesis, Warsaw University, 2011.

[10] S.-C. Huang, R. Coulom, and S. Lin, "Monte-Carlo Simulation Balancing in Practice," *Computers and Games*, pp. 81–92, 2011.

[11] L. Ralaivola, L. Wu, and P. Baldi, "SVM and Pattern-Enriched Common Fate Graphs for the Game of Go," in *European Symposium on Artificial Neural Networks*, pp. 485–490, 2005.

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Pearson, third ed., 2010.
[13] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[14] "Oakfoam." `http://oakfoam.com`.
[15] J. House, "Groups, liberties, and such." `http://go.computer.free.fr/go-computer/msg08075.html`, 2005.
[16] "Game records in SGF format." `http://www.u-go.net/gamerecords/`.
[17] "GNU Go." `http://www.gnu.org/software/gnugo/`.

A Multiple Decision Tree Descent Paths

It is possible that a NEW decision tree query may not be able to identify a unique node from the ASG to add to the discovered graph. In this situation, a sequence of conditions are considered, in an attempt to enforce uniqueness. Each condition will select the node(s) that best satisfy the condition and eliminate the others. These conditions are designed to enforce invariance to changes in rotation and reflection as far as possible. If the conditions are not able to identify a unique node, then each of the possibilities is considered.[7] The sequence of conditions used in this work is as follows:

- Select node(s) closest to the candidate move.
- Select black over white over side nodes.
- Select node(s) closest to nodes already in the discovered graph, in reverse order of discovery.
- Select node(s) with the most stones in its respective chain.
- Select node(s) with the most liberties around its respective chain.

Even though these conditions are not always able to find a unique node, empirical results showed that a single leaf node is reached in about 85% of tree descents. It was therefore decided to only return one of these nodes, namely the left-most node in the tree. Investigation showed that this option made negligible difference to move prediction accuracy, while providing a large reduction in training time and an increase in the size of training data set that could be handled. This is due to the leaf nodes of each decision tree becoming mutually exclusive, allowing decision trees to be treated as single features.

B Reproducibility

All source code used in this work is available in the codebase of Oakfoam, an open-source MCTS-based Computer Go player [14]. Version 0.1.3 was used for the work in this paper and is tagged in the code repository. Default parameters were used unless specified otherwise.

The MM tool of Rémi Coulom was used to train feature weights. This tool is available at: `http://remi.coulom.free.fr/Amsterdam2007/`.

[7] Note that this violates the conceptual view that the decision tree partitions the input space, since one position may ultimately correspond to multiple leaf nodes.