UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EXPERT MOVE PREDICTION FOR COMPUTER GO USING SPATIAL PROBABILITY TREES

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

ZACK TIDWELL
Norman, Oklahoma
2012

EXPERT MOVE PREDICTION FOR COMPUTER GO USING SPATIAL PROBABILITY TREES

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Acknowledgments

Back in my early days at the university, when I was still a math major, I thought that research took place alone, in chalkboard-covered attics from which lone men emerged with results after months of scribbling proofs on the walls. It turns out that this is not the case, and this research project would not have succeeded without the help and support of a number of people.

In particular, I would like to thank my adviser, Dr. Amy McGovern, whose support I probably don't deserve.

I would also like to thank Rachel Shadoan, who dragged me through the writing process kicking and screaming, and who kept me well-fed on cruciferous vegetables even while her adviser menaced her for not working enough.

Acknowledge Hellman
Scott, my SPT buddy
thanks for the help, dude

No acknowledgements section would be complete without mention of my compatriots in the IDEA lab, much the way no week is complete without a lab meeting. Their clever puns and good humor provided a much needed break in the thesising process, and I appreciate that they only harass me a little bit for being late to every meeting.

I would like to thank Chris Rosin and the folks at Adventium, who collaborated with us on this project.

This work was supported in part by the DARPA GALE project, Contract No. HR0011-08-C-0110.

Table of Contents

List of Tables

List of Figures

Abstract

We introduce a method of learning spatial probability distributions for discrete gridded domains. We develop a decision tree based approach, which we call Spatial Probability Trees (SPTs). SPTs split the data using statistical questions about local regions in the domain. The leaves of each tree are local probability density functions. By aggregating these regional probabilities across the entire space, SPTs can be used to represent global probability density functions. We also introduce an ensemble of SPTs called a SPT forest, which significantly improves performance.

We apply SPTs to computer Go to predict expert moves. The learned SPTs can then be used to provide advice to Monte Carlo based computer Go players. We specifically demonstrate that SPTs and SPT forests can be used to significantly improve the playing strength of a state-of-the-art computer Go player that uses Upper Confidence Bounds for Trees.

Chapter 1

Introduction

Developing an agent to play the board game Go is one of the most difficult problems currently available in gaming (Müller 2002; Hsu 2007). While super-computers have been beating chess masters since the mid-90s, no computer agents have been produced that play Go at the skill level of top expert human players.

The state of the art computer Go players use Monte Carlo-based search, which has produced the best existing computer Go players. Much of the literature has shifted away from new algorithms to play Go towards providing information that improves Monte Carlo search. One approach to this is predicting the best next move based on what an expert player would play. Existing research in expert move prediction often claims improving Monte Carlo search as a motivation, but few algorithms have been demonstrated to improve the search. Furthermore, it is unclear how effective existing expert move prediction algorithms would be at evaluating thousands of board positions per second, as is required by Monte Carlo search, as many expert move prediction algorithms search very large databases to evaluate a move. Therein lies the motivation for this work: to develop an expert move prediction algorithm sufficiently simple and yet fast enough to be practically applicable to improve Monte Carlo search.

In the next chapter, we discuss the rules of go. In chapter 3, we discuss in greater detail the previous approaches to computer Go players and expert move prediction. Then, in chapter 4, we introduce a novel spatial prediction method, the Spatial Probability Tree (SPT). In chapter 5, we demonstrate that SPTs enable a top computer Go player to significantly improve its playing strength and that they are powerful predictors of expert moves. Finally, in chapter 6 we discuss our conclusions and possible future work.

Chapter 2

Rules of Go

Go is a two-player board game that is typically played on a 19 x 19 board covered with a grid. Players take turns placing stones on the board at the intersection of gridlines, with one player placing black stones and the other white stones. Go differs from games such as chess in that, other than color, all stones are functionally identical. The goal of Go is to control the most territory at the end of the game, which one accomplishes by surrounding the territory with stones of one's color. However, the balance of territory controlled can change many times throughout the game, and small, local battles can take many turns to play out. In part due to the above reasons, static evaluation in Go is quite difficult.

The rules for Go are simple, but the resulting game is incredibly complex. The 19×19 board means that there are 361 points on the board. Each point on the board can be empty, have a black stone, or have a white stone. This yields 3^{361} ($\approx 10^{172}$) possible board positions (Hsu 2007), of which approximately 1.2% ($\approx 10^{170}$) are legal (Müller 2002). Then, there are numerous important considerations beyond the legality of a move, regarding the location of a stone relative to other stones in play.

The first important definition is a group. A group is a collection of one or more stones of the same color that are connected on the board by the lines of the grid. The stones must be horizontally or vertically adjacent to each other–stones placed diagonally across a grid square from each other are not considered adjacent and may not be part of the same group. Stones in a group all share a common fate. Two useful special conditions of a group are *alive*, meaning those stones can never be captured, and *dead*, meaning that nothing can be done to prevent that group of stones from being captured.

A group is *captured* and removed from the board when it has no *liberties*. A *liberty* is an empty point adjacent to a group. (Adjacency for liberties follows the same rules as adjacency for stones: an empty point diagonally across a grid square from a group is not considered adjacent to that group.) Figure 2.1 shows examples of black groups with various numbers of liberties.

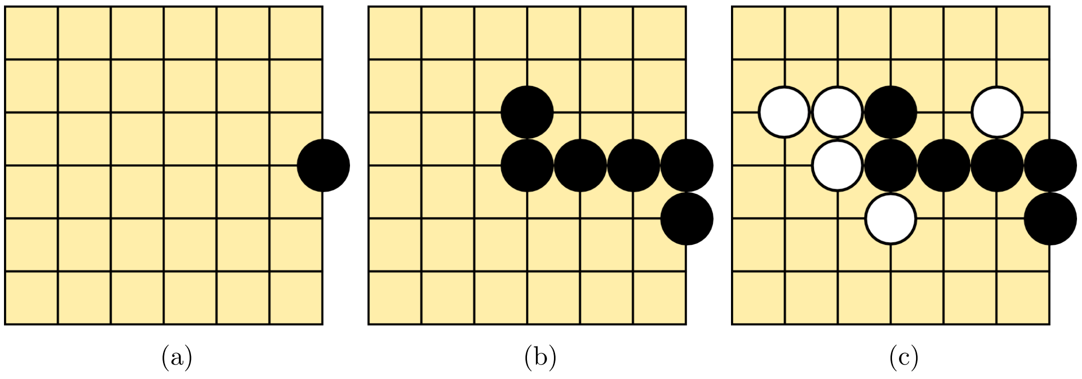

Figure 2.1: Example Go board positions with different numbers of liberties. In (a), the black group has 3 liberties. In (b), the black group has 10 liberties. In (c), the black group has 6 liberties.

Liberties that are completely surrounded by the stones of a group are called *eyes*, and they are important in determining whether a group is dead or alive. Figure 2.2 (a) shows a group of black stones with a single eye; Figure 2.2 (b) shows a group of black stones with two eyes surrounded by white stones.

Generally, it is not legal to play a stone in such a way that it would have zero liberties, as it would be immediately captured. Thus, it is generally illegal to play inside eyes, as the stone played in the eye would have no liberties. There is one exception to this rule: if a group of stones has an eye as its *only* liberty, an opponent can play a stone in the eye to capture the group.

If a group has two or more eyes in it, then it is impossible for that group to be captured. We refer to groups that cannot be captured as *alive.*

In capturing, the only things that matter are the number of liberties a group has. Because liberties are empty points in which a stone can be played, stones or groups along the edge or corner have fewer liberties than stones in the middle of the board, thus making them easier to capture. To conceptualize this, it is often convenient to imagine that the board is surrounded by an additional border of edge stones that can never be captured.

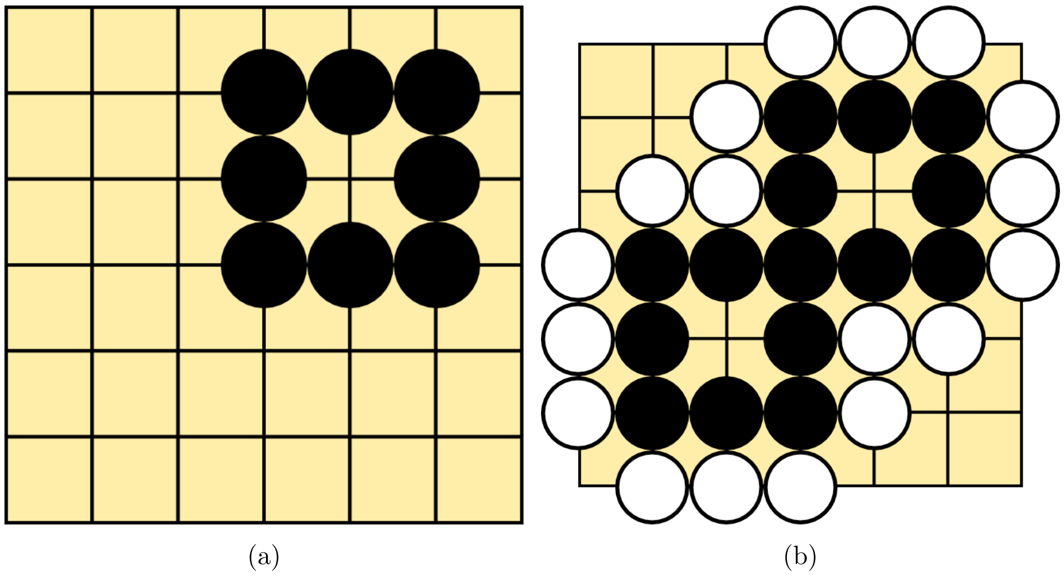

Figure 2.2: Examples of groups with *eyes*. (a) shows a group of black stones with a single eye; (b) shows a group of black stones with two eyes surrounded by white stones.

Chapter 3

Related Work

The complexity of Go results in a difficult and interesting problem for learning and search. However, Go's state space is prohibitively large for the application of traditional search methods. In order to apply such methods, it is necessary to reduce Go's state space. TsumeGo is a good example of this reduction in search space. TsumeGo is a puzzle variation of Go in which the player is given a board position and instructions for what results the next move should have. In this case, however, the puzzles are solved–there is a correct solution that can be found. Wolf (2000) demonstrated that TsumeGo solutions can be efficiently found using forward-pruning on search. Dabney and McGovern (2007) achieved 65% accuracy on TsumeGo using Relational UTree. While TsumeGo is an interesting sub-problem in solving Go, the full problem of Go cannot be directly solved through search. As a result, approaches other than traditional search must be used. The approaches we will discuss here fall into two categories: Monte Carlo methods and expert move prediction.

Most of the work on Monte Carlo methods in Go use Upper Confidence-Bound Trees (UCT), introduced by Kocsis and Szepesvári (2006). Prior to UCT, state of the art methods used other Monte Carlo methods (Cazenave and Helmstetter 2005). They apply the UCB1 N-Armed Bandit algorithm to Monte-Carlo rollout-based planning, which enables them to develop confidence bounds on each move (Auer et al. 2002). Monte Carlo planning methods take advantage of the fact that while the state space is too large to look at every action to determine which is best over the course of the game, the structure of the game provides a generative model. That is, given a state and an action, future states and rewards can be simulated. The near-optimal action can be found, then, by "sampling the future." In naïve Monte Carlo search, a set of actions is uniformly selected from the actions available at a particular state. Then, each of those actions is "rolled out" by simulating the future states and rewards each action would give. The best action can be selected based on any of a number of statistics, but usually the best average reward is chosen.

UCT improves upon this by selectively sampling from the actions available at a state. The goal is to reduce the number of actions that must be rolled out at each state by removing actions that are clearly sub-optimal. The difficulty in removing actions that appear suboptimal is that the sub-optimality of an action may evolve rapidly over the course of a game, and the estimation of optimality must be updated accordingly. It is therefore necessary to balance the exploration of sub-optimal actions with the exploitation of optimal-appearing actions, a familiar problem in machine learning.

Currently, virtually all state-of-the-art Go playing programs use UCT as a major component of their implementations (Rosin 2010a; Lee et al. 2009; Enzenberger et al. 2010). The focus of much of the literature has shifted to finding ways to improve UCT. One area of improvement is the estimation of the optimality of an action.

Expert move prediction is motivated by the potential to improve estimations of optimality in UCT (Werf et al. 2003; Stern et al. 2006; Araki et al. 2007; Coulom 2007; Sutskever and Nair 2008; Gelly and Silver 2008). Expert move prediction tries to supply the best next move given a particular board position, where an expert's move in the same case is used as the correct answer. Essentially, predicting an expert's next move is used as a proxy for finding the best next move: the expert's move is assumed to be the best.

Expert move prediction approaches are data-driven methods that develop models by training on large numbers of games or board positions. There are several databases of expert best moves on which to train a model, all of which are essentially recordings of games. There are two primary classes of datasets, distinguished by the level of the human players whose moves are recorded and the availability of the data. The GoGod dataset is a collection of games by professional players and requires a fee for its use. Online datasets, like the KGS Go server, contain a large number of freely available games played by highly-ranked amateur players (Shubert 2006). While it is presumed that the professional moves are of higher quality than moves made by amateurs, the amateur moves are still of sufficiently high quality to be considered expert level moves.

Expert move prediction methods work by attempting to determine the relationship between local features and the best next move. The restriction to local features is important: because the state space is so large, searching for the global state is infeasible. For most game databases, there will be many possible board configurations for which there are very few or no corresponding best next moves.

Most of the literature in expert move prediction features the application of existing machine learning algorithms, often with small modifications to make them more suitable to Go. Convolutional neural nets (Sutskever and Nair 2008), Bayesian nets (Stern et al. 2006; Michalowski et al. 2011), and the Maximum Entropy Method (Araki et al. 2007) have all been applied to this problem.

The accuracy of these methods, measured as the percentage of time that the algorithm's predicted next move is the same as the expert's move that is being replayed, is generally between 15 and 40 percent. This is because there may be many moves that are good for a given board position. In fact, if a particular board were shown to a number of experts, they may come to many different conclusions about what the best next move is. Because of this plurality of good next moves, it is helpful to consider other goodness measures. An accepted approach is to have the algorithm rank all of the legal moves on the board to see where the expert's move falls within that range. Calculating the percentage of time in which the expert's move falls within the top 10 or top 20 moves provides an indicator of goodness. For current state of the art algorithms, the expert's move falls within the moves that the algorithm has ranked as the top 20 moves between 75%-85% of the time (Araki et al. 2007).

The algorithm presented in this work is expert move prediction using KGS data.

Chapter 4

Methodology

In this chapter, we introduce a novel spatial prediction algorithm, the Spatial Probability Tree (SPT). The goal for the SPT is to provide an expert move prediction algorithm that evaluates board positions fast enough to be practically applicable to improving Monte Carlo based search. The specifics of the SPT algorithm were motivated by the observation that, in Go, the shape (patterns) of stones in a small area of the board can be very important in that local region of the board. We developed a model that explicitly works with the idea that local features can provide predictive information, not just for the cell where the pattern is centered, but for a small spatial area around the pattern. This local information is not necessarily useful by itself. However, when taken in aggregate over an entire space, meaningful probabilities for an event occurring in that space can be generated.

This chapter first describes in detail the structure of the SPT, and in what domains it is useful for prediction. Then, we explain how to use a SPT for prediction tasks. After defining the structure and use of the SPT, we describe the method for growing the trees. Finally, we describe ensembling multiple SPTs into a SPT forest.

A SPT has a decision tree-like structure with conditional probability tables at the leaf nodes. The conditional probability tables at the leaf nodes store the probability of an event occurring at a single point in space. We call this single point of interest the *focus* of the SPT, and a fixed sized square area around the focus is a *window*. The conditional probability tables are constructed using the results of simple questions asked about the window around the focus.

4.1 SPT Structure

A SPT has a decision tree-like structure, as shown in the example tree in Figure 4.1. Internal nodes of the tree contain questions about the area surrounding the focus point. The only types of question asked in the SPT are counting questions. In a discrete domain, a counting question takes the form "Is the number of objects of a type t in a window of a size w_q, greater than or equal to some threshold e?" A window of size w_q refers to an $w_q \times w_q$ square centered on the focus point. Because of the requirement of the center point, w_q must be odd. Different questions in the tree are not required to ask about the same window size; however, the focus (the center of the windows) must be the same for all questions in a tree. An example of a question that might appear in a SPT is, "Is the number of edge points in a three by three window around the focus point greater than or equal to one?" Figure 4.2 shows an example of this question evaluated at two points on the board. Algorithm 4.1 describes this process in pseudocode.

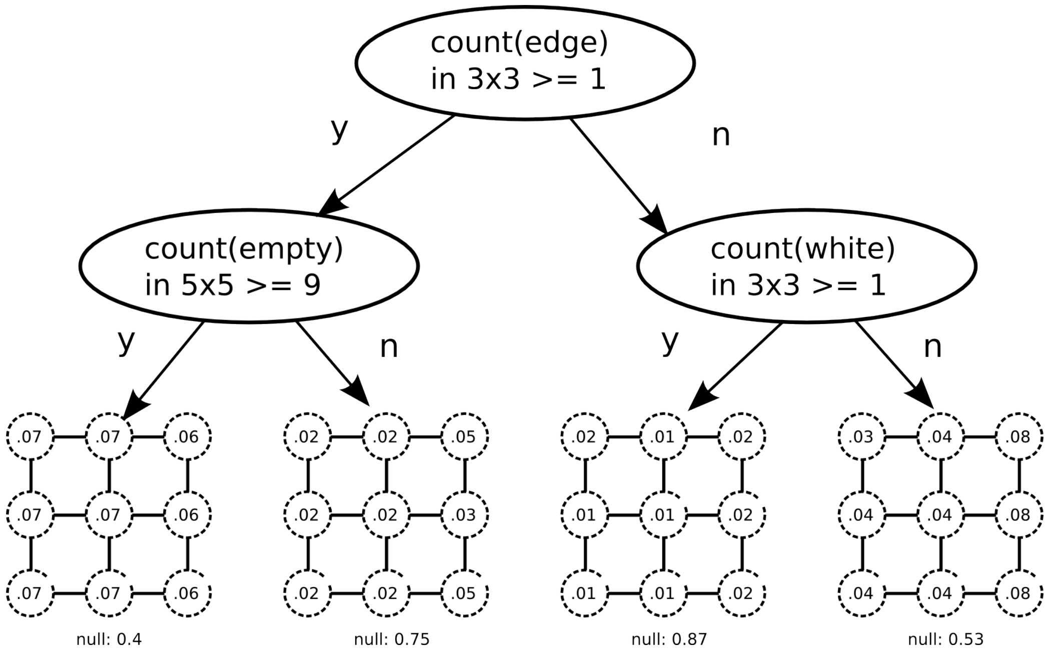

Figure 4.1: A Spatial Probability Tree with a window size w_p of 3 and a depth of 2. The root of the tree splits on the question "Is the number of edge positions in a 3×3 window around the focus point greater than or equal to 1?" The *yes* branch from the root node splits on the question "Is the number of empty points in a 5×5 window around the focus point greater than or equal to 9?" The *no* branch from the root node splits on the question "Is the number of white stones in a 3×3 window around the focus point greater than or equal to 1?"

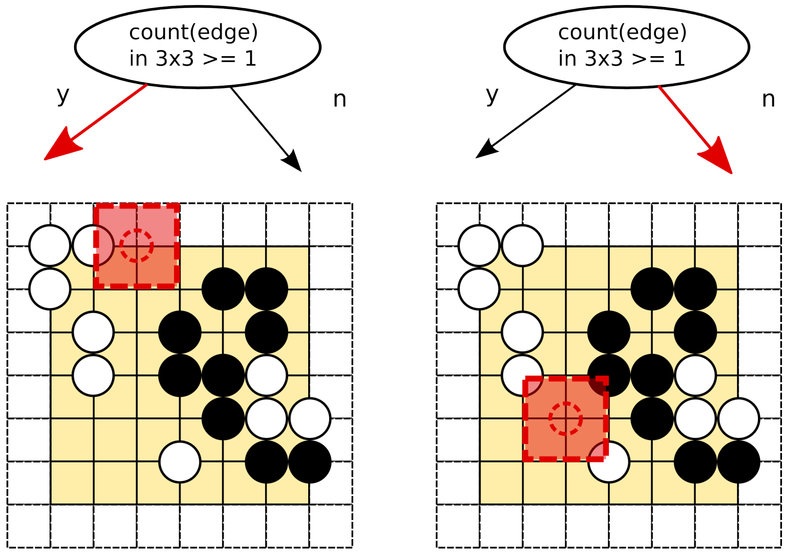

Figure 4.2: Sample Counting Question: In the example on the left, the question evaluates to yes, indicating that there is more than one edge point in the window. In the example on the right, the question evaluates to no, indicating that there are no edge points.

Algorithm 4.1: AskNodeQuestion

```
Input: $Q$ = Question, $B$ = Board Position, $F$ = Focus Point
Output: $Answer \in \{True, False\}$
$type, w_q, threshold \leftarrow GetCountingQuestion(Q)$
$offset \leftarrow w_q/2$
$count \leftarrow 0$
for $x \leftarrow GetFocusX(F) - offset$ to $GetFocusX(F) + offset$ do
    for $y \leftarrow GetFocusY(F) - offset$ to $GetFocusY(F) + offset$ do
        if $B(x, y) = type$ then
            $count \leftarrow count + 1$
        end
    end
end
return $count \geq threshold$
```

For simplicity's sake, in our discussion of the SPT, we reference only counting questions drawn from the unprocessed board. However, counting questions can be made more powerful through the use of knowledge fields. Knowledge fields and their application to SPT are discussed in Section 4.5

The leaves of the tree are fixed-size probability tables representing the board points in the window of size w_p around the focus point. For each board point, the table stores the probability that the next stone will be placed there. The leaf nodes also store the probability that the stone will not be played at any point inside the window, which we call the null probability. Figure 4.3 shows an example leaf node of window size three. It is worthwhile to note that while the probability tables at the leaf nodes are used both for prediction and growing the trees, the null probability is only used for growing the tree.

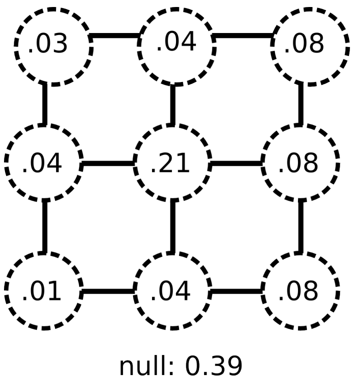

Figure 4.3: Example Leaf node with a window size w_p of three. The center cell stores the probability that the next stone will be played at the focus point. Each of the cells surrounding the focus point stores the probability that the next stone will be played at that cell. The null probability, listed below the cells, gives the probability that the next stone will not be played within the window represented by this leaf node.

4.2 SPT Domain

SPTs were designed for the task of expert move prediction in the game of Go. They answer the question, "Given a board position, what is the probability that a Go expert would place a stone at each empty point?" However, SPTs are not limited to Go. They can be used directly in any domain that matches the spatial constraints of Go, which has the following properties:

- the space is finite,
- the space is gridded,
- the space is either integer or real valued,
- the event being predicted occurs only at a single location in the space.

4.3 Evaluating a SPT

Before discussing how to use a SPT to predict expert moves, it is necessary to define what it means to evaluate a SPT. To evaluate the tree, we start from the root of the tree and ask the question at the root with respect to the focus point being considered. The answer to the question at the root leads either to another internal node containing another question to ask, or to a leaf node. If we arrive at an internal node, we ask the question contained there with respect to the focus being considered and then continue down the branch indicated by the answer. The tree-traversal process continues until a leaf node is reached. An algorithmic description of this process can be found in Algorithm 4.2. As discussed in Section 4.1, each leaf node contains a table of probabilities representing, for each board point in the window around the focus point, the probability that the next stone will be placed at the point.

Algorithm 4.2: EvaluateTree

```
Input: $R$ = SPT Root Node, $B$ = Board Position, $F$ = Focus Point
Output: $L$ = Leaf Node
$currentNode \leftarrow R$
while $!IsLeafNode(currentNode)$ do
    if $AskNodeQuestion(GetNodeQuestion(currentNode), B, F)$ then
        $currentNode \leftarrow GetYesBranch(current)$
    end
    else
        $currentNode \leftarrow GetNoBranch(current)$
    end
end
return $currentNode$
```

In using a SPT for expert move prediction, we are answering the question: "For each legal move on the board, what is the probability that an expert would play at that point?” In order to answer this question, we combine the local probability tables stored at the leaves of the tree to construct an approximation of a probability table for the entire board. The pseudocode for this process can be found in Algorithm 4.3.

Algorithm 4.3: ApplyLeafNodeToGlobalTable

```
Input: $L$ = Leaf Node, $G$ = Global Probability Table, $F$ = Focus
$window \leftarrow GetWindowSize(L)$
$offset \leftarrow window/2$
for $x \leftarrow -offset$ to $offset$ do
    for $y \leftarrow -offset$ to $offset$ do
        $gx \leftarrow GetFocusX(F) + x$
        $gy \leftarrow GetFocusY(F) + y$
        $G(gx, gy) \leftarrow G(gx, gy) + L(x, y)$
    end
end
```

Constructing the table of probabilities for the entire board (hereafter referred to as the *global table*) is straightforward. We begin with a blank table with one cell for each board point. For each board point, the SPT is evaluated, providing a local table of probabilities centered on that board point. That local table of probabilities is added to corresponding cells of the global table. As the local probability tables produced by evaluating the SPT at each board point overlap, it is necessary to combine values as the global table is populated. To combine overlapping values, we simply sum them. Figures 4.4 - 4.16 show an example of this process for a very small SPT. This example is intended to graphically explain Algorithms 4.2 - 4.4.

Because of the way the table is constructed, certain cells on the table are constructed from fewer local tables of probabilities. Specifically, cells corresponding to corners and edges on the board are constructed from fewer local tables of probabilities than board points in the interior of the board. For example, for a 3×3 window size, corners are covered by four local tables of probabilities and edges are covered by six, whereas every other board point is covered by nine. To account for this, we divide each cell by the number of leaf nodes that covered it. Finally, the table is normalized so that it sums to one. A pseudocode description of this process can be found in Algorithm 4.4.

Algorithm 4.4: ConstructGlobalTable

```
Input: $R$ = SPT Root Node, $B$ = Board Position
Output: $G$ = Global Probability Table
$G \leftarrow$ table of same size as $B$, initialized to zeros.
for $x \leftarrow 0$ to $GetSizeX(B)$ do
    for $y \leftarrow 0$ to $GetSizeY(B)$ do
        $focus \leftarrow (x, y)$
        $leaf \leftarrow EvaluateTree(R, B, focus)$
        $ApplyLeafNodeToGlobalTable(leaf, G, focus)$
    end
end
$Normalize(G)$
return $G$
```

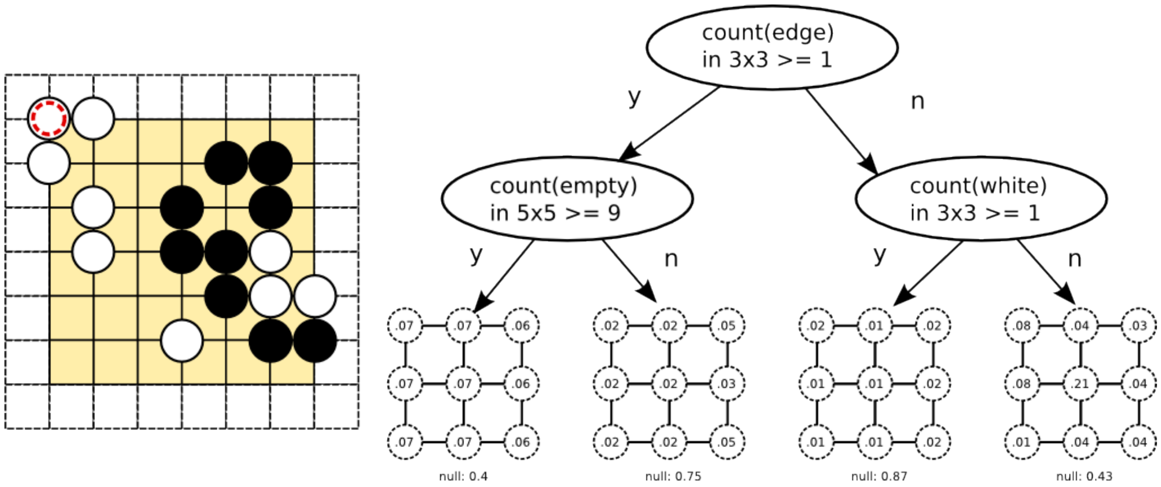

Figure 4.4: An example of the SPT evaluation process: The starting focus point is in the upper left corner of the board, as indicated by the dotted red circle. The focus point is indicated in this manner in all subsequent examples. The board represents the empty global table of probabilities that will be populated as the tree is evaluated.

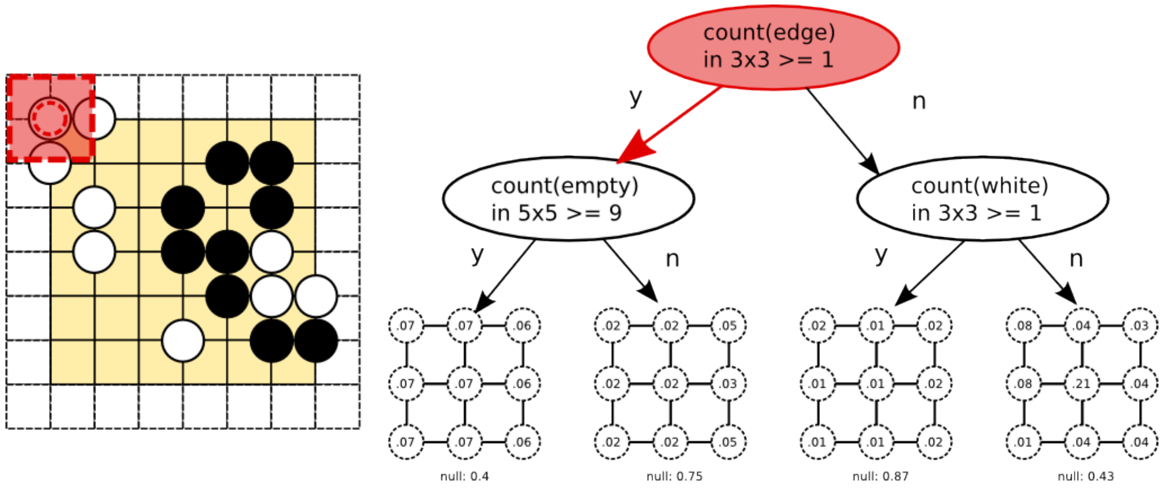

Figure 4.5: An example of the SPT evaluation process: The first question is evaluated, counting the number of edge nodes in the 3 × 3 window. The shaded square with a dotted outline indicates the window being considered by the question. In this case, there are five edge points within the window, so the answer is yes, and the SPT continues down the *yes* branch.

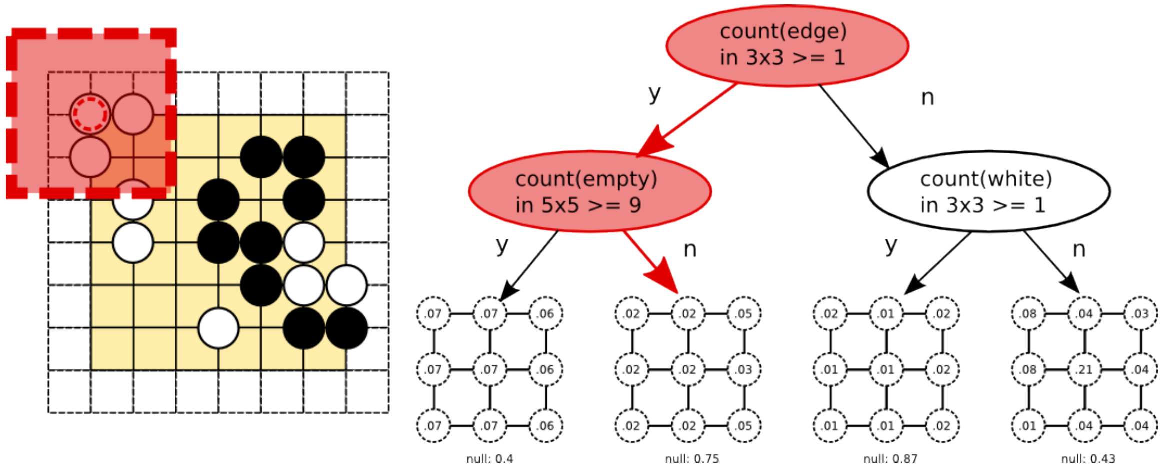

Figure 4.6: An example of the SPT evaluation process: The next question evaluated counts the number of empty points (points without stones) in a 5 × 5 window around the focus. In this case there are only five empty points in the window, which sends the SPT down the *no* branch.

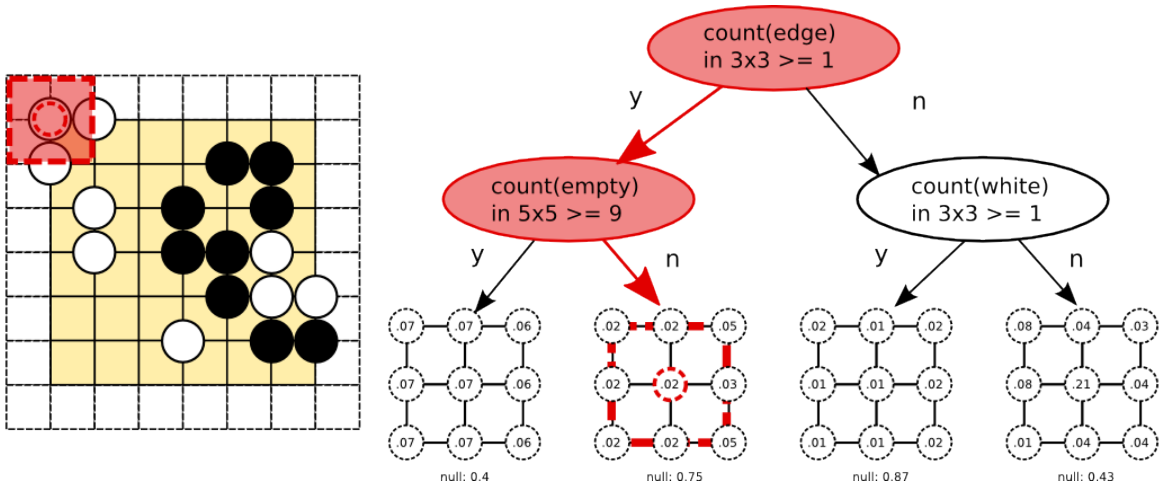

Figure 4.7: An example of the SPT evaluation process: The SPT is now at a leaf node covering the area represented by the shaded square.

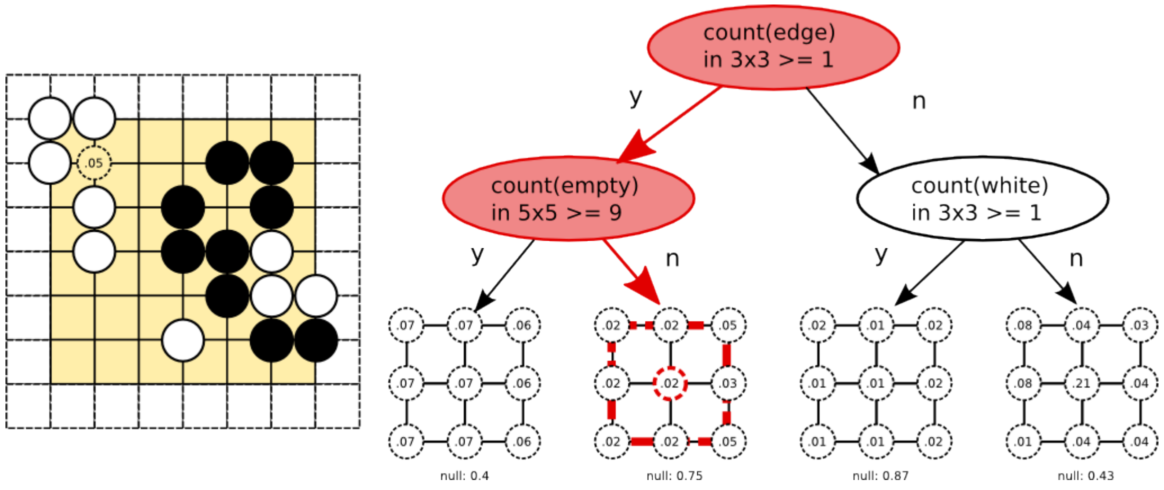

Figure 4.8: An example of the SPT evaluation process: The probabilities from the leaf node are added to the corresponding cells in the empty global table of probabilities.

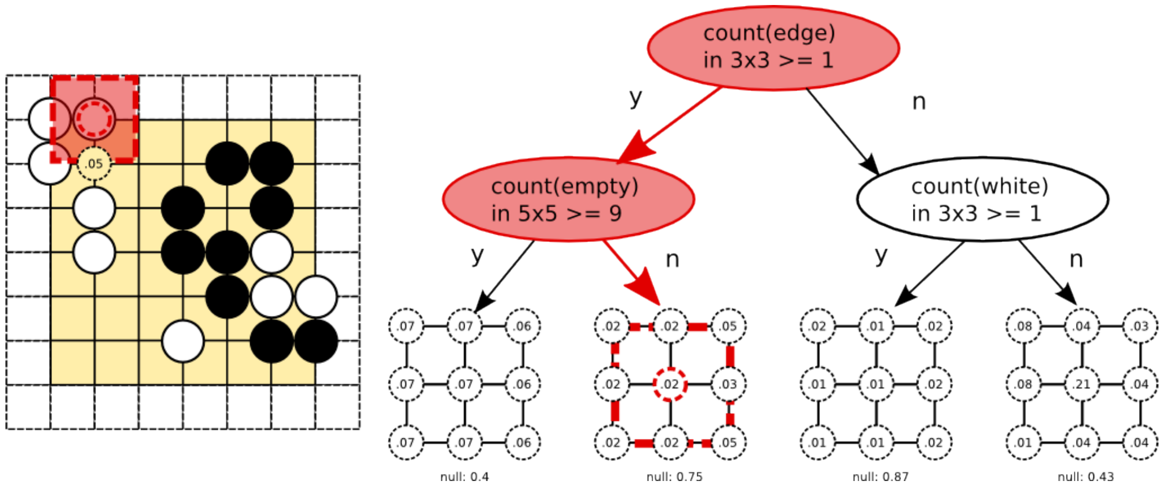

Figure 4.9: An example of the SPT evaluation process: The focus point moves over and the tree is evaluated once more, reaching a leaf node.

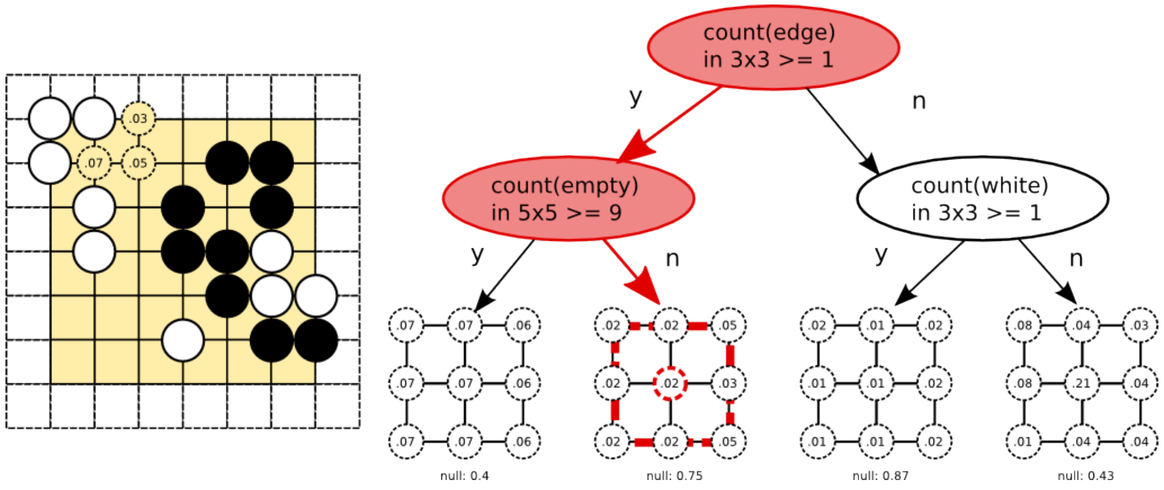

Figure 4.10: An example of the SPT evaluation process: The values from the leaf node's table of probabilities are once again added to the global table of probabilities.

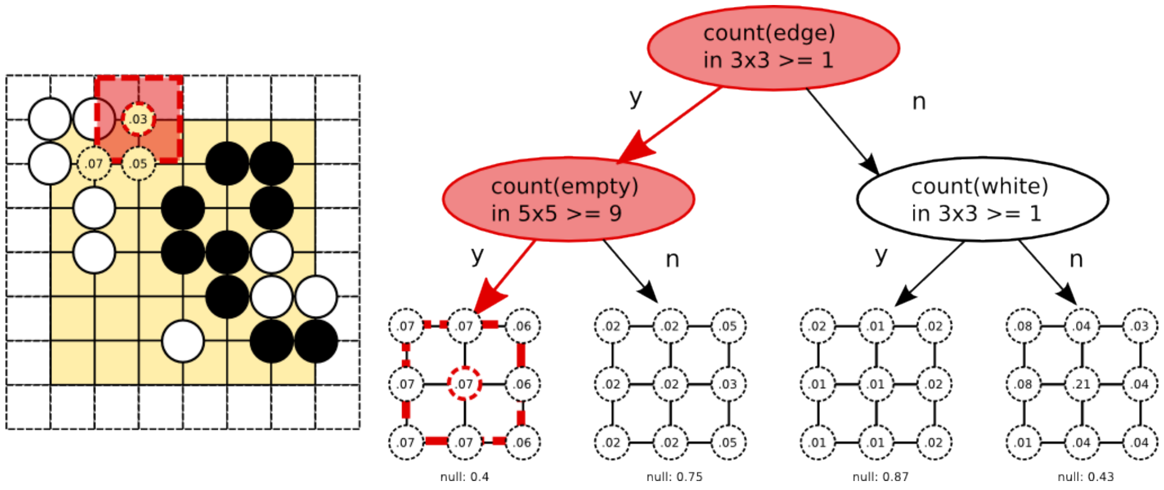

Figure 4.11: An example of the SPT evaluation process: The tree is evaluated at the next focus point and the probabilities in the global table are then updated.

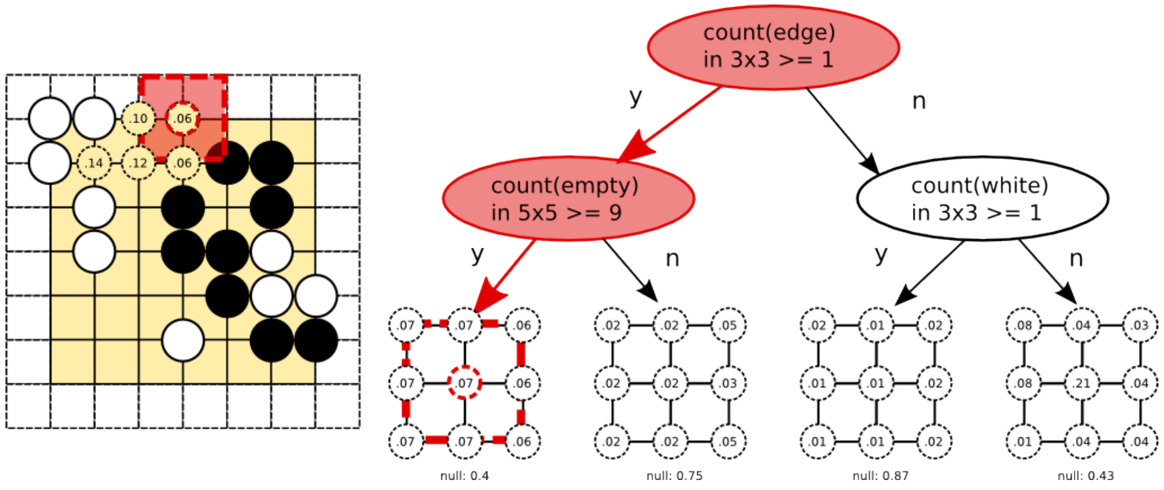

Figure 4.12: An example of the SPT evaluation process: The tree is evaluated at the next focus point and the probabilities in the global table are then updated.

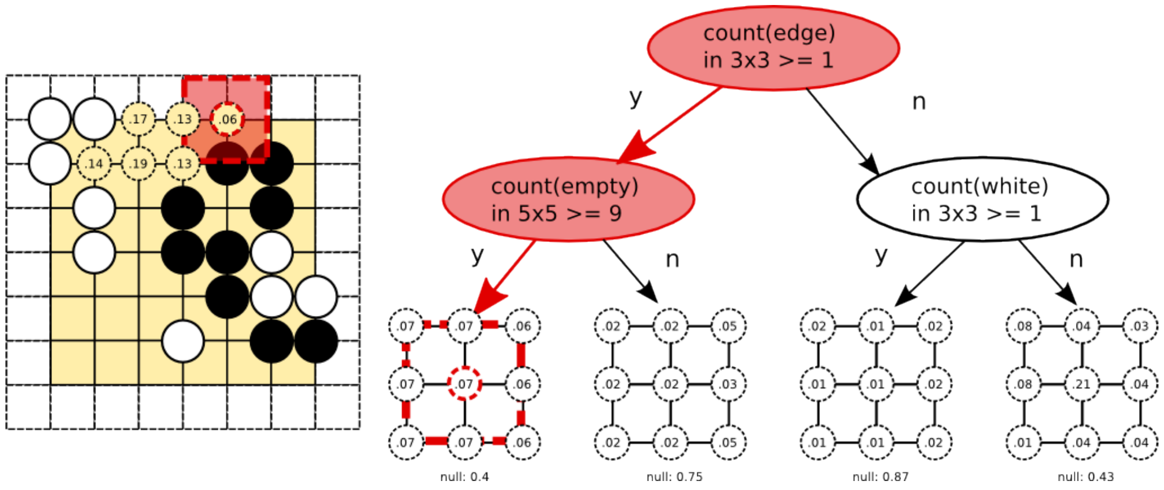

Figure 4.13: An example of the SPT evaluation process: The tree is evaluated at the next focus point and the probabilities in the global table are then updated.

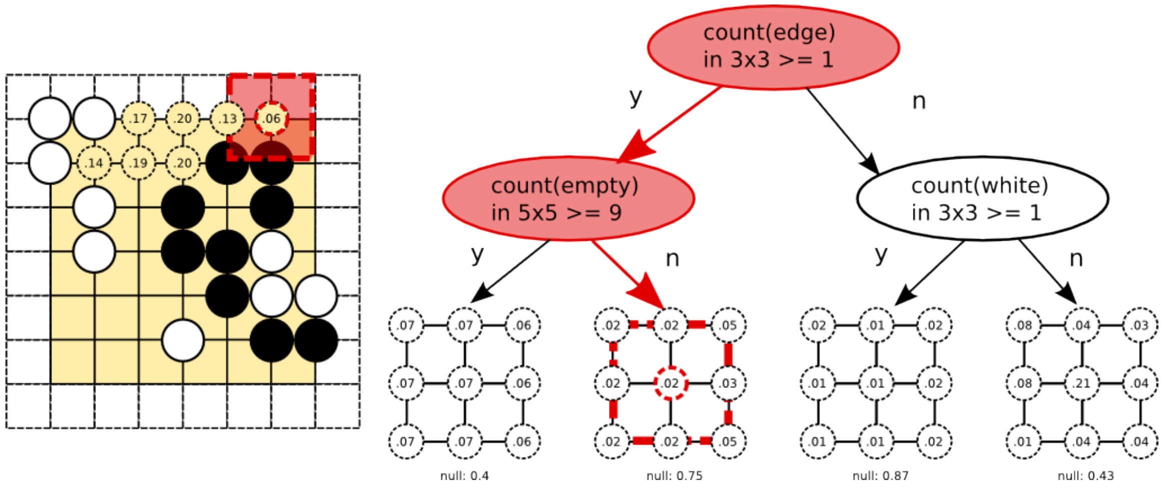

Figure 4.14: An example of the SPT evaluation process: The tree is evaluated at the next focus point and the probabilities in the global table are then updated.

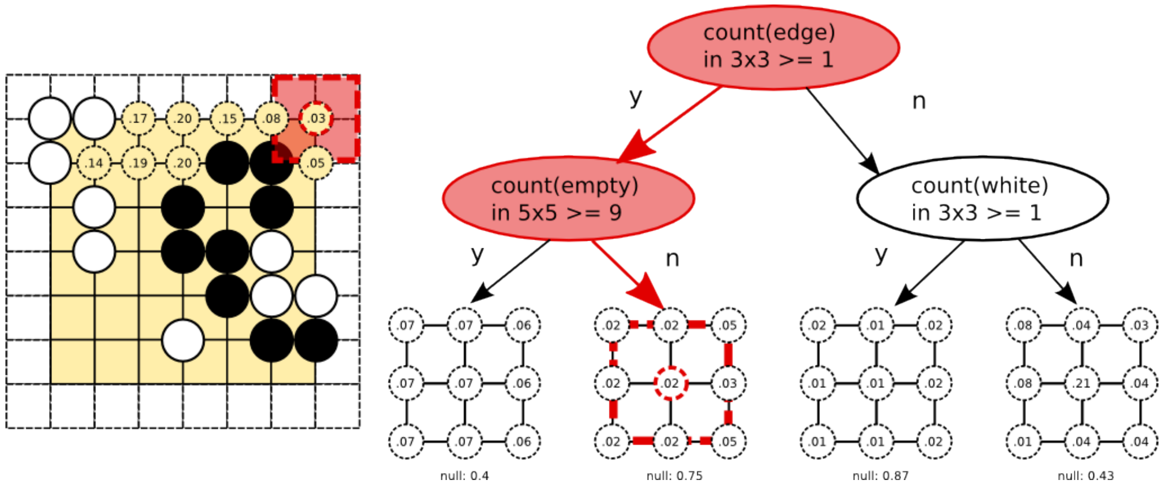

Figure 4.15: An example of the SPT evaluation process: The tree is evaluated at the next focus point and the probabilities in the global table are then updated.

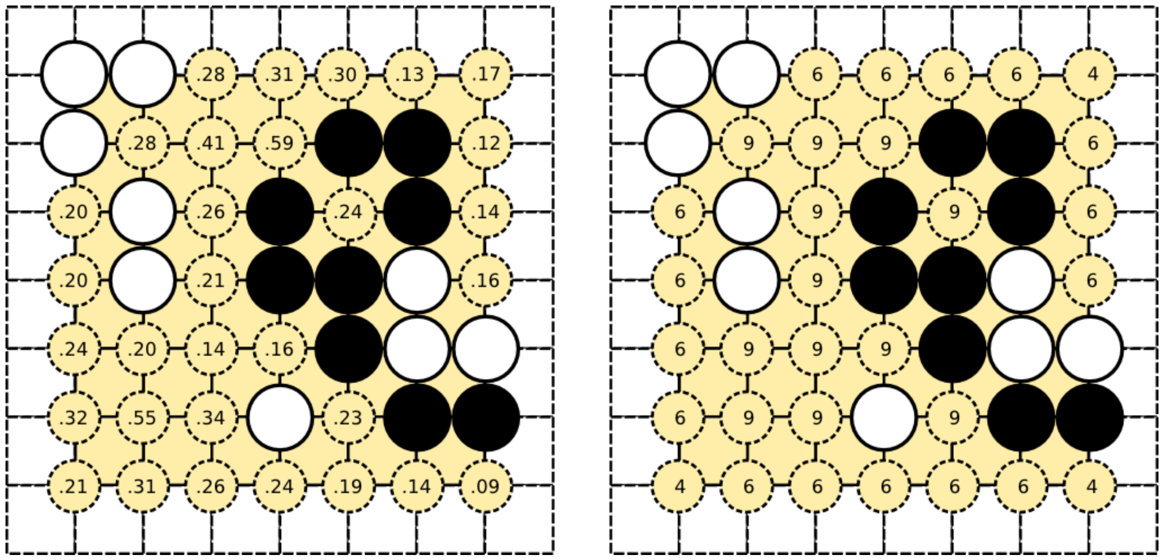

Figure 4.16: An example of the SPT evaluation process: On the left is the global table of probabilities. The board on the right shows how many leaf nodes cover each point on the board. The global table of probabilities is normalized to account for the uneven coverage of leaf nodes. Each value in the global table of probabilities is divided by the leaf node cover count at the corresponding point. The board is then normalized to sum to one by dividing the value of each cell by the sum of all of the cells.

4.4 Growing a SPT

SPTs are grown using *local* board position and next move pairs. Global board positions, as mentioned in previous sections, refer to the entire Go board. *Local* board positions are a *perspective* on the global board position with respect to a particular focus point. This is a subtle distinction, but it is convenient for undersampling. To transform a global board position into a local board position, we pair a focus point with the global board position. Figures 4.17 - 4.19 shows an example of the transformation process when the reader can see the process of undersampling the data. Note that prior to undersampling, the global board position is paired with every point on the board; for a 19×19 board, a single global board position becomes 361 local board positions.

The local board positions created from the global board position are classified according to whether or not the next move occurs within the window of size w_p around the focus point. A local board observation is positive if the next move occurs within the window around the focus point. An observation is negative if the next move does not occur in the window around the focus point. While the ratio of positive to negative examples will vary depending on window size, the number of negative examples, in general, is far greater than the number of positive examples. When we originally implemented the SPTs, we attempted to learn on the highly skewed data and results were poor. This is a well-known problem in machine learning and we used the standard approach of solving this problem by undersampling the negative observations (Drummond and Holte 2003). We chose to make the number of negative examples equal to the number of positive examples.

Once we have constructed a training set by transforming the global board positions into local board positions and undersampling the negative examples, we can grow the trees.

The tree-growing process begins by creating a tree with a single leaf node containing an empty $w_p \times w_p$ table at the root. Then, we aggregate all the examples in the training set at the single leaf node at the root of the tree. This means, for each example in the training set, we increment the values in the leaf node according to whether the next move occurred within the $w_p \times w_p$ window around the focus point. If the next move occurred within the leaf node's window, its position is recorded by incrementing the value of the cell corresponding to the board point where the next move was played. If the next move did not fall within the leaf node's window, the null

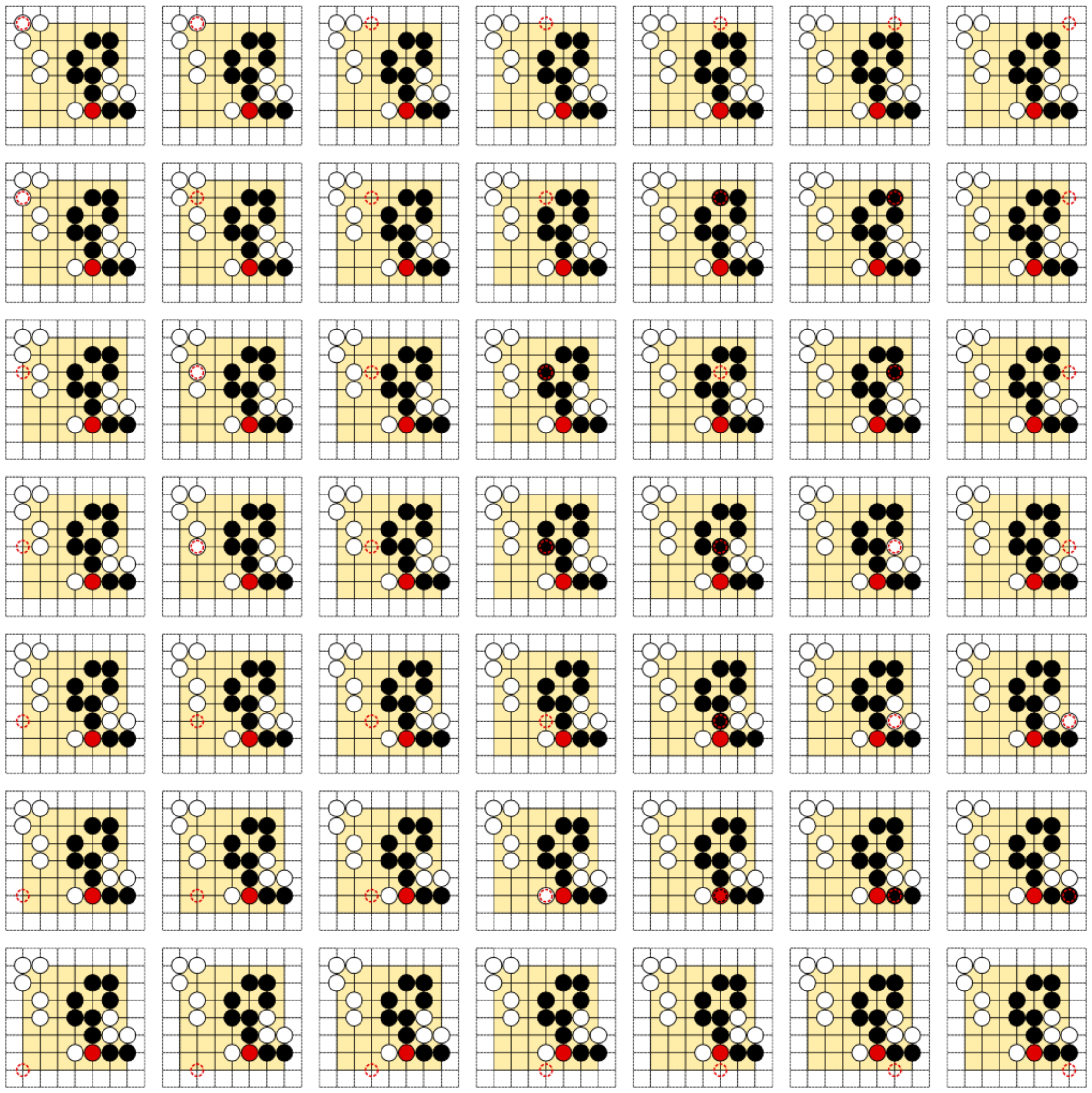
Figure 4.17: Undersampling process: First we transform the global board position into local positions by pairing it with every possible focus point. The red dotted is the focus point: the red shaded circle is the next move.

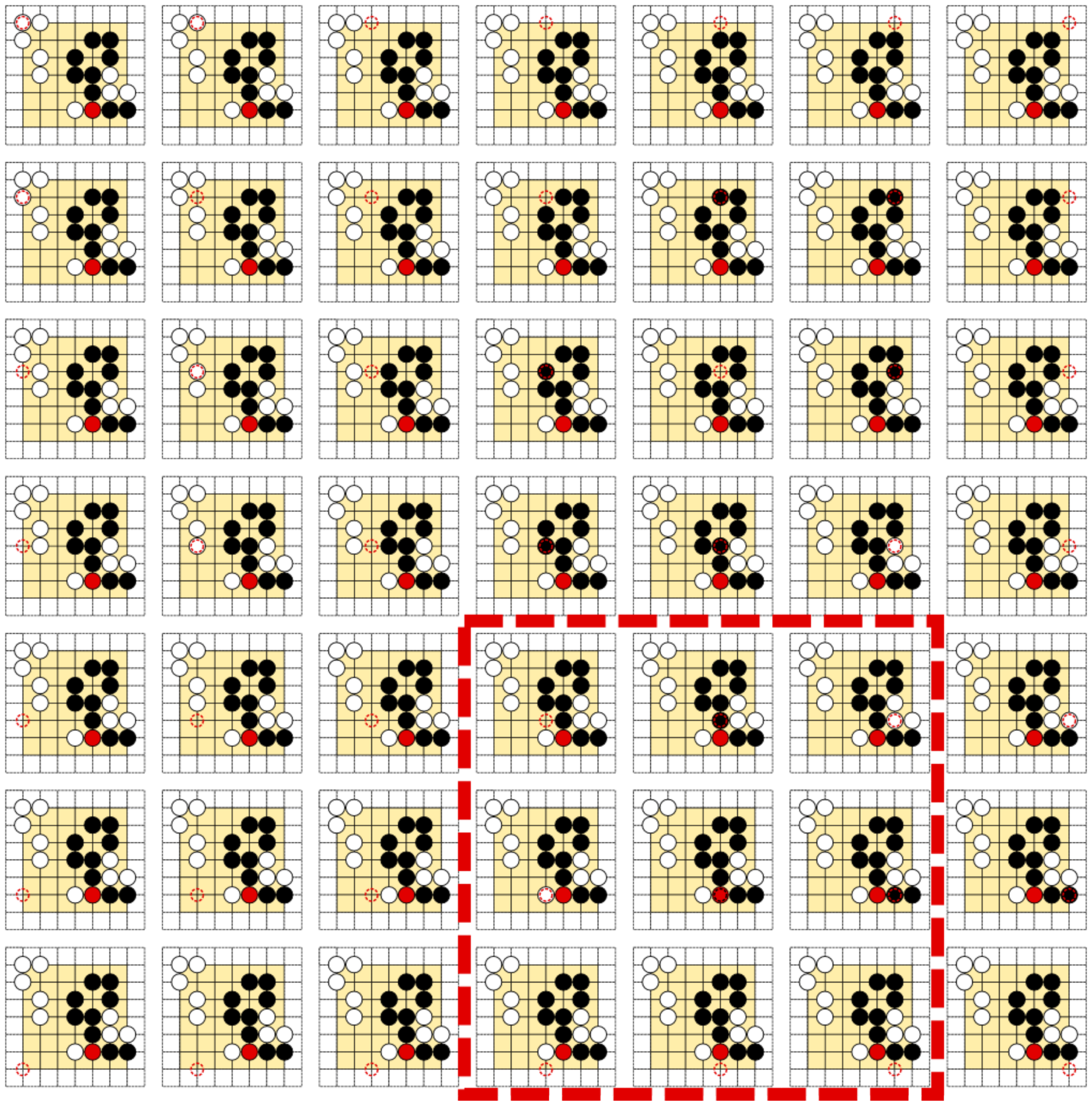

Figure 4.18: Undersampling process: The red dotted square indicates the positive examples, where the focus point is within a 3×3 window of the next move.

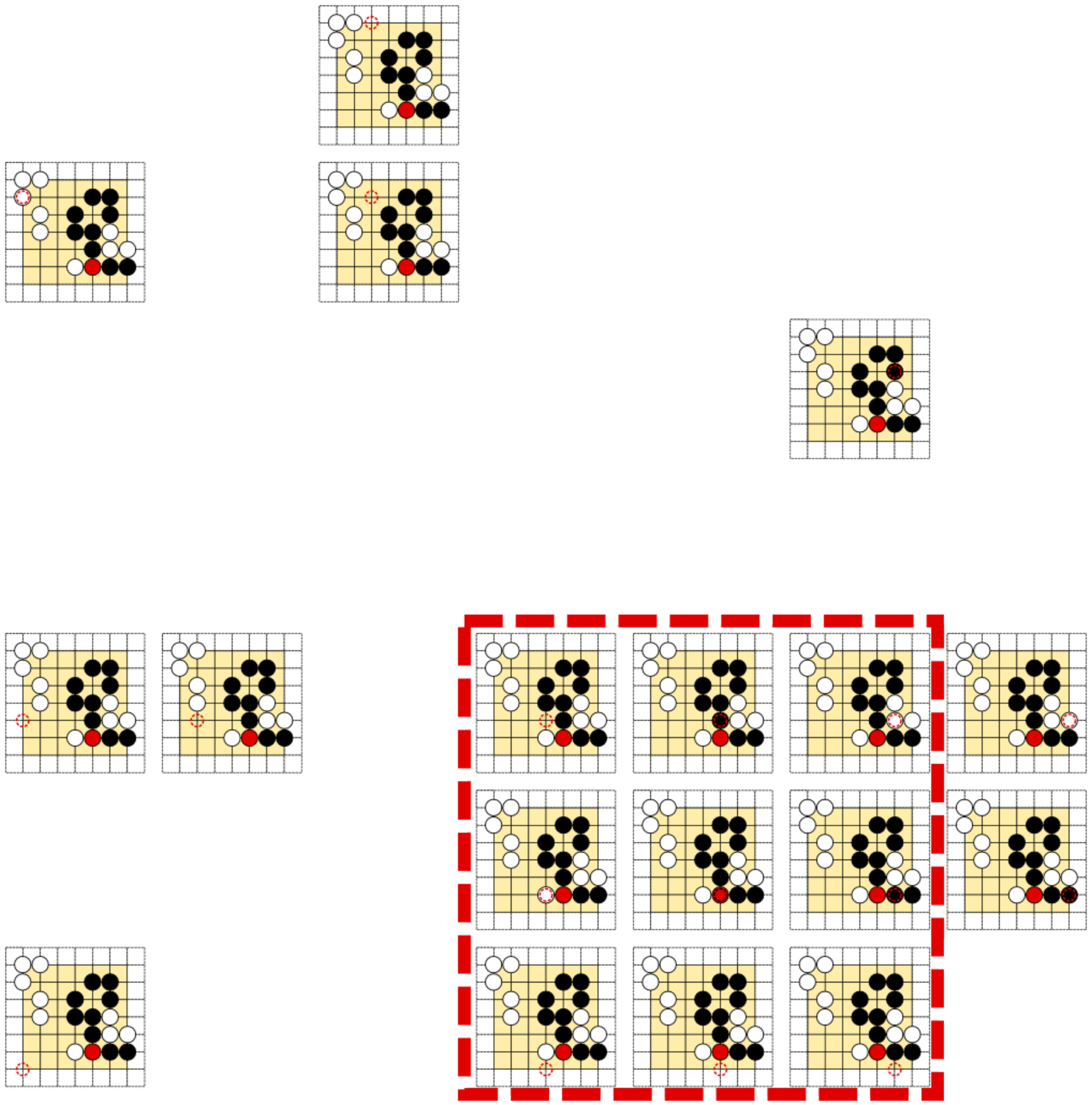

Figure 4.19: Undersampling process: All of the positive examples are added to the training set, in addition to an equal number of randomly selected negative examples.

count is incremented. The number of examples the leaf node has seen is also recorded; when there are no more examples to aggregate at that leaf, the counts in the cells are divided by the total number of examples seen by the leaf node, transforming the table of counts into a table of probabilities. This process is described in Algorithm 4.5.

Algorithm 4.5: CreateLeafNode

```
**Input**: $D$ = Processed Data
**Output**: A Leaf Node
$L(x, y) \leftarrow 0\ \forall x, y \in w_p$
$nullProbability \leftarrow 0$
**for** $d = boardPosition, focus, expertMove \in D$ **do**
    $w_p \leftarrow GetWindowSize(L)$
    **if** $InWindow(w_p, focus, expertMove)$ **then**
        $L(expertMove) \leftarrow L(expertMove) + 1$
    **end**
    **else**
        $nullProbability \leftarrow nullProbability + 1$
    **end**
**end**
$L \leftarrow L/Length(D)$
$nullProbability \leftarrow nullProbability/Length(D)$
$SetNullProbability(L, nullProbability)$
**return** $L$
```

The single leaf node tree we have created contains a table of probabilities representing the probability that the next move will occur in a window of size w_p anywhere on the board. This is not a useful predictor.

To develop more useful leaf nodes, it is necessary to *split* the data in a meaningful way. A split consists of a question (as discussed in Section 4.1), two leaf nodes containing the probability the next stone will be played in a window of size w_p around the focus point given the answer to the question, and the set of examples used to populate the table of probabilities at each leaf node. Figure 4.20 shows an example split. The process of splitting the data is described in Algorithm 4.6.

To create a split, we require a question. We generate a number of questions equal to the parameter s using examples in the training set. For each question, a random

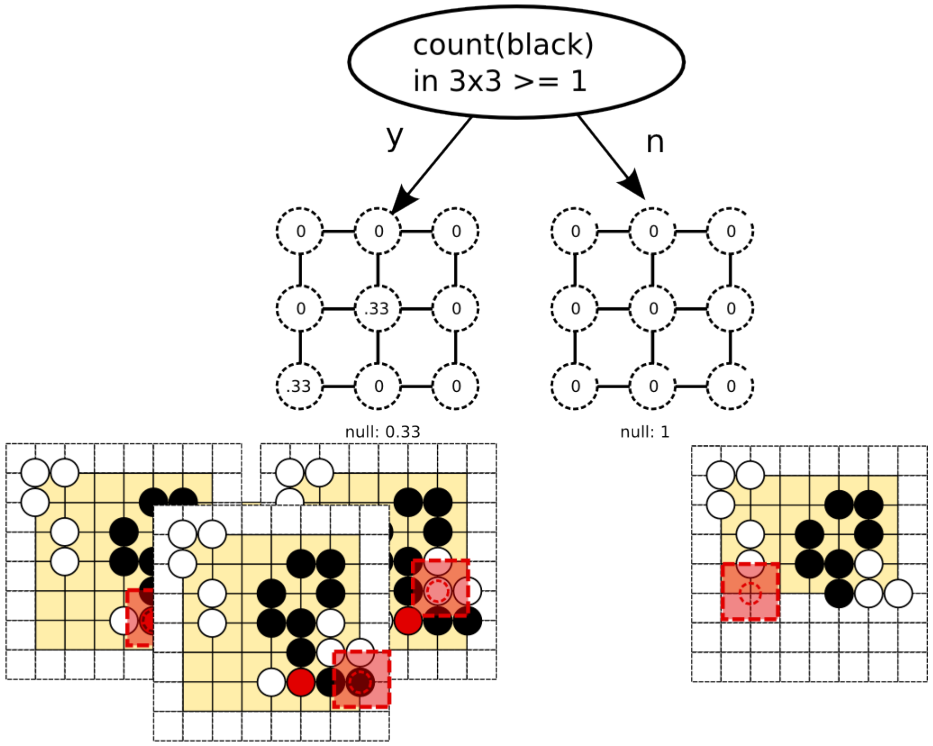

Figure 4.20: Example Split: A split consists of a question, two leaf nodes, and the set of examples that accumulate at each leaf node.

Algorithm 4.6: SplitData

```
Input: $Q$ = Question, $D$ = Data
Output: yesData, noData
$yesData \leftarrow \emptyset$
$noData \leftarrow \emptyset$
for $d = boardPosition, focus, expertMove \in D$ do
    if $AskNodeQuestion(Q, boardPosition, focus)$ then
        $Append(yesData, d)$
    end
    else
        $Append(noData, d)$
    end
end
return $yesData, noData$
```

stone type t is selected, along with a random focus point p and window size w_q. Then, for a randomly selected example, we ask the question, "How many stones of type t are present in the window of size w_q around the focus point p?" The answer to that question, e becomes the threshold used in the question for the node, which is, "Is the number of stones of type t present in the window of size w_q around the focus point p greater than the threshold e?"

Once we have the question to use to split the data, the actual splitting occurs. For each example in the training set, the question is evaluated. If the answer to the question is yes, that example is aggregated at the leaf down the *yes* branch. If the answer is no, that answer is aggregated at the leaf down the *no* branch. This process is repeated until no examples remain.

We want to replace the single leaf node root of the tree with a split that divides the data in a useful way. Further, we would like to replace the single leaf node root with a split that divides the data in the best way, or at least in a way that is better than the other splits generated. Therefore, we need a method to assess the goodness of a split relative to other splits.

The method used to assess the goodness of a split is a score based on the Likelihood Ratio. Likelihood, for a leaf node n and example x, is the likelihood that the example can be explained by the leaf node. This value is simply read from the table of probabilities at the leaf node. For example, if the next stone in x was placed one point to the left of the focus point, the likelihood that x is explained by n is the value in the cell one cell to the left of the cell corresponding to the focus point in the table of probabilities stored at n. If the next stone was not played within the window covered by n, the likelihood that x is explained by n is the null probability stored at n.

Algorithm 4.7: EvaluateSplit

```
Input: $Q$ = Question, $D$ = Processed Data
Output: Split Rating
$nullLeaf \leftarrow CreateLeafNode(D)$
$yesData, noData \leftarrow SplitData(Q, D)$
$yesLeaf \leftarrow CreateLeafNode(yesData)$
$noLeaf \leftarrow CreatLeafNode(noData)$
$yesLikelihood \leftarrow GetLogLikelihood(yesLeaf, yesData)$
$noLikelihood \leftarrow GetLogLikelihood(noLeaf, noData)$
$nullLikelihood \leftarrow GetLogLikelihood(nullLeaf, D)$
$likelihoodRatio \leftarrow yesLikelihood + noLikelihood - nullLikelihood$
return $likelihoodRatio$
```

The likelihood that a set of examples is explained by a particular leaf node is the product of the likelihoods for the individual examples, given that leaf node. However, because many of the likelihoods are very small, the likelihood of large sets of examples eventually becomes zero. The standard approach to mitigate this is to use the log likelihoods. Therefore, instead of multiplying the likelihoods of the individual examples in the set, we add the logs of the individual likelihoods.

The log likelihood of a leaf node, then, is calculated by adding the log likelihood of each example shown to that leaf node with respect to that leaf node.

To calculate the Likelihood Ratio, we first calculate the log likelihood of the split. This is done by calculating the log likelihood for the set of examples that were aggregated at the leaf node at the end of the yes branch with respect to the leaf node at the end of the yes branch and adding it to the log likelihood for the set of examples that were aggregated at the leaf node at the end of the no branch with respect to the leaf node at the end of the no branch.

The log likelihood of the original leaf node must also be calculated. This value represents the null likelihood.

The Likelihood ratio is simply the difference between the likelihood of the split and the null likelihood, $\ln(L_s) - \ln(L_{null})$. We prefer the split with the higher Likelihood Ratio. The split evaluation process is described in Algorithms 4.7 and 4.8.

Algorithm 4.8: GetLogLikelihood

```
Input: $L$ = Leaf Node, $D$ = Data
Output: Log Likelihood that D is explained by L
$likelihood \leftarrow 0$
for $d = boardPosition, focus, expertMove \in D$ do
    $w_p \leftarrow GetWindowSize(L)$
    if $InWindow(w_p, focus, expertMove)$ then
        $likelihood \leftarrow likelihood + Log(L(expertMove))$
    end
    else
        $likelihood \leftarrow likelihood + Log(GetNullProbability(L))$
    end
end
return $likelihood$
```

To replace the leaf node at the root of the tree with a split, we generate several splits and choose the one with the highest likelihood ratio. Now there are two new leaf nodes that can potentially be better explained by a split. For each new leaf node, we generate new splits, using only the examples that were aggregated at that leaf node. The split with the highest Likelihood Ratio replaces the leaf node. This process is repeated until one of the following conditions is met:

- if the number of examples at a leaf node is less than the minimum node size m,
- if the maximum tree depth d has been reached,
- if the leaf node explains the data better than any of the splits generated.

The entire process of growing a SPT is summarized in Algorithm 4.10.

4.5 Knowledge Fields

One way to expand the power of counting questions is through the concept of *Knowledge Fields*. Knowledge fields are gridded spaces that are the result of some transform

Algorithm 4.9: FindBestSplit

```
Input: D = Processed Data, s = Questions Sampled per Split
Output: Best distinction found given parameters, otherwise $null$
$bestQ \leftarrow null$
$bestV \leftarrow 0$
for $i = 1$ to $s$ do
    $Q \leftarrow$ generate random question
    $value \leftarrow EvaluateSplit(D, Q)$
    if $value \geq bestV$ then
        $bestQ \leftarrow Q$
        $bestV \leftarrow value$
    end
end
return $best$
```

or calculation on the Go board. These fields can be used to incorporate expert knowledge. For example, in our experiments we used the following knowledge fields:

- Liberty Field - the number of of liberties for each stone on the board.
- Capture Field - the number of opponent's stones that will be captured by placing a stone for each empty point on the board.
- Opponent Capture Field - the number of stones that the opponent could capture by placing a stone for each empty point on the board.

Knowledge fields must be grounded to the Go board. That is, each point of the knowledge field must correspond to a single point on the Go board. Counting questions can then be asked with respect to a knowledge field in addition to the Go board.

4.6 SPT Forests

Although a single SPT can be a powerful spatial predictor, much of the recent work in machine learning demonstrates that an ensemble of models is more powerful than any single model. An ensemble is simply a collection of models whose predictions are combined to produce a single prediction. Common ensembling techniques include

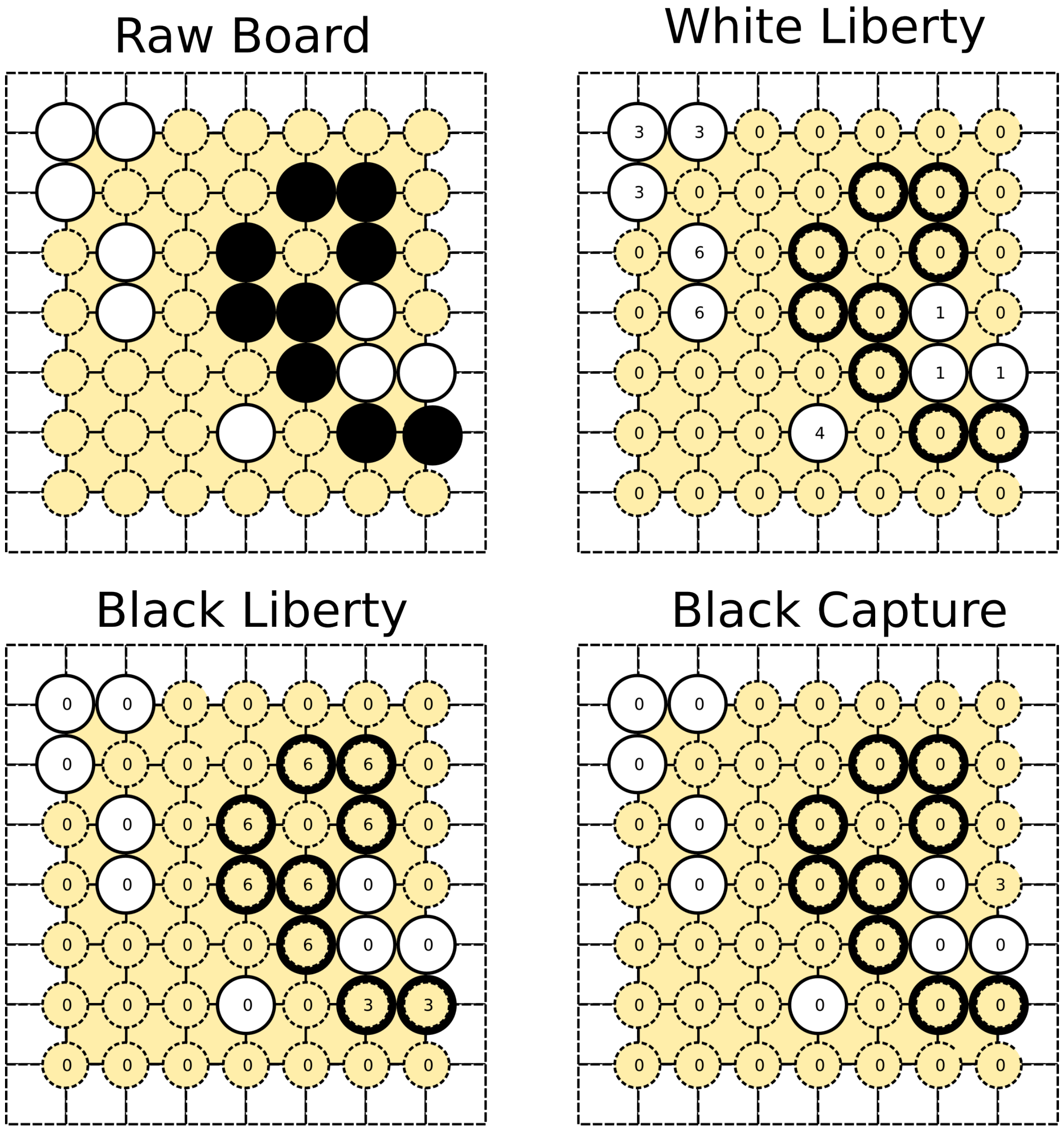

Figure 4.21: Example Knowledge Fields. The top left shows the Go board position; the other three boards show knowledge fields built off of the top left Go board. The White Liberty Field is defined relative to white stones. If a point does contain a white stone, then the value at that point is the number of liberties of the group containing the stone at that point. If a point does not contain a white stone, the value is zero. The Black Liberty is defined the same as the White Liberty field, except for black stones instead of white. The Black Capture Field is zero at occupied spaces. At every empty space, it's the number of white stones that black could capture by playing a stone at that empty space.

Algorithm 4.10: GrowSPT

```
Input: $D$ = Processed Data, $c$ = Current Depth, $d$ = Maximum Depth, $m$ =
        Minimum Leaf Node Support, $s$ = Questions Sampled per Split
Output: A SPT
if $c < d$ and $Length(D) > m$ then
    $node \leftarrow CreateNode(FindBestSplit(D, s))$
    if $node \neq \emptyset$ then
        $yesData, noData \leftarrow SplitData(GetQuestion(node), D)$
        $SetYesBranch(node, GrowSPT(ydata, c + 1, d, m, s))$
        $SetNoBranch(node, GrowSPT(ndata, c + 1, d, m, s))$
        return $node$
    end
end
return $CreateLeafNode(D)$
```

boosting, Bayesian Model Averaging, and Random Forests (Breiman 2001; Witten et al. 2011). A forest is an ensemble containing only decision trees. For this work, we ensemble a set of SPTs into an SPT forest.

Growing a forest is straightforward. The desired number of trees is grown independently as described in Section 4.4, each training on a random set of board positions drawn from the training data. If there was insufficient training data, we could ensure diversity in the forest by following the bootstrap randomization approach used by Random Forests (Breiman 2001). However, because the number of board positions used for training each tree is small relative to the number of board positions in the training data, this is not necessary in our case. Another approach we have explored was to train each tree on games from an individual player but it was not more powerful than simply training on different sets of games.

Once we have a grown a forest, it can be evaluated. To evaluate a SPT forest, each tree in the forest is evaluated, producing one probability table describing the whole board per tree. The resulting tables are summed and then normalized to provide the forest's probability table.

Chapter 5

Empirical Results

We empirically evaluated SPTs in computer Go using two measures. The first measured the Kth-move performance and the second assessed the performance of Fuego[1] (Enzenberger et al. 2010), a top computer Go player, both with and without the advice provided to UCT search by the SPTs. We used Fuego version 0.4.1, the current version at the time of the experiments.

The Kth-move measure of performance evaluates the ability of a model to predict expert moves. A model produces a ranked list of potential moves and we identify where in this ranked list of moves the actual expert move falls. For example, if the expert move was the fifth move, then it was within the top 5 moves. From this, we can ask a more general question: "Given multiple test examples, how often is the expert's move within the top k moves?" By varying k, and computing the percentage of the time the expert move is within the top-k moves across all test examples, we can compare to published results from state-of-the-art methods that predict expert moves (Stern et al. 2006). The motivation behind the Kth-move measure is that, in Go, there is not necessarily only one correct move in a given situation. Kth move data for each forest was gathered over 30 full Go games, selected randomly from 671 testing games.

In the Fuego integration experiments, we use a forest of SPTs to give advice to the Fuego UCT player. Fuego uses this advice to bias its UCT rollouts towards better moves. This is done using a theoretically-motivated additive modification to the UCT formula (Rosin 2010a). To test the performance of the player with the SPT advice, we play 1000 games against GnuGo version 3.6 level 10.[2] Each game uses 10,000 rollouts per move. These parameters were chosen based on advice from an expert in the field (Rosin 2010b).

[1]http://fuego.sourceforge.net/

[2]http://www.gnu.org/software/gnugo/

Parameter	Description
w_p	Leaf node window size
s	Number of questions sampled per split
m	Minimum leaf node support threshold
d	Maximum tree depth
n	Number of training examples

Table 5.1: SPT parameters

Our training data consists of 19×19 games drawn from the KGS Go Server[3] where both players are ranked at least 6 dan and the game lasts at least 70 moves. Dan indicates the level of skill possessed by the Go player. Expert players are ranked into nine dan grades, with players of higher dan possessing greater skill. By focusing on only the top players, we expect the moves to be more consistent. By including only games that last 70 moves or longer, we ensure that our data consists of full games.

Our first experiments analyzed the impact of the parameter choices on the performance of the SPT. (Table 5.1 shows the parameters of the SPT.) We measured the SPT's ability to correctly predict the expert move within the first move, the top 5, the top 10, and the top 20 moves. Figures 5.1, 5.2, 5.3, 5.5, and 5.4 show the results of this measure as a function of the five main parameters of the SPT. All of the experiments were conducted with the same parameters, except for the one being varied. For the parameter not being varied, the choices were: a window size $w_p = 3$, minimum leaf node support threshold $m = 30$, maximum tree depth $d = 10$, number of questions sampled per split $s = 300$, and number of training examples $n = 300$. These observations were drawn randomly from 781 training games. We examined the performance of forests of sizes 1, 5, 10, and 20.

As the tree depth is increased (Figure 5.1), the SPT is better able to predict the expert moves. The performance quickly levels out around depth 10, most likely in an interaction with the minimum leaf node support m and the number of training observations available. Figure 5.2 shows the same results as a function of the minimum number of observations required to split a leaf node. Here we see a decrease in performance as the minimum size is increased because the tree is unable to fully grow. As expected, increasing the number of questions sampled per split s improves the

[3]http://www.gokgs.com/

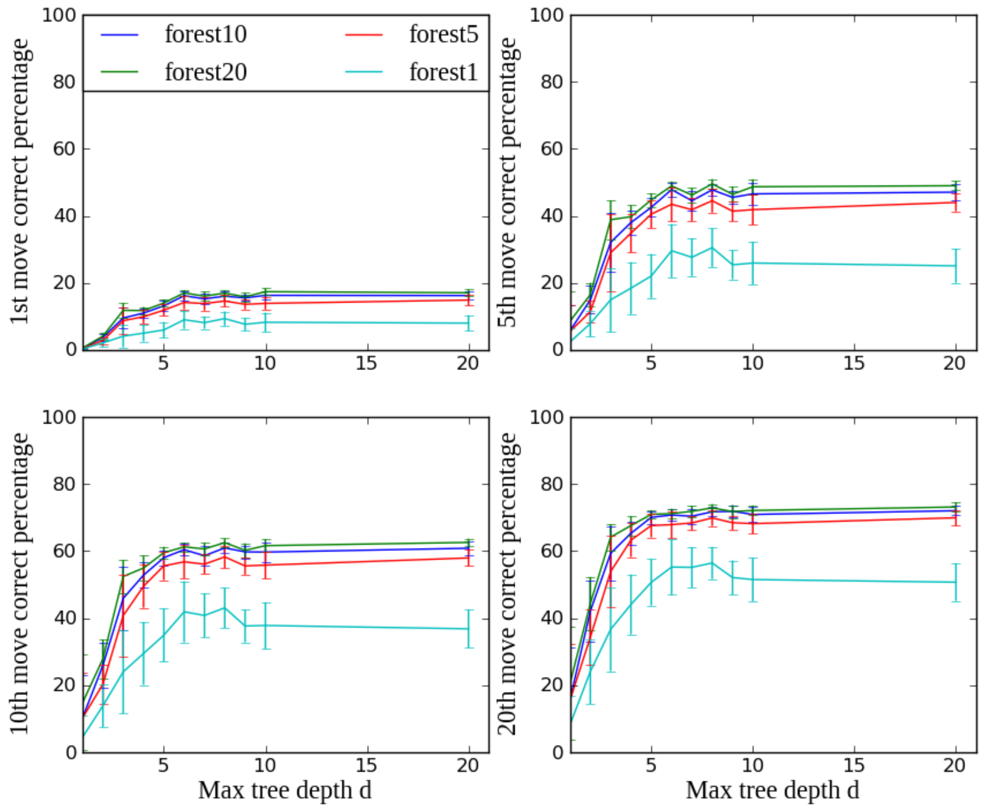

Figure 5.1: Percent of the time that the expert move is correctly predicted by the top 1, 5, 10 and 20th ranked moves as a function of the maximum tree depth d. The error bars represent the standard deviation.

performance of the tree (Figure 5.3). Performance asymptotes around 200 samples. The performance as a function of the size of the leaf node (Figure 5.4) peaks with leaf nodes of 3-5 and then shows a clear decrease as the size of the pdf increases. Finally, the performance as a function of the number of training examples quickly increases but levels off at around 1000 examples. These results show a robustness to parameter selection.

We performed n-way analysis of variance to explore parameter significance and potential interaction effects between parameters. In addition to expecting that all SPT parameters (found in table 5.1) impact performance, we expected that there would be an interaction effect between the number of questions sampled s and both the depth of the tree d and the number of training examples n. The results of the analysis

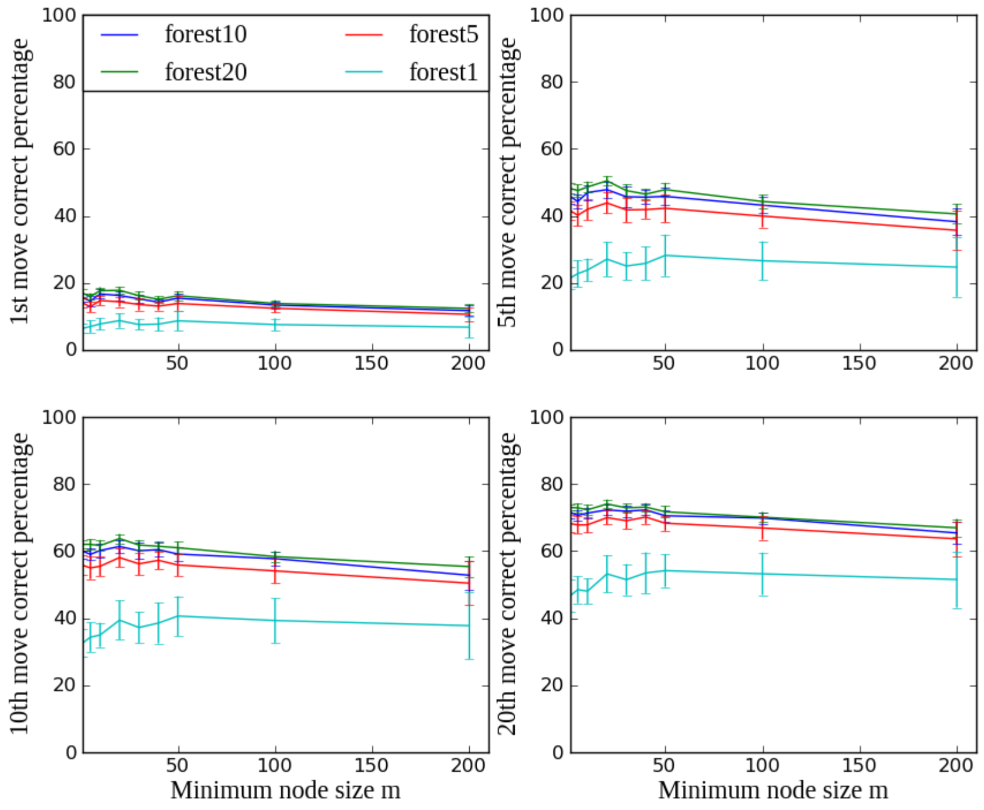

Figure 5.2: Percent of the time that the expert move is correctly predicted by the top 1, 5, 10 and 20th ranked moves as a function of the minimum number of instances required to split a leaf m. The error bars represent the standard deviation.

are shown in Table 5.2. All of the SPT parameters are significant at an α -value of 0.05, meaning that all parameters to the algorithm impact its performance. All of the interaction effects between parameters are significant except for the interaction between the number of observations and the leaf node window size, the interaction between the number of observations and the minimum node size, and the interaction between the maximum tree depth and the minimum node size.

Table 5.3 shows a comparison of our work with the work of Stern et al. (2006) and Araki et al. (2007). While these results suggest that SPT does not perform as well as Stern or Araki's system, it is important to note that since the systems use different sets of expert Go moves, the results are not directly comparable. Additionally, the SPT trains on only 300 board positions while the other systems train on millions of

Source	P-value
Number of questions sampled per split s	0
Number of training examples n	0
Maximum tree depth d	0.0255
Leaf node window size w_p	0.0360
Minimum leaf node support threshhold m	0
$s \times n$	0.0227
$s \times d$	0
$s \times w_p$	0
$s \times m$	0
$n \times d$	0.0196
$n \times w_p$	0.7009
$n \times m$	0.0972
$d \times w_p$	0.0001
$d \times m$	0.3564
$w_p \times m$	0

Table 5.2: N-way analysis of variance

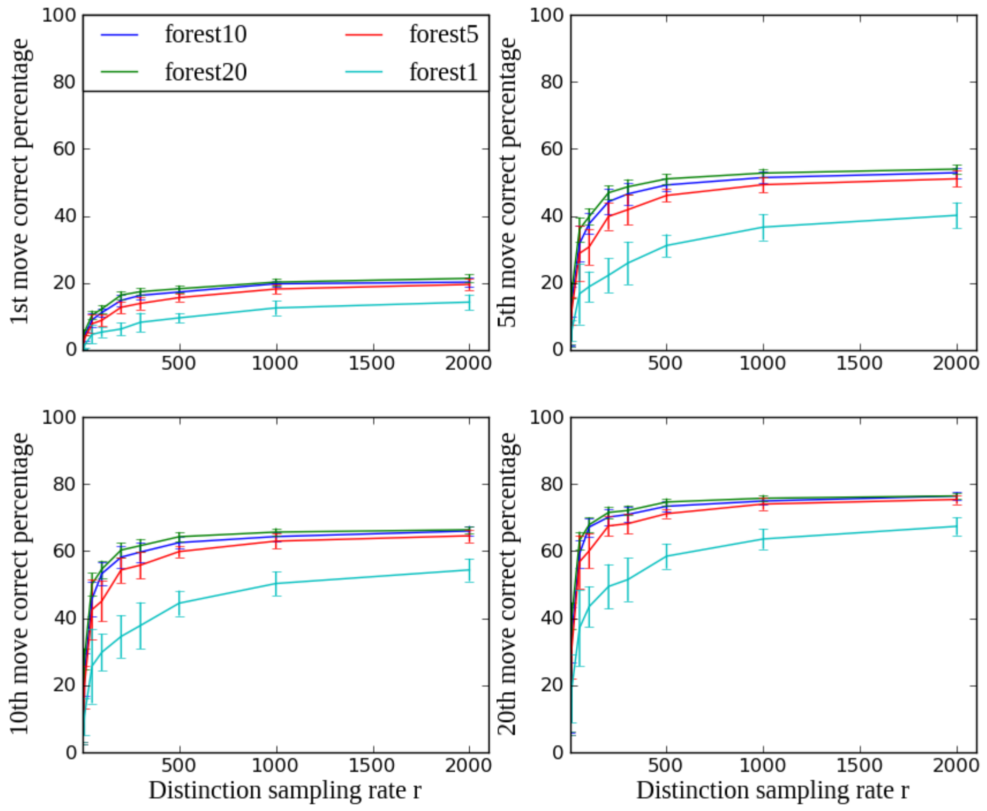

Figure 5.3: Percent of the time that the expert move is correctly predicted by the top 1, 5, 10 and 20th ranked moves as a function of the number of questions sampled per split s. The error bars represent the standard deviation.

positions. In some applications, the drastic reduction in necessary training data may be worth the reduction in performance.

Given the relative insensitivity to the SPT parameters, we used the same parameter set described above for our comparison against Fuego. We trained 30 SPTs and the forests of varying sizes were created by randomly selecting trees from these 30.

We compared our model to the collection of heuristic Go players provided by the Fuego framework. These heuristic players, like SPT, perform static evaluation of the board.

Figure 5.6 shows the Kth-move curves for SPT forests of size 1, 5, and 10, as well as random and the best two Fuego heuristic players. From this, we see that even a single SPT outperforms the best heuristic players. Furthermore, we see that a forest

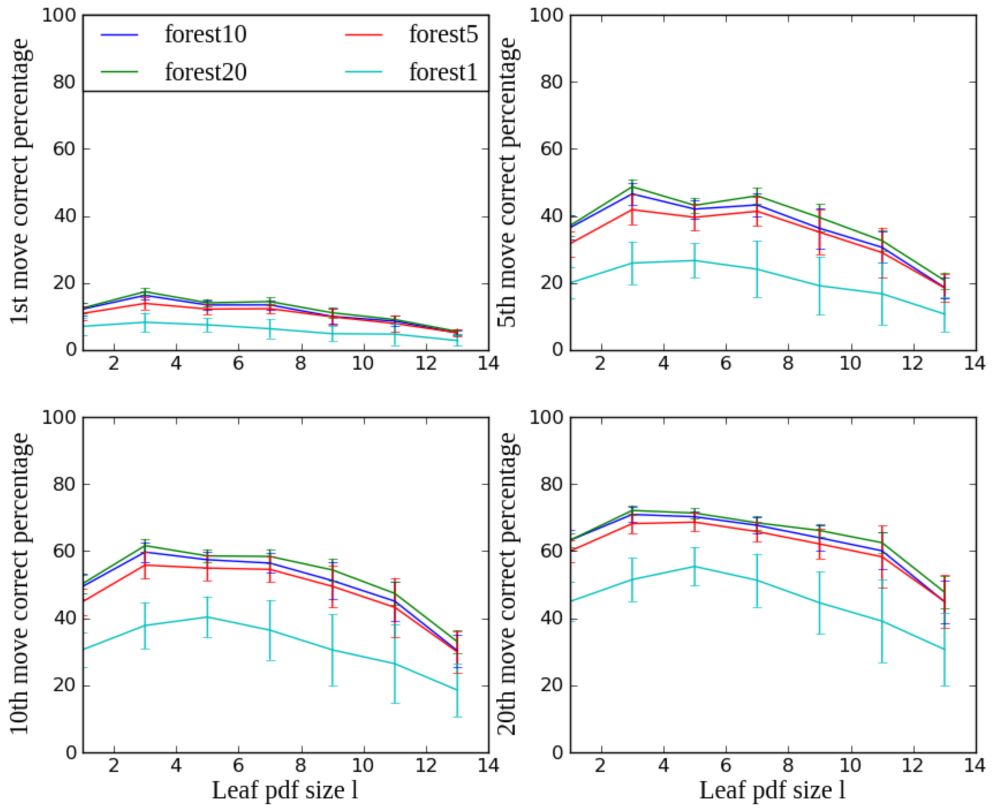

Figure 5.4: Percent of the time that the expert move is correctly predicted by the top 1, 5, 10 and 20th ranked moves as a function of the leaf node window size w_p. The error bars represent the standard deviation.

of 5 trees performs noticeably better than a single tree. Increasing the forest size beyond 5 does not appear to greatly affect the performance.

The Kth-move graphs indicate that the SPTs can predict expert moves well. To assess whether SPT can be used to effectively guide UCT search, we integrated the forests into Fuego and compared the results against GnuGo level 10. Table 5.4 summarizes the results. Unmodified Fuego won approximately 68.5% of games and Fuego using a single SPT won 73.6% of games. We tested the significance of these results using a chi-squared test with one degree of freedom. The test statistic, $\chi^2 = 12.054$, is significant at a p-value of 0.01, confirming that SPTs were able to significantly improve the performance of Fuego. Fuego using a forest of 5 SPTs won 74.2% of games, which is not significantly different from Fuego using a single SPT.

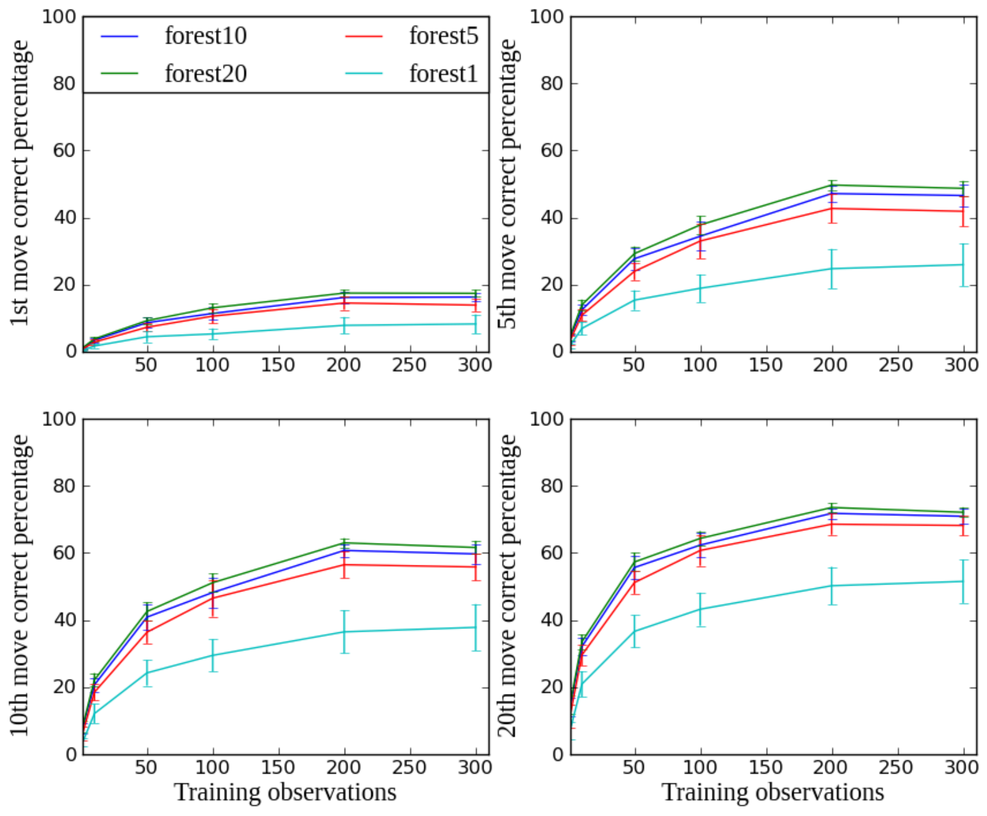

Figure 5.5: Percent of the time that the expert move is correctly predicted by the top 1, 5, 10 and 20th ranked moves as a function of the number of training observations.

Rank	SPT	Stern 2006	Araki 2007
1	14%	34%	34%
5	42%	66%	60%
10	56%	76%	70%
20	69%	86%	77%

Table 5.3: Cumulative prediction probability

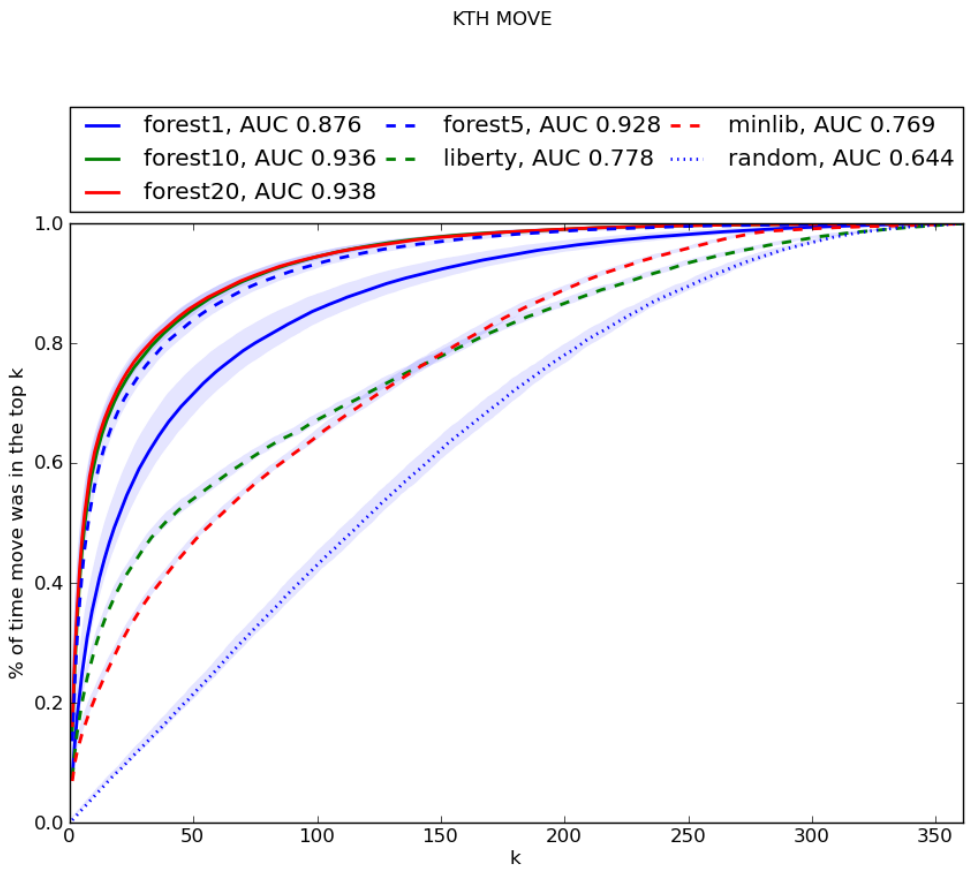

Figure 5.6: Kth move comparison against Fuego heuristics.

Go Player	Win Rate
Unmodified Fuego	68.5
Fuego w single SPT	73.6
Fuego w 5 SPT forest	74.2

Table 5.4: Win rate vs GnuGo version 3.6, level 10

Chapter 6

Conclusions and Future Work

We have introduced a new model for predicting events in spatial domains, the Spatial Probability Tree. An SPT is a tree with a conditional probability table at the leaf nodes that stores the probability of an event occurring at a single point in space, given the results of several questions asked with respect to that point.

We have applied SPT forests to the problem of predicting expert moves in the board game Go. We demonstrated that SPTs enable Fuego, one of the top computer Go players, to significantly improve its performance.

One of the downsides of integrating knowledge into a UCT player is that it slows down the search process. Although the evaluation of the SPTs is very efficient, evaluating hundreds of thousands of moves within a game adds to the complexity of any system. We are investigating how to best add a limited set of knowledge for timed competitions.

Although we applied the SPT only to computer Go, we are investigating the application to several very different domains. The SPT can be used to predict the location of anything that can be represented as a two-dimensional field, and as such could be useful in weather prediction, terrain classification, and character and image recognition, in addition to many other areas.

While the application of SPT to prediction tasks is relatively straightforward, its potential use for classification tasks bears explanation. Essentially, the SPT "colors" cells of a field. For Go, it assigns a probability to each location on the board. If it were given images, which are simply fields of pixels, it could mark pixels with the probability that this pixel or set of pixels is part of a face or letter. Individual trees could even be trained on specific faces or letters.

Another possible extension of SPT involves stacking trees to learn higher-level ideas. First, each of the leaf nodes of an SPT is uniquely numbered and the algorithm is run on a field. For each focus point in the field, the number of the leaf node to which each example went is recorded: these values provide a different discretely valued field which can be input to another SPT. It is essentially a domain transformation–the lower tree transforms information about the field into higher level objects, and the upper tree is trained on those higher level objects. These trees can be stacked indefinitely, though there is likely a limit to the depth at which the stacking is beneficial.

There are many tasks to which SPT might be applied; however, a few limitations need to be addressed. Currently, the SPT can only predict the location of a single item in space. In Go, for instance, it predicts a single next move. Future work could include expanding the SPT so that it can predict the location of any number of objects or objects larger than a single cell. The latter would be a useful extension for the character and image recognition problems.

Another limitation of the SPT is the requirement that the input field be discretely valued. Many real-world spatial prediction tasks are continuously valued; in order to use SPT on those tasks, they must be transformed to discrete values, and information is lost during that process. A modification of SPT that can predict in continuous value fields would therefore be useful.

Reference List

Araki, N., Yoshida, K., Tsuruoka, Y., and Tsujii, J. (2007). Move prediction in Go with the maximum entropy method. In *IEEE Symposium on Computational Intelligence and Games*, pages 189–195.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2-3):235–256.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Cazenave, T. and Helmstetter, B. (2005). B.: Combining tactical search and monte-carlo in the game of go. In *IEEE Symposium on Computational Intelligence and Games*, pages 171–175.

Coulom, R. (2007). Computing Elo ratings of move patterns in the game of Go. *International Computer Games Association Journal*, 30(4):198–208.

Dabney, W. and McGovern, A. (2007). Utile distinctions for relational reinforcement learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 738–743, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Drummond, C. and Holte, R. C. (2003). C4.5, class imbalance, and cost sensitivity why under-sampling beats over-sampling.

Enzenberger, M., Müller, M., Arneson, B., and Segal, R. (2010). An open-source framework for board games and Go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259 – 270.

Gelly, S. and Silver, D. (2008). Achieving master level play in 9 x 9 computer Go. In *Proceedings of the 23rd Conference on Artificial Intelligence, Nectar Track*, pages 1537–1540.

Hsu, F.-H. (2007). Cracking Go. *IEEE Spectrum*, 44(10):50–55.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *Machine Learning:*

ECML 2006, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin / Heidelberg. 10.1007/11871842_29.

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009). The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73 –89.

Michalowski, M., Brody, M., and Neilsen, M. (2011). Bayesian learning of generalized board positions for improved move prediction in computer go. In *Twenty-Fifth Conference on Artificial Intelligence*, pages 815–820.

Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134:145–179.

Rosin, C. (2010a). Multi-armed bandits with episode context. In *International Symposium on Artificial Intelligence and Mathematics*, page Electronically Published.

Rosin, C. (2010b). Personal communication.

Shubert, W. M. (2006). The KGS Go server.

Stern, D., Herbrich, R., and Graepel, T. (2006). Bayesian pattern ranking for move prediction in the game of Go. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 873–880.

Sutskever, I. and Nair, V. (2008). Mimicking Go experts with convolutional neural networks. In Kurkov, V., Neruda, R., and Koutnk, J., editors, *Artificial Neural Networks - ICANN 2008*, volume 5164 of *Lecture Notes in Computer Science*, pages 101–110. Springer Berlin / Heidelberg.

Werf, E. V. D., Uiterwijk, J., Postma, E., and Herik, J. V. D. (2003). Local move prediction in Go. In *In LNCS 2883: Computers and Games. Third International Conference*, pages 393–412.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd edition.

Wolf, T. (2000). Forward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122(1):59 – 76.