Efficient Single State Agents for Computer Go

David Flemström
davidfl@kth.se

April 29, 2014

Abstract

When it comes to Computer Go agents, the most common strategy is to traverse the game state tree to find as close to optimal solutions as possible. The goal is to sacrifice performance for a higher result accuracy.

Going in the other direction — sacrificing accuracy for performance — is a less explored approach to the problem.

In this report, a method combining the Copy-Move algorithm from the field of image forensics and computer vision with some pattern-based Computer Go digestion methods is tested against a large database of Go games. The partial soundness and fitness of the strategy is evaluated against a distinct subset of games by evaluating the predictability of human moves.

The result is that while pattern-based Go strategies are still clearly inferior to game-tree search algorithms, they sometimes yield promising results that can be used to adjust the pruning approaches typically employed in tree traversal algorithms.
## Contents

1 Introduction 4  
   1.1 The game 4  
   1.2 Previous work 4  
   1.3 Problem 5  

2 Background 6  
   2.1 The Copy-Move algorithm 6  
      2.1.1 General idea 6  
      2.1.2 Source data 7  
      2.1.3 Scalar classification 7  
      2.1.4 Digestion 7  
      2.1.5 Fingerprinting 8  
      2.1.6 Clustering 8  
   2.2 Efficient ordered observation storage 8  

3 Method 9  
   3.1 Source data 9  
      3.1.1 Win conditions 9  
      3.1.2 Ratings 10  
   3.2 Normalization 10  
   3.3 Fingerprinting 11  
   3.4 Platform 12  
   3.5 Technical implementation strategy 12  
   3.6 SGF format 13  
   3.7 Tree interpreter 13  
   3.8 Scoring 13  
   3.9 Vision index storage 14  
   3.10 Querying vision index 14  

4 Result 15  
   4.1 Partial database ingestion 15  
   4.2 Prediction speed 15  
   4.3 Database integrity 15  
   4.4 Prediction fitness 15
5 Discussion

5.1 Difficulty of treating Go as a single-state game .......................... 17
5.2 Hardware is still the bottleneck ............................................ 17
5.3 Optimization ................................................................. 17
5.4 Hybrid algorithms ......................................................... 17

A Source code

A.1 Normalized Fingerprinter .................................................. 20
A.2 SGF Parser ................................................................. 20
A.3 SGF Tree Interpreter ..................................................... 22
A.4 Index Storage ............................................................. 24
A.5 Database ingestion system .............................................. 25
1 Introduction

Deep Blue's victory over Kasparov in 1996 is both recognized as an important event in the history of computer science, and a turning point within the realm of artificial intelligence research. Shifting focus to games in general, the science community has recognized that there are interesting classes of problems within game theory beyond alpha-beta pruning search or similar methods, which were previously seen as one of the only approaches to game tree-based actor strategies.

In particular, the game of Go has evoked significant interest in recent times[11]. Being not only a popular board game world-wide, but also one of the typical games that cannot be approached with a naive tree search strategy, Go has laid the foundation for numerous research topics revolving around new approaches to game agent strategies.

1.1 The game

The game of Go is played on a board consisting of a grid of lines, typically with $19 \times 19$ intersections. Two players take turns placing respectively black and white stones onto the board, and simple rules govern when a move is legal or not, and whether a certain move will remove stones from the board. When both players agree that no more stones can be placed onto the board, the player with the greatest number of stones wins.

Because of the simplicity of the game, it is perhaps surprising that it should be the cause of such complexity in the realm of artificial intelligence, but the simplicity is also the source of its complexity.

1.2 Previous work

Some of the common approaches include using Monte-Carlo tree searching[6], using upper confidence trees[9, 13, 15], balancing simulations[10], parallel search[7], non-linear optimization and active learning[1], parallels to other games[14] or different heuristics[5].

Common to all of these approaches is that they generally boil down to performing searches on the game state tree. This idea seems to yield promising results, yet at the expense of the tremendous amount of computing power required for computing the traversed tree.

In general, research on game playing agents for computer Go has typically revolved around finding optimal strategies for winning the game, in search of a generalized “ideal strategy” that would beat any other agent. The purpose of such research is to fully understand the game and its complexities; as such, to gain insight into the mathematical structure behind the game. Beneficial side effects have also included advances in parallel and distributed computing to allow for the necessary and expensive computations to take place.

Despite the effort that has been invested into finding new strategies for Go, it still remains a game where a human can typically beat a computer consistently no matter
what method is used by the agent. This is especially interesting considering the fact that the computer often has an order of magnitude more time and multiple orders of magnitude more resources (in terms of energy) at its disposal.

1.3 Problem

The realm of game tree searching algorithms is a well-explored topic at this point, and while there still are significant advances to be made in the field, there are other approaches which yet remain less explored.

Especially interesting is considering in more detail how humans perform compared to computers when playing Go. Experience suggests that humans will typically still beat computers even when both agents are under extreme time restrictions. The reason why this would be the case is not immediately obvious; limiting the available time does also in turn limit the strategical options available to either party to more trivial ones, which would theoretically bring both parties on an even footing. Is this really the case?

In particular, an investigation of how agents with extreme or even real-time performance requirements would perform compared to games played by human actors would yield insight into how similar the approaches taken by either party really are.

It would be desirable to investigate whether a reasonable strategy for predicting Go moves can be found simply by analyzing a single game state, but constructing that strategy solely based on associations that can be derived from that game state. The agent is allowed to possess a significant amount of pre-computed “experience,” but may not traverse a game state tree, try out an exhaustive set of options, or otherwise do expensive calculations. The agent must also “make a move” within a very limited time frame; on commodity hardware amounting to a few milliseconds.
2 Background

The initial approach taken in this report is to employ algorithms from image forensics and computer vision in order to classify, index and predict Go game states.

2.1 The Copy-Move algorithm

Contrary to intuition, game state analysis shows some clear parallels to image forensics and computer vision data mining. In particular, the “Copy-Move” algorithm[2] is proven to be useful when detecting patterns and links between structured data.

The purpose of the algorithm is to find common patterns within a large space of data points, where the data points are arranged in a finite set of matrices. The algorithm identifies partial segments in the matrices and the closeness and centrality between them. Thus a graph is built that represents the distance between subsets of data points, and implicitly the distance between the matrices. It uses a very general method that is widely applicable.

This particular algorithm was chosen for its characteristics of requiring a great deal of up-front computation, but allowing conclusions to be drawn about the data set in a very efficient way.

In this section, the parallels between the subjects of image forensics and game state analysis related to the “Copy-Move” algorithm will be explored. The trade-offs that were considered for the analysis in this article are outlined as well. The algorithm itself can be decomposed into multiple independent steps, which is reflected by the sectioning below.

2.1.1 General idea

The goal of “Copy-Move” is to identify patterns/similarities between a large set of N-ary matrices. The general strategy is roughly as follows:

1. Identify the source data that is to be employed. In other words: How can the data be seen as a matrix?

2. Transform the data in such a way that data points are classified as scalars. In other words: How can each cell in each matrix be assigned a number?

3. Digest the scalar representation to highlight the trait that is assumed to display the pattern that is being identified. In other words: How can each cell be represented such that one can measure the amplitude of something that might exhibit a pattern?

4. Fingerprint the digested data so that cells are contextualized and can be lexicographically ordered. In other words: How can each cell's digest be ordered in such a way that similar patterns are close to each other?

5. Cluster fingerprints, such that when a particular set of fingerprints are selected, they can be grouped in such a way that each group points to one or many of
the source matrices that were used to construct those fingerprints. In other words: Given some fingerprints, is there a way to determine whether a significant group of fingerprints exhibit the same pattern?

6. **Efficiently store** fingerprints, so that ordered look-ups of fingerprints is efficient. In other words: Given one fingerprint, how is it possible to quickly find similar fingerprints?

### 2.1.2 Source data

The "Copy-Move" algorithm acts on any data that is structured in an N-ary matrix. In particular, it is heavily explored as a tool for analyzing 2D matrices. The data must be formulated as a set of (sparse or dense) N-ary matrices. The matrices need not be the same size.

Typically, in the case of image forensics, an image consists of a 2D matrix of “pixels,” and typically each image has only one such associated matrix. Thus each matrix can be grouped by the image that it belongs to.

In the case of Go, each Go game consists of a number of game states, representing the contents of the board after each move made by a player during the game. A game state is a matrix consisting of positions, each position being one of $19 \times 19$ intersections on the board. Each position can either be occupied by a white, black, or no stone.

### 2.1.3 Scalar classification

Each datum in the matrix needs to be classified by a scalar, or some other attribute that produces an ordinal mapping for the particular datum. This is to allow one to absolutely order samples from the matrix.

Typically, in the case of images, one would convert the image to gray-scale in order to map each pixel in the image to a scalar.

In the case of Go, some form of normalizing operator is needed to classify each board position with a scalar, since the game's semantics do not yield an inherent ordinal for board positions. There are many ways of doing this, depending on the required information. The particular methods that were explored for this analysis are outlined further below.

### 2.1.4 Digestion

If the data that is analyzed exhibits a high variance or noise ratio, it may be necessary to digest the data using some well-known digestion algorithm, that brings the data into a representation that eliminates the noise and brings forward some characteristic to be analyzed.

For image data, usually a discrete cosine transform (DCT) is employed. Such a transform is useful because the parameters in the resulting tensor are structured in such a way that top/left-most cells describe the “coarse” structure of the initial data set, while the remaining cells describe “finer” details of the data set. This allows DCTs to
be ordered in such a way that coarse data differences dominate the ordering. A sliding window of a fixed size is used to fold the matrix into a four-dimensional tensor, mapping each former pixel to a DCT matrix.

An alternative to DCT is to use (fast) Fourier transforms. The disadvantage of using this method is that the resulting continuous histogram has no absolute ordering, and needs to be sampled to determine some canonical digestion.

The simplest digestion method is the identity digestion, which leaves the source matrix unchanged. Depending on the level of detail exhibited by the source data, this might be sufficient to allow one to draw some conclusions about the data. This is especially useful if the values in the source matrix are discrete, since such data does not lend itself well to being transformed by DCT or FFT.

2.1.5 Fingerprinting

The matrices are fingerprinted, meaning that every cell in the matrix is classified by its context in the matrix.

The simplest fingerprint is to classify each cell as itself. This yields results where the distance between matrices effectively is the magnitude of the vector between the histograms of the matrices.

For a DCT, it can simply be unwound from most significant cell to least significant cell. For other kinds of data, different fingerprinting methods can be used. One such alternative is to sample cells in a specific order. The fingerprints need to be strictly ordered, typically lexicographically.

2.1.6 Clustering

Ordering the fingerprints of the classified objects yields a strict distance estimation between the objects. They can then be clustered according to some criterium.

2.2 Efficient ordered observation storage

Storing a large amount of ordered data is not trivial, especially when the number of data points is in the billions.

By using an approach where storage is “sharded” on a granular level and strictly ordered on a lower level yields predictable behavior. Fingerprints can be sorted into rough groups by using a prefix of their descriptors as a sharding hint. When querying the set, only fingerprints in the same sharding group need to be compared. With a sufficiently large set of fingerprints, it does not matter that the fingerprints are only ordered within each shard, since it is very likely that an arbitrarily chosen fingerprint will be in the middle of a shard rather than close to the edge of one.

This is already a solved problem, and a widely used solution such as Apache Cassandra is suitable for the storage model.
3 Method

The general approach to solving this problem can be summarized as follows:

1. A large database of Go games is acquired and curated.
2. The database is ingested and normalized to a context-free representation.
3. Each game state where an agent makes a move is characterized and a “fingerprint” is computed that describes each board position.
4. The fingerprints are stored in a distributed data store.
5. The above process is massively parallelized using map-reduce concepts to build a knowledge index faster.
6. A query engine is constructed that evaluates a user-provided game state and uses computer vision to find the suggested next move.

3.1 Source data

In order to do any analysis of Go games, a large database of games needs to be acquired. In particular, games that have very constrained move times are ideal for this purpose. For this evaluation, the online service called “The KGS Go Server” seems to be the most suitable, because the games are typically short, with players from all skill levels, and the database is vast.

All games between 2001 and 2013 are used for this purpose, totaling 157284 games. Of these, 67809 were won by black and 87457 were won by white, 6 games were invalid and 2 were draws.

3.1.1 Win conditions

Where games were decided, 89249 games were won by resignation, 28133 by time limits, 110 by one player forfeiting the game, and the rest by finishing the game until no more moves were possible.

<table>
<thead>
<tr>
<th>Win condition</th>
<th>Black</th>
<th>White</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forfeit</td>
<td>18</td>
<td>92</td>
</tr>
<tr>
<td>Resign</td>
<td>39736</td>
<td>49513</td>
</tr>
<tr>
<td>Time</td>
<td>10422</td>
<td>17711</td>
</tr>
<tr>
<td>Score</td>
<td>17633</td>
<td>20141</td>
</tr>
</tbody>
</table>

The finishing scores were between 0.50 and 8194.00, with the median being 9.00, the mean being 13.93, the first quartile being 4.50 and the third quartile being 16.50. The standard deviation of the set of scores is 104.3111.
3.1.2 Ratings

Players are generally highly rated. This minimizes the problem that new players win by the opponent making mistakes and not by skill. Players are counted once per game participated in, so this count is not for distinct players. The “d” stands for “dan” (lit. Japanese: degree) and players are ranked from 1d (lowest advanced amateur rank) to 9d (highest advanced amateur rank). Traditionally, the “dan” ranking system only exhibits 8 levels, but a 9th level was used in this particular database, for reasons unknown.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Number of players</th>
</tr>
</thead>
<tbody>
<tr>
<td>9d</td>
<td>16524</td>
</tr>
<tr>
<td>8d</td>
<td>37639</td>
</tr>
<tr>
<td>7d</td>
<td>104982</td>
</tr>
<tr>
<td>6d</td>
<td>101221</td>
</tr>
<tr>
<td>5d</td>
<td>19279</td>
</tr>
<tr>
<td>4d</td>
<td>11673</td>
</tr>
<tr>
<td>Below</td>
<td>18409</td>
</tr>
</tbody>
</table>

3.2 Normalization

An algorithm that digests a game state into a desirable lesser description more suitable for quick look-up has been designed. A pattern-based recognition algorithm has been constructed based on previous work in similar areas[12, 3], simplified to accommodate the approach of this article.

Each analyzed game consists of a set of states. A state consists of the current placed stones on the board along with their positions, and the position where the next player will place a stone.

During normalization, each position in a state is first classified according to some criteria. The criteria are listed below along with a set of states each criterium may use to classify a position.

1. The color that already occupies a position.
   (Black, White, n/a)
2. The colors that can legally put a stone at a position without violating any of the NoGo rules, which means any position that does not lead to the suicide/capture of one of the player’s groups[12].
   This is determined by doing a simple BFS flood fill of the area around the position. The rules of NoGo further state that Ko has to be respected as well, meaning that no previously seen game state may be repeated in any particular game, but since these game states originate from real games that were verified by a computer, the rule of Ko is never violated.
   This criterium is of limited usefulness, since NoGo is semantically different from Go, and not just a reduced problem.
   (Black, White, Black + White, n/a)
3. The territory that the unclaimed position belongs to. The territory of a position is determined by the color of the stone that is closest to the position. If there is a tie, the position will not belong to any territory. (Black, White, n/a)[3].

4. The morphological group that the position belongs to.

Let $B$ be all of the black positions on the board, and $W$ be all of the white positions. The morphological groups of these sets are simply $X(B)$ and $X(W)$, the connected sets of the groups. “Outsideness” is determined by whether both nodes for an edge in a morphological group have the same color[12].

(Outside Black, Inside Black, Outside White, Inside White, n/a)

The necessity of all of these classifiers diminishes with the size of the fingerprint database. Indeed; with a database of a significant size, only classifier 1 is needed to get good results for queries.

In the most detailed case, each classifier having 3, 4, 3 and 5 possible states, respectively, it means that each position is mapped to one of 180 possible states. This is too much entropy for the “Copy-Move” algorithm, and without a DCT transform (which is impractical for discrete data as described above) or some similar digestion method, the data becomes too noisy to use. It is therefore more practical to use only a few of these classifiers at a time.

The game state is further transformed such that the colors of stones are separated into “Ours” and “Theirs,” the former being the stones of the player who has the next move.

One final normalization step replicates the game state four times, each time rotating the game state by 90 degrees. Since Go does not define semantics for orientation, any game state is isomorphic with its four rotations. Each game does thus map to 4 normalized states.

### 3.3 Fingerprinting

For a particular normalized game state $N$, there is one fingerprint per position in $N$. Typically, this yields $19 \times 19$ fingerprints per game state.

Each fingerprint is calculated with a very simple process. The process has to be simple in order to be performant, since this classification process represents the bottleneck of the analysis process.

For each position, a number of relative positions are “tapped” (meaning that the classifier value — one of the up to 180 values as outlined above — is read for the cell at that relative position) in a certain order. This order is defined as:

$$\text{taps} = [(0, 0), (-1, 0), (0, -1), (1, 0), (0, 1),$$
$$(-1, 1), (-2, 0), (-1, -1), (0, -2),$$
$$(1, -1), (2, 0), (1, 1), (0, 2), (-1, 2),$$
$$(2, 1), (-3, 0), (-2, -1), (-1, -2),$$
$$(0, -3), (1, -2), (2, -1), (3, 0), (2, 1),$$
$$(1, 2), (0, 3), (-1, 3), (-2, 2)]$$

(1)
When drawn with pen and paper on an euclidean coordinate system, it becomes clear that this sequence of taps forms an arbitrary spiral of positions. The order of the taps is not significant, as long as the magnitudes of the coordinates are in semi-increasing order. The extra noise introduced by the arbitrary ordering (since the spiral is not a Fibonacci spiral or some similarly strictly growing spiral) helps improve query results by blurring the significance of each new element in the spiral.

For an example of how this strategy works, see listing A.1. For the sake of brevity, this only shows the most trivial fingerprinting strategy, where the classifiers being read adhere to classification rule (1) above.

### 3.4 Platform

The program code for the indexing system is written in Haskell. The declarative nature of this language makes creating massively parallel programs very easy. For optimal performance, a more low-level language could have been used, but seeing as this program will be IO-bound, it is not an important decision to make.

For search index storage, Apache Cassandra was used. This database engine supports massively parallel indexing and distributed storage. It is also optimal for doing fingerprint range queries.

The version of GHC is 7.6.3, and the version of Cassandra is 2.0.6.

### 3.5 Technical implementation strategy

There are a total of 157284 games in the database that is used to build the learning model. In these games, a total of 14253463 moves were made, which means that there are 14253463 game states to analyze. Each state contains 361 fingerprinted positions, which yields in total 5145500143 fingerprints to be built. Each fingerprint is also computed for 4 different orientations, and for two different players, but these scenarios play a lesser role, even though they impact the number of data points by another order of magnitude.

This is a hopelessly large number, and even if the program could run on very many computers with a significant amount of processing power, it would still take a very long time to process all of the game states.

Therefore, an adaptive model is used to incrementally improve the intelligence of the system. The system is functional with partial indexes of games, and indexing can occur at the same time as queries.

To build the vision database, a cluster of indexing nodes is used, each with a set of games to analyze. The Map-Reduce model is used for indexing. Each node parses a game, breaks it down into game states (map), normalizes states and picks the ones that are deemed suitable for ingestion because they were part of a won game and did not violate any of the rule constraints (reduce). The node will then compute the fingerprints for a game state and order them (map) and store them in the fingerprint cluster for querying (reduce).

A single-process implementation of this can be seen in the simple database building system in listing A.5.
3.6 SGF format

The game database is in the SGF format. A combinator/coroutine-based parser as in listing A.2 for this format is used to stream partial game trees to the rest of the analyser pipeline. To maintain performance, the parser is suspendable and can yield partial results in real-time.

Typically, an SGF database is used to describe a set of game trees. The database of Gō games used in this case however does not describe game trees. Instead, each SGF file describes a single game, and there is one tree per game in the database.

Some simplifications have been made taking this into consideration. While the parser conforms to the BNF description of SGF[4], it is assumed that each node in the SGF tree will have only one child.

3.7 Tree interpreter

SGF game trees produced by the parser are interpreted so as to get a set of game states describing the game at hand, as can be seen in listing A.3. This is done by simulating a game board and performing the moves described in the game tree.

Metadata is also extracted during this process. Different metadata is used when scoring search results while doing queries.

3.8 Scoring

Each game state is assigned a score. This is done for the purpose of ordering clustered groups after “Copy-Move” clusters of game states have been identified. The scoring method works like this:

1. If the game ended in a draw, was aborted, was illegal or had an unknown outcome, all game states in the game get a score of 0.
2. If a winner could be determined, all of the game states where the winner made a move are awarded a positive score, and all of the game states where the loser made a move are awarded a negative score. The absolute value of the score is determined by:
   - Resignation The score value is 10, which is close to the mean score by exhaustion.
   - Time-out The score value is 5, which is half of the mean score by exhaustion.
   - Forfeit The score value is 2, with low impact.
   - Exhaustive game The score value is whatever score the remaining stones on the board dictated.
3.9 Vision index storage

The fingerprints are stored in a lexicographical ordering in a Cassandra database, using the code in listing A.4. The storage is very straight-forward, but the data model and semantics are non-trivial. The data model being used can be described using the following Cassandra Thrift statements:

```csharp
create keyspace go
  with placement_strategy =
      'org.apache.cassandra.locator.SimpleStrategy'
  and strategy_options = {
      replication_factor : 1};

create column family go
  with comparator = UTF8Type
  and key_validation_class = UTF8Type
  and default_validation_class = BytesType;
```

A fingerprint is converted into a series of human-readable characters. The first two characters determine the shard that the fingerprint will be stored in. The whole fingerprint is stored in that shard in lexicographical order. Along with the fingerprint is some metadata describing whence the fingerprint came and what score it was assigned.

3.10 Querying vision index

A query into the vision index is a game state. The objective of the query is to predict what the next game move of that state will be.

The query process follows similar steps as the ingestion process, and the relevant description for each step can be seen above:

1. The game state is digested into fingerprints.
2. One query is made into the vision index for each fingerprint.
3. A range of similar fingerprints, which are the \( n \) fingerprints around the queried fingerprints, are selected. This yields \( 19 \times 19 \times n \) fingerprints.
4. Fingerprints are grouped by source game ID and implicitly by spatial proximity and scored by adding their game state scores.
5. The cluster of fingerprints with the highest collective score corresponds to a game state where the wanted move was performed. The other clusters are ordered by decreasing compound score.
6. The relative location of the move compared to the cluster is computed.
7. The moves are reported in score order to the user.
4 Result

The database that was outlined above is partially ingested into a vision index. Queries have successfully been made against the index. It was possible to make significant observations on the data using these queries.

4.1 Partial database ingestion

It is important to note that it was not possible to ingest the entire game database into an index, due to time and hardware constraints. Out of the roughly 10–20 billion fingerprints that could theoretically be built from the database, only about 40 million were ingested. This means that the index only knew of about 0.2% of the potential scenarios it could have known about.

4.2 Prediction speed

The index responds to lookups within an average of 15.09 microseconds on the same machine, and since $19 \times 19 = 361$ lookups are required to predict a move, about 5.41 milliseconds are needed for each prediction.

These statistics were taken from Cassandra's internal metrics, using 40630902 read samples.

4.3 Database integrity

When querying for a state from a game that had been ingested verbatim into the search index, the result was exactly the next state from that game, or, if the state was from an early move, exact copies of that state from higher ranked games, in 19638 out of 20000 test cases. In the other states, a move was chosen which was clearly illegal. The error can probably be attributed to sometimes overwriting fingerprints in a race condition. During ingestion, no care was taken to prevent overwrites (such as requesting quorum consistency from Cassandra), and this leads to the eventual integrity of the index sometimes erasing fingerprints, overwriting a “highly scored” fingerprint with a “lower scored” one.

4.4 Prediction fitness

When querying for game states from games that had not previously been indexed in the database, there is only one metric which can be tracked: Whether the predicted move corresponds to what the player did next in the actual game. Other fitness indicators seem to be too unreliable to be practical. This one metric was correlated to which generation the game state was sampled from.

A total of 20000 games were investigated. From these, the first 10 generations were queried.
<table>
<thead>
<tr>
<th>Generation</th>
<th>Prediction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.26%</td>
</tr>
<tr>
<td>2</td>
<td>8.72%</td>
</tr>
<tr>
<td>3</td>
<td>12.93%</td>
</tr>
<tr>
<td>4</td>
<td>9.05%</td>
</tr>
<tr>
<td>5</td>
<td>3.43%</td>
</tr>
<tr>
<td>6</td>
<td>1.88%</td>
</tr>
<tr>
<td>7</td>
<td>0.60%</td>
</tr>
<tr>
<td>8</td>
<td>0.96%</td>
</tr>
<tr>
<td>9</td>
<td>0.34%</td>
</tr>
<tr>
<td>10</td>
<td>0.31%</td>
</tr>
</tbody>
</table>

This result is to be expected. The numbers have the following potential explanations:

1. The first move can hardly ever be predicted, as querying for the empty game state yielded the same hit every time (And every Go game does usually not start with the same move). The surprisingly high number of successful predictions can still be explained by the fact that many games in the database started with “pre-placed” stones for a handicap (for example 5 black stones in a pattern), and the moves made in relation to those stones seemed to be predictable.

2. The following few moves are usually part of standard openings, so the high prediction rate reflects that. Even then, there are multiple variations so that the prediction was incorrect much of the time.

3. The remaining moves (6 and beyond) are generally unpredictable. The numbers are still higher compared to random selection, which would yield a 0.2% success rate.

One has to keep in mind that this metric is very unreliable, as the suggested move is compared to what an user actually did. The computer might have chosen a better move than the player; this metric does not reflect that.
5 Discussion

The method employed in this analysis had some known flaws from the start. However, additional insights could also be inferred from the outcome of the experiment.

5.1 Difficulty of treating Go as a single-state game

Judging by the diminishing prediction fitness numbers as presented in the results section, a strategy which looks at a single game state in Go is not going to be successful against a strategy which does an in-depth game tree traversal.

This was something which was known from the start, but quantifying the success rate of the method gave some insight into exactly how unreliable it truly was.

5.2 Hardware is still the bottleneck

It was not practical to compute all of the fingerprints in the game database. Given that only about 0.2% of the database was ingested, it was surprising that the results were as positive as they turned out to be. With more nodes and more hardware, the predictions might have become significantly better.

5.3 Optimization

The approach offered here is a naive attempt, both in theory and in practice. The method is by no means fully refined or optimal in any way. Perhaps it will be possible to one day refine the method so that fingerprints can be coalesced, and predictions can take small game state trees into consideration.

5.4 Hybrid algorithms

Combining pattern search methods and game tree search methods might also yield interesting results. This approach of using computer vision to find interesting move candidates would in some sense replace some of the pruning models used in tree search algorithms.
References


A  Source code

Here are some listings of essential parts of the source code for this project. Non-essential source code has been omitted to save space and paper.

A.1  Normalized Fingerprinter

module  AI . Go . Fingerprint . Normalized  where

import  AI . Go . Fingerprint
import  AI . Go . Normalized
import  Control . DeepSeq
import  Control . Monad ( forever , forM_ , when )
import  Data . Matrix
import  Pipes

normalizedFingerprint :: Monad m
=> Pipe ( NormState NormPosition ) Fingerprint m ( )

normalizedFingerprint = forever $ do
  state <- await
  let board = _nsBoard $ state
  ( (lx , ly ) , (hx , hy ) ) = ( (1 , 1) , (ncols board , nrows board ) )
  when ( _nsNextTurn state == Self ) $
    forM_ $ [ ( x , y ) | x <- [lx .. hx ] , y <- [ly .. hy ] ] $ \ ( x , y ) -> do
      let positions = flip map taps $ \ ( dx , dy ) ->
        if x + dx <= hx && x + dx >= lx &&
        y + dy <= hy && y + dy >= ly
        then case _nsBoard state ! ( x + dx , y + dy ) of
          Current c -> Stone c
          _ -> Empty
        else Outside
    yield $ positions 'deepseq' Fingerprint
    ( FingerprintMetadata ( _nsGameId state )
    x y ( _nsScore state ) ) positions

where
taps = [ ( 0 , 0) , ( -1 , 0) , ( 0 , -1) , ( 1 , 0) , ( 0 , 1) ,
        ( -1, 1) , ( -2, 0) , ( -1, -1) , ( 0, -2) ,
        ( 1, -1) , ( 2, 0) , ( 1, 1) , ( 0, 2) , ( -1, 2) ,
        ( -2, 1) , ( -3, 0) , ( -2, -1) , ( -1, -2) ,
        ( 0, -3) , ( 1, -2) , ( 2, -1) , ( 3, 0) , ( 2, 1) ,
        ( 1, 2) , ( 0, 3) , ( -1, 3) , ( -2, 2) ]

A.2  SGF Parser
{-# LANGUAGE OverloadedStrings #-}
module AI.Go.SGF.Parser where

import Prelude hiding (sequence, takeWhile)
import AI.Go.SGF hiding (sequence)
import Control.Applicative (pure, (>), (<*), (<**))
import Data.Attoparsec.Text
import qualified Data.HashMap.Lazy as HashMap (fromList)
import Data.Functor ((<$>))
import Data.Text (Text)

sgfSingleFile :: Parser SGF
sgfSingleFile = skipSpace *> sgfSingle <*> skipSpace

sgfSingle :: Parser SGF
sgfSingle = SGF . pure <$> gameTree

sgfFile :: Parser SGF
sgfFile = skipSpace *> sgf <*> skipSpace

sgf :: Parser SGF
sgf = SGF <$> gameTree 'sepBy1' skipSpace

gameTree :: Parser GameTree
gameTree =
  <$> (char '(' *> skipSpace *> sequence <*> skipSpace)
  <*> ((gameTree 'sepBy' skipSpace) <*> skipSpace <*> char '))

sequence :: Parser Sequence
sequence = Sequence <$> node 'sepBy1' skipSpace

node :: Parser Node
node =
  Node . HashMap.fromList
  <$> (char ':' *> skipSpace *> property 'sepBy1' skipSpace)

property :: Parser (Text, [Text])
property =
  <$> propIdent <*> skipSpace
  <*> propValue 'sepBy1' skipSpace

propIdent :: Parser Text
propIdent = takeWhile1 $ inClass "a-zA-Z"

propValue :: Parser Text
propValue = char ' [' => takeWhile (/= ' ] ') <* char ' ] ' 

A.3 SGF Tree Interpreter

{-# LANGUAGE OverloadedStrings #-}  
{-# LANGUAGE RankNTypes #-}  
module AI.Go.Game.SGF where

import Prelude hiding (sequence)
import AI.Go.Game
import AI.Go.SGF
import Control.Lens
import Control.Monad hiding (sequence)
import Control.Monad.Trans.State.Strict
import Data.Matrix
import Data.Char (ord)
import Data.Default (def)
import Data.Functor ((<$>))
import Data.Text (Text)
import qualified Data.Text as Text
import Pipes hiding (each)

type GameBuilder m = StateT Game m

sgfToGame :: (Functor m, Monad m) => Pipe SGF Game m ()
sgfToGame = void . flip execStateT 0 . forever $ do
  sgf <- lift await
  convertSGF sgf

convertSGF :: (Functor m, Monad m) => SGF -> StateT Integer (Proxy x ' x () Game m) ()
convertSGF sgf = forMOf_ (trees . each) sgf $ \ tree -> do
  g <- get
  modify (+1)
  game <- execStateT (buildFromTree g tree) def
  lift $ yield game

buildFromTree :: (Functor m, Monad m) => Integer -> GameTree -> GameBuilder m ()
buildFromTree g tree = do
  gid .= g
  forMOf_ (sequence . nodes . each) tree $ \ node -> do
    iforM_ (view properties node) buildFromProperty

buildFromProperty :: (Functor m, Monad m)
GameBuilder m ()

buildFromProperty "SZ" [ sz ] =
  size ?= (read . Text.unpack $ sz)
buildFromProperty "PW" [pw] =
  whiteName ?= pw
buildFromProperty "PB" [pb] =
  blackName ?= pb
buildFromProperty "WR" [wr] =
  whiteRating ?= (read . init . Text.unpack $ wr)
buildFromProperty "BR" [br] =
  blackRating ?= (read . init . Text.unpack $ br)
buildFromProperty "RE" [ r ] =
  winCondition ?= case Text.unpack r of
    "Void" -> Invalid
    "0" -> Draw
    "?" -> Unknown
    'B' : '+' : score -> Winner Black $ parseScore score
    'W' : '+' : score -> Winner White $ parseScore score
    other -> error $ "Result not recognized: " ++ other
  where
    parseScore "R" = Resign
    parseScore "Resign" = Resign
    parseScore "T" = Time
    parseScore "Time" = Time
    parseScore "F" = Forfeit
    parseScore "Forfeit" = Forfeit
    parseScore num = Score $ read num
buildFromProperty "AW" moves =
  startMoves <=>
    map (Move White . parseCoord . Text.unpack) moves
buildFromProperty "AB" moves =
  startMoves <=>
    map (Move Black . parseCoord . Text.unpack) moves
buildFromProperty "W" [move] =
  performMove . Move White . parseCoord . Text.unpack $ move
buildFromProperty "B" [move] =
  performMove . Move Black . parseCoord . Text.unpack $ move
buildFromProperty _ _ = return ()

performMove :: (Functor m, Monad m)
  => Move -> GameBuilder m ()
performMove (Move bw cd) = do
  st <- use states
  when (null st) $ do
    s <- maybe 19 id <$> use size
    states .=
      [GameState Black . matrix s s $ const Nothing]
  ms <- use startMoves
  forM_ ms $ \ (Move bw' cd') -> case cd' of
    Coord r c ->

23
states.last.board %= forceMatrix.setElem (Just bw) (r, c)
Pass -> return []
s <- use $ states.singular_last
let newState = case cd of
  Coord r c ->
    s & board %= setElem (Just bw) (r, c)
    & nextTurn = invert (view nextTurn s)
  Pass -> s
invert Black = White
invert White = Black

-- TODO: use better data structure for append
states <= [newState]
parseCoord :: String -> Coord
parseCoord "tt" = Pass
parseCoord [a, b] =
  Coord (ord a - ord 'a' + 1) (ord b - ord 'a' + 1)
parseCoord "" = Pass
parseCoord m = error $ "Unrecognized move." ++ m

A.4 Index Storage

module AI.Go.Fingerprint.Store where

import AI.Go.Normalized
import AI.Go.Fingerprint

import Control.Monad (forever)
import Data.Binary (encode)
import Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString
import Database.Cassandra.Basic
import Pipes

storeCassandra :: (MonadCassandra m) => Consumer Fingerprint m ()
storeCassandra = do
  pool <- lift getCassandraPool
  forever $ do
    Fingerprint meta poss <- await
    let description = describe poss
    liftIO . runCas pool $ insert "go" (ByteString.take 2 description)
    ONE [packCol (description, encode meta)]
describe :: [FingerprintPosition] -> ByteString
describe =
  ByteString.pack . map characterize
  where
    characterize (Stone Self) = 83
    characterize (Stone Other) = 79
    characterize Empty = 32
    characterize Outside = 64

A.5  Database ingestion system

{-# LANGUAGE OverloadedStrings #-}
module Main (main) where

import AI.Go.Fingerprint.Normalized (normalizedFingerprint)
import AI.Go.Fingerprint.Store (storeCassandra)
import AI.Go.Game.SGF (sgfToGame)
import AI.Go.SGF.Parser (sgfSingleFile)
import AI.Go.Normalized.Game (gameToNormalized)

import Control.Monad (void, forM, forever)
import Control.Monad.Trans.State
import Control.Concurrent.Async
import Database.Cassandra.Basic hiding (get)

import Pipes
import Pipes.Attoparsec
import Pipes.Concurrent
import Pipes.Text.IO (stdin)

main :: IO ()
main = do
  pool <- createCassandraPool [defServer] 4 16 5 "go"
  (output, input) <- spawn $ Bounded 65536
  as <- forM [1..(4 * 16)] :: Integer $ \ i -> async $ do
    putStrLn $ "Starting_Cassandra_worker_" ++ show i
    runCas pool . runEffect $ fromInput input >>= storeCassandra
    putStrLn $ "Stopping_Cassandra_worker_" ++ show i
    performGC
  a <- async $ do
    putStrLn "Starting_game_state_producer"
    runEffect $ void (parsed sgfSingleFile stdin)
      >>= sgfToGame
      >>= gameToNormalized
>> pv
>> normalizedFingerprint
>> toOutput output

putStrLn "Stopping game state producer"
performGC

mapM_ wait (a:as)

pv :: Pipe a a IO ()
pv = void . flip execStateT (0 :: Integer) . forever $ do
a <- lift await
i <- get
modify (+1)
liftIO . putStrLn $ "Processed" ++ show i
lift $ yield a