Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 go

Bruno Bouzy
Université Paris 5, C.R.I.P.5
45, rue des Saints-Pères 75270 Paris Cedex 06 France
tél: (33) (0)1 44 55 35 58, fax: (33) (0)1 44 55 35 35
email: bouzy@math-info.univ-paris5.fr

Guillaume Chaslot
Ecole Centrale de Lille
Cité Scientifique - BP 48, 59651 Villeneuve d'Ascq Cedex
email: chaslot.guillaume@ec-lille.fr

Abstract- This paper describes the generation and utilisation of a pattern database for 19x19 go with the K-nearest-neighbor representation. Patterns are generated by browsing recorded games of professional players. Meanwhile, their matching and playing probabilities are estimated. The database created is then integrated into an existing go program, INDIGO, either as an opening book or as an enrichment of other pre-existing hand-crafted databases used by INDIGO move generator. The improvement brought about by the use of this pattern database is estimated at 15 points on average, which is significant on go standards.

1 Introduction

Because the branching factor and the game length forbid global tree search in go, and because evaluating non terminal go positions is hard [14], computer go remains a difficult task for computer science [15, 13]. In addition, computer go is an appropriate testbed for AI methods [8]. INDIGO [7] is made up of the Monte Carlo (MC) module and the knowledge module. The MC module has been described recently [9, 4], and the knowledge module was described before 2003 [8, 5, 6]. To briefly present the current INDIGO move decision process, the knowledge module provides the MC module with ns moves, and, in order to select the best move, the MC module plays out a lot of complete random games starting with these moves and computes mean values.

The knowledge module includes various pattern databases built manually. Hand-crafted databases have many downsides: they contain errors, they have holes, and they cannot be easily updated. Furthermore, the various pattern bases in INDIGO do not share the same format: the first one (FORME_M) includes domain-dependent features used by the conceptual evaluation function, the second one (FORME_3X3) contains 3x3 patterns optimized for fast simulations, and the last one is dedicated to large patterns useful at the beginning of the game (FORME_B and FORME_C). Due to the success of the MC module within INDIGO, we aimed at using statistics in the knowledge module too. Thus, it was the right time to test the automatic creation of a new pattern database and observe its positive effects within the INDIGO architecture. The automatic creation of patterns avoids errors and holes in the database. The automatic creation is performed by browsing recorded professional games to create patterns and to estimate both their matching probabilities and their playing probabilities when matching. In other words, the approach we adopted is a bayesian approach.

In order to avoid any limitation due to the size of patterns, particularly at the beginning of the game, we used the K-nearest-neighbor representation in which the relevant neighbors are the occupied intersections and the edges. For this reason, this database is named FORME_K.

Section 2 is a summary of works related to the current paper. Section 3 defines the K-nearest-neighbor representation used. Then, section 4 describes the creation of patterns and their probabilistic features. Section 5 underlines the experiments performed to integrate this work within INDIGO, and assesses the improvements. Before conclusion, some interesting perspectives are highlighted by section 6.

2 Related work

Despite of its importance within go programs, the litterature about pattern acquisition, local move generation or recorded professional games is not very abundant. [2] by Mark Boon was the first paper to describe a pattern-matcher in great details: the 5x5 window pattern-matching algorithm of Goliath, best program in 1990. But this paper did not deal with the pattern acquisition. Recently, Erik van der Werf described a neural network approach using professional recorded games to generate local moves [17], predict life and death [19], or score final positions [18]. Since it also browses recorded games to produce local moves, the current work is similar to Erik van der Werf's approach, but it is less sophisticated because it uses the K-nearest-neighbor representation instead of a neural network. Moreover, it is not intended to predict life and death or to score positions. Tristan Cazenave worked on automatic acquisition of tactical patterns for eyes or connections [10], even including liberties [11]. The current work is similar to Cazenave's work because it consists in automatic acquisition of patterns but it is quite different because Cazenave's patterns were generated in a specific tactical context: connecting or making eyes, by using explanation-based learning. Finally, [3] was an attempt to generate 4x4 patterns by retrograde analysis. Although it dealt with automatic acquisition of patterns, this work was completely different from the present work because it was limited to small boards. Furthermore, it did not use the bayesian approach but retrograde analysis.

3 K-nearest-neighbor representation

The K-nearest-neighbor representation is common in pattern recognition [1]. This section defines the K-nearest-neighbor representation used in this work.

3.1 K-nearest-neighbor patterns

The picture below shows an example a K-nearest-neighbor pattern.

```
  +@+
 +++++
 ++*+O
 +++++
  ++@
```

The pattern always advises to move *in its center* marked by a '*'. '+' represents an empty intersection. 'O' represents a white stone. '@' represents a black stone. '+' is an *unimportant* fact in this representation. Conversely, a black or white stone, an edge or a corner are *important* facts. A pattern contains a number of important facts, named K. In the example above, K=3.

The center of the pattern being given, we assume the neighboring intersections are ordered according to a distance. Moreover, we assume that this pre-defined order avoids ties between intersections situated at the same distance from the center of the pattern. With such assumption, the pattern matching principle remains simple and can be programed efficiently.

The upside of this representation lies in the lack of limitation on the size of the patterns. In go, many moves are played in the neighborhood of stones and edges. To simplify the work we constrained the patterns to advise one move in its center only, and not elsewhere.

The K-nearest-neighbor representation does not explicitly contain "don't care" points usually managed by go patterns [2]. However, in other representations managing "don't care" points, the patterns are roughly centered around the move played, and the "don't care" points are often situated far from the center of the patterns, while the crucial points are situated near from the center of the patterns. Therefore, we may say that the K-nearest-neighbor representation implicitly contains don't care points. Besides, not managing these points explicitly simplifies the pattern matching algorithm. Moreover, because replacing a pattern containing a "don't care" point by four patterns containing one of the four explicit values (black, white, empty, edge) is still possible, this representation does not lose generality provided that the memory space is sufficient.

Pattern-matching must deal with the symmetries, rotations and black and white inversions of board pieces in a way or another. Upon the 16 patterns that belong the same equivalence class when considering the symmetries, rotations, and black and white inversions, a first approach to match a given pattern with a piece of board consists in storing one pattern only in memory, and let the pattern-matching algorithm compare the actual piece of board with the 16 patterns equivalent to the given pattern. The other approach consists in storing explicitly the 16 patterns equivalent to a pattern, and lightens the pattern-matcher algorithm with the symmetries, rotations and black and white inversions. In our first release, not yet concerned with memory limitation but with fast development, we have chosen the second approach.

3.2 Creating patterns

For a given set of games, the creating process is straightforward. It corresponds to the following pseudo-code:

```
Forme_k::createPatterns() {
  For k = 1 up to Kmax
    For each game
      For each move i of the game
        createPattern(k, i);
}
```

If the pattern does not exist yet, the function $createPattern(k, i)$ creates the pattern centered on i with k neighbors following the predefined order between intersections. The patterns are stored in a tree whose nodes have four children: the node "if empty", the node "if black", the node "if white" and the node "if edge". Thanks to such a tree pattern-matching is efficient.

4 Bayesian generation

This section describes the bayesian aspect of the work, classical in classification tasks [1]. First, we define and name the relevant probabilities with the bayesian properties of a pattern. We show how we compute the pattern probabilities. Finally, we discuss the way our system eliminates bad patterns.

4.1 Definitions

P names a probablility. i names either an intersection or a move being played on it. p names a pattern. $P(p)$ is the probability that pattern p matches on an arbitrary intersection. $P(i)$ is the probability that the move is being played on i. $P(i, p)$ is the probability that the move is being played on i and that pattern p matches on i. $P(i|p)$ is the probability that the move is being played on i given that pattern p matches on i. Finally, $P(p|i)$ is the probability that pattern p matches on i given that the move is being played on i. $P(i)$ and $P(p)$ are prior probabilities. $P(i|p)$ and $P(p|i)$ are posterior probabilities.

At playing-time, the underlying idea remains to perform pattern-matching on every i intersection of the board, and to use $P(i|p)$ as an estimation of the urgency of the move played on i. At building-time, we adopt a frequentist approach, a probability that an event arises is approximated by the number of times that the event arises divided by the number of tests performed. We say that a pattern is *frequent* when $P(p)$ is high, *good* when $P(i|p)$ is high, and *useful* when $P(p|i)$ is high. Therefore, we defined a class

FORME_K whose bayesian properties are specified below. The term "static" is a C++ keyword which refers to a feature of the whole class.

```
class Forme_k {
  static int n_test;
  static int n_play;
  int n_match;                // p.n_match
  int n_play_given_match;     // p.n_play
  static float p_play;        // P(i)
  float p_match;              // P(p)
  float p_play_given_match;   // P(i|p)
  float p_match_given_play;   // P(p|i)
  ...
};
```

The formula to approximate the probabilities by counting the events are:

```
  P(i)   = n_play/n_test;
  P(p)   = p.n_match/n_test;
  P(i|p) = p.n_play/p.n_match;
  P(p|i) = p.n_play/n_play;
```

We do not use the Bayes formula but posterior probabilities only. However, with such definitions, the Bayes formula remains valid:
$(p.n\_match/n\_test).(p.n\_play/p.n\_match) = (n\_play/n\_test).(p.n\_play/n\_play)$

4.2 Computing the pattern probabilities

For a given set of games and a given set of patterns, the bayesian process corresponds to the following pseudo-code:

```
Forme_k::computeProbabilities() {
  n_play = n_test = 0;
  For each pattern p,
    p.n_match = p.n_play = 0;
  For each game {
    For each move of the game {
      n_play++;
      For each intersection i,
        test(i);
    }
  }
  For each pattern p,
    p.p_play = p.n_play/p.n_match;
}

Forme_k::test(i) {
  n_test++;
  patternMatching();
  For each matched pattern p on i {
    p.n_match++;
    if move played on i then
      p.n_play++;
  }
}
```

A test on an i intersection on a given position of a given game answers the two questions: is the move played on i, and which patterns are matching on i ? On 19x19 boards, 200 or 300 tests are performed by position and a game lasts approximately 200 moves, thus 50,000 tests are performed during one game. With the 2,000 professional games we currently have, we reach about 100,000,000 tests.

4.3 Eliminating bad patterns

The underlying idea of this subsection consists in eliminating the patterns which are not good enough or computed with too low a confidence level. First, because the low playing probability patterns are less interesting than the high playing probability patterns, the extracting process only kept p patterns such as $P(i|p) > 0.01$. Second, we can estimate the confidence on the computed probabilities P (being $P(i|p)$) computed at building-time. Basic statistics [12] yield $\sigma = \sqrt{P(1-P)}$. For most patterns we have $P << 1$, thus $\sigma = \sqrt{P}$. The relevant quantity to assess the confidence level is $s(i|p) = \sigma/\sqrt{p.n\_match} = \sqrt{p.n\_play}/p.n\_match$

Then, the system may eliminate p patterns such as $P(i|p) < threshold \times s(i|p)$. However, in practice, we decided to apply this rule only when our set of games is larger. With such pragmatic decision, our system extracted K-dependent databases, K being the maximal number of neighbors considered during generation. Table 1 provides the number of patterns generated for some values of K.

K	6	9	15
patterns	8,000	27,000	85,000

Table 1: Number of generated patterns for $K = 6, 9, 15$.

5 Experiments

There are two possible ways of using FORME_K: direct play as an opening book without MC verification (subsection 5.1), and integration with MC verification (subsection 5.2).

For each way, we set up experiments to assess the effect of FORME_K. One experiment consists in a 100-game match between the program to be assessed, KATIA, and the experiment reference program, the 2004 release of INDIGO that attended the 2004 Computer Olympiads, each program playing 50 games with Black. The result of one experiment is a set of relative scores provided by a table assuming that KATIA is the max player. A positive number in a cell corresponds to a successful integration. Given that the standard deviation of 19x19 games played by our programs is roughly 75 points, 100 games enable our experiments to lower σ down to 7.5 points (only) and to obtain a 95% confidence interval of which the radius equals 2σ, i.e., 15 points. We have used 2.4 GHz computers. INDIGO and KATIA both use the handcrafted databases FORME_B, FORME_C, FORME_M and FORME_3X3. Besides, KATIA uses FORME_K.

5.1 Using Forme_K as an opening book without MC verification

The initial idea was to replace FORME_B and FORME_C by FORME_K. FORME_B and FORME_C are parts of the MC preprocessor, implying that their moves are verified by the MC module. However, to assess the effect of the FORME_K more frankly and quickly, we decided that KATIA will directly play the move advised by FORME_K, using it as an opening book without MC verification. Figure 1 yields the first 40 moves of a go opening self-played by KATIA in such a way. This opening is played with a very good style indeed, and may appear as very smart by human go players. One should believe that this opening is produced by strong human players. In fact, this appearance can be misleading. Against weak opponents that do not play with professional style, KATIA would not be able to confirm the good behavior shown by Figure 1. As shown in the following, the result of this book approach decreases rapidly after move forty. However, this good opening reflects the strength of a bayesian approach on a K-nearest-neighbor representation in go.

After this qualitative assessment, it is now important to assess FORME_K in terms of quantitative results. Table 2 shows the results between KATIA(K, BEGIN) and INDIGO. During the first $begin$ moves, the move played by KATIA is the move advised by FORME_K. After the opening stage, KATIA keeps using the same move selection as INDIGO.

	6	9	15
20	-9	-2	-5
30	-9	-8	+3
40	-30	-10	-6

Table 2: Average result of KATIA(K, BEGIN) against INDIGO for k = 6, 9, 15 and begin = 20, 30, 40.

As expected, the result is improved when K increases, but until $K = 15$ the results remain negative. At this point, KATIA(K=15, BEGIN=30) is the only positive result. We wondered why the results were not satisfactory. In fact, replacing FORME_B and FORME_C by FORME_K was misleading. Actually, FORME_B and FORME_C do have importance in INDIGO, and it was better to keep them in KATIA as well.

Thus, giving up the initial idea to replace FORME_B and FORME_C by FORME_K, we have added FORME_K in KATIA and we have kept both FORME_B and FORME_C. This addition gave better results provided by Table 3. Moreover, the correct value of $begin$ remained to be determined. First, this result shows that KATIA was not weaker than INDIGO. Second, because KATIA(BEGIN=40) plays instantly during the first forty moves of the game, she saves about 30% of the thinking process throughout one game. Therefore, at this point, the integration already showed a positive effect in terms of both playing level and time.

0	10	20	30	40	50
	+1.0	+5.4	+0.6	+3.5	-11.1

Table 3: Average result of KATIA(BEGIN) also using FORME_B and FORME_C against INDIGO for begin = 0, 10, 20, 30, 40, and 50.

5.2 Integrating Forme_K with MC verification

Within INDIGO, the knowledge-based preprocessor uses several databases along with the conceptual evaluation function to select ns moves for the MC module. The idea developed by this subsection is then to integrate FORME_K within the MC preprocessor.

We name KATIA(NK) the release of KATIA that selects nk moves with FORME_K, and selects $ns - nk$ moves with the existing preprocessor, finally providing them to the MC module for verification. In 2004, INDIGO used $ns = 7$. Because life and death knowledge is necessary to a go program, and because FORME_K does not contain life and death knowledge, we expected results for nk to vary from 0 up to 4, to keep at least 3 moves concerning life and death. Table 4 shows these results.

	0	10	20	30	40	50
0		+1.0	+5.4	+0.6	+3.5	-11.1
1	-1.3	+4.9	+1.8	+3.6	+2.1	-2.9
2	+9.6	+15.8	+10.0	+6.1	+5.8	-13.0
3	+3.9	+8.6	-0.7	-1.5	-6.7	-20.1
4	+5.1	-2.5	+6.7	-4.2	+1.0	-16.9

Table 4: Average result of KATIA(BEGIN, NK) against INDIGO for begin = 0, 10, 20, 30, 40 and 50 and for nk = 0, 1, 2, 3, 4.

Some of these results are then clearly positive. KATIA(BEGIN=10,NK=2) averages about fifteen points better than INDIGO. It is interesting to comment upon the KATIA(NK=2) results. First, the KATIA(BEGIN=0,NK=2) result shows the effect of integrating FORME_K with MC verification independently of using FORME_K as an opening book. It is interesting to point out the 10 point improvement resulting from the insertion of 2 FORME_K moves within the 7 moves selected by the pre-processor. This fact reflects the lack of patterns within the hand-crafted databases, and the presence of these patterns within FORME_K. Second, the KATIA(BEGIN=10,NK=2) result is also amazing. It shows that KATIA improves by 5 points on average by playing the first 5 moves by using FORME_K as an opening book. The first 5 moves corresponds to the very early beginning in go standards. This result shows that appropriate first 5 moves can already show a positive effect. Third, the KATIA(BEGIN=20,NK=2) result corresponds to a compromise between time and average playing level. The playing level is about the same than KATIA(BEGIN=0,NK=2), but about 20% of thinking time is saved by playing the first 10 moves instantly. Finally, KATIA(BEGIN=30 OR 40,NK=2) can be considered as possible compromises between time and playing level, but due to its very negative result KA-

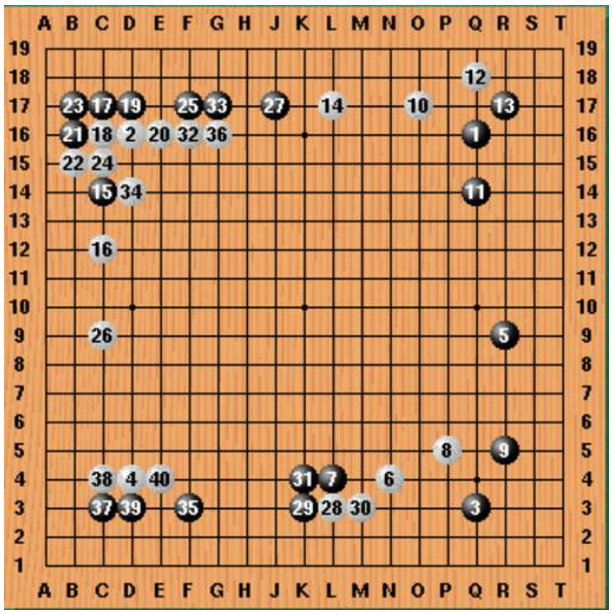

Figure 1: Katia first 40 moves during a self-play opening

TIA(BEGIN=50,NK=2) cannot be. Besides, the results can be commented by column. The best results are obtained for $nk = 2$. FORME_K does not include any life and death information, while the conceptual evaluation function does. It is then normal to observe that a module including life and death is more useful (it provides 5 moves upon 7) than FORME_K which provides 2 moves upon 7.

Finally, by copying the appropriate release of KATIA into INDIGO, may be KATIA(BEGIN=20,NK=2), we can conclude that FORME_K can be successfully integrated into INDIGO, and we are now waiting for the next computer go competition to observe the result against differently designed programs.

6 Perspectives

We plan to re-generate the FORME_K database with a number of games greater than 2,000. For instance, the GoGod CDROM contains about 30,000 professional games, and has the appropriate size for the next assessment. This regeneration would refine the probabilities estimation, and consequently the move urgencies at playing time. An improvement should be observed. Although numerous 9x9 and 13x13 professional games are not massively available, checking the non-regression results of KATIA on 9x9 or 13x13 boards is a mandatory task. Taking the symmetries, rotations and black and white inversions into account within the pattern matching and probability estimations is also an important perspective that will enhance the confidence level of probability estimations. Besides, we also plan to extend the patterns by allowing moves being played not only in the center of the pattern but also on the intersections situated near the center.

Moreover, we have two other interesting perspectives: first, integrating a relevant subset of FORME_K within the conceptual evaluation function module to replace FORME_M, and, second, integrating another appropriate subset of FORME_K within the MC engine to replace FORME_3X3. The first integration should be a difficult knowledge engineering task because FORME_M is used by the conceptual evaluation function in a very intricated way. The second one should be possible provided the patterns are limited to a pre-defined neighborhood of the center of the pattern, because speed considerations are crucial within the MC engine. Another interesting challenge of this second integration is the off-line computation of move urgencies. This can be done either by using a function of the probabilities computed by browsing recorded games, or by reinforcement learning technique [16].

7 Conclusion

We have suggested a method to extract patterns automatically from professional recorded games. This method uses basic probability estimations, and does not assume any domain-dependent knowledge. To this extent, it is a good continuation of a MC go program. The representation used is the K-nearest-neighbor representation. The bayesian generation of K-nearest-neighbor patterns gives an opening book that produces very good openings indeed. This work experimentally demonstrates that the strength of this method lies in the K-nearest-neighbor representation adapted to the game of go. Its weakness lies in its lack of life and death understanding, life and death being the cornerstone of any strong go program. Thus, this approach cannot be used as such, and must be combined with other existing approaches.

We have integrated the database built along such a method into the go playing program INDIGO. The results are positive. Adding the database within the preprocessor of the MC module enables INDIGO to improve by 15 points on 19x19 boards on average, which is significant in go stan-

dards. Furthermore, in the opening of games, the quality of the twenty or thirty first moves provided by the database allows INDIGO to play these moves directly without MC verification. Consequently, 20% of the thinking time of INDIGO can be saved, allowing room for other future improvements.

Bibliography

[1] C. Bishop. *Neural networks and pattern recognition*. Oxford University Press, 1995.

[2] M. Boon. A pattern matcher for Goliath. *Computer Go*, 13:13–23, 1990.

[3] B. Bouzy. Go patterns generated by retrograde analysis. In *Computer Olympiad Workshop*, Maastricht, 2001.

[4] B. Bouzy. Associating knowledge and Monte Carlo approaches within a go program. In *7th Joint Conference on Information Sciences*, pages 505–508, Raleigh, 2003.

[5] B. Bouzy. Mathematical morphology applied to computer go. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(2):257–268, March 2003.

[6] B. Bouzy. The move decision process of Indigo. *International Computer Game Association Journal*, 26(1):14–27, March 2003.

[7] B. Bouzy. Indigo home page. www.math-info.univ-paris5.fr/∼bouzy/INDIGO.html, 2004.

[8] B. Bouzy and T. Cazenave. Computer go: an AI oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

[9] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *10th Advances in Computer Games*, pages 159–174, Graz, 2003. Kluwer Academic Publishers.

[10] T. Cazenave. Automatic acquisition of tactical go rules. In *3rd Game Programming Workshop in Japan*, pages 10–19, Hakone, 1996.

[11] T. Cazenave. Generation of patterns with external conditions for the game of go. In B. Monien H.J. van den Herik, editor, *Advances in Computer Games*, volume 9, University of Limburg, Maastricht, 2001.

[12] CISIA CERESTA, editor. *Aide-mémoire statistique*. 1999.

[13] M. Müller. Computer go. *Artificial Intelligence*, 134:145–179, 2002.

[14] M. Müller. Position evaluation in computer go. *ICGA Journal*, 25(4):219–228, December 2002.

[15] J. Schaeffer and J. van den Herik. Games, Computers, and Artificial Intelligence. *Artificial Intelligence*, 134:1–7, 2002.

[16] R. Sutton and A. Barto. *Reinforcement Learning: an introduction*. MIT Press, 1998.

[17] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In Yngvi Björnsson J. Schaeffer, M. Müller, editor, *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2002.

[18] E. van der Werf, J. Uiterwijk, and J. van den Herik. Learning to score final positions in the game of go. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges*, volume 10, pages 143–158. Kluwer Academic Publishers, 2003.

[19] E. van der Werf, M. Winands, J. van den Herik, and J. Uiterwijk. Learning to predict life and death from go game records. In *7th Joint Conference on Information Sciences*, pages 501–504, Raleigh, 2003.