GENERATION OF PATTERNS WITH EXTERNAL CONDITIONS FOR THE GAME OF GO

T. Cazenave[1]

Labo IA
Saint-Denis, France

Abstract

Pattern databases are constructed to improve the search process in game-playing programs. We have generated pattern databases for the game of Go. The generated patterns are associated to conditions that are external to the pattern. This method enables a pattern to cover much more positions, but it leads to new problems for pattern generation too. We explain how we managed to solve these problems and provide experimental results. Moreover, we believe that patterns associated to external conditions can be useful in other games.

1. INTRODUCTION

In this paper, we explain the generation of pattern-based knowledge associated to external conditions for the game of Go. The approach shows promising properties for the construction of an adequate Go-playing program. In the game of Go, Tsume-Go is an important problem. It consists of finding whether a group is alive (the opponent cannot remove the group) or dead. Finally, patterns have been generated too for connecting and removing stones.

In Section 2, we relate our work to similar work performed in other games, especially in chess and checkers. Then we explain why associating external conditions to patterns is useful in the game of Go. In Section 3, we present how we manage the special pattern-based knowledge. Section 4 is devoted to the explanation and the optimisation of the algorithm that generates the patterns. Section 5 presents the results obtained with this approach, and Section 6 provides conclusions.

---

[1] Labo IA, Dept Informatique, Université Paris 8, 2 Rue de la Liberté, 93526 Saint-Denis, France. E-mail: cazenave@ai.univ-paris8.fr

Throughout the paper, Black is our colour (the friend colour) and White the opponent's colour.

2. DATABASES OF PATTERNS WITH EXTERNAL CONDITIONS

2.1 Game Databases

Perfect-knowledge databases are an effective means for significantly controlling and reducing search trees in many planning domains. In a given planning domain a pattern database enumerates all possible subgoals required by any solution, subject to constraints on the subgoal size. Extensive work on chess endgame databases was initiated by van den Herik and Herschberg (1985), and Thompson (1986). It was pushed further with 6-piece endgames databases by Stiller (1996) and Thompson (1996). Moreover, endgame databases enabled us to discover new chess knowledge (Nunn, 1993) and to play some endgames better than any human. Another well-known application is CHINOOK's set of endgame databases for checkers (Lake, Schaeffer, and Lu, 1994). In single-agent planning, pattern databases have been used successfully to reduce the total number of nodes searched on a standard problem set of 100 15-puzzle positions by over a 1000-fold (Culberson and Schaeffer, 1998), and to find optimal solutions to Rubik's Cube (Korf, 1997). Dynamic pattern-databases construction has been used as a real-time learning algorithm to speed up Sokoban problem solving (Junghanns and Schaeffer, 1998). Some simple raw pattern databases have also been computed for the game of Go (three by three eye patterns in the centre) (Cazenave, 1996).

2.2 Go Patterns with External Conditions

Figure 1 is a position where black stones are alive. Figure 2 contains a group that can live if Black plays first at A. If White plays first, the group cannot make two eyes. The position is called *critical*.

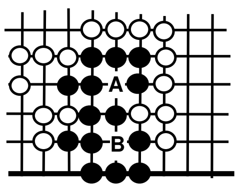


Figure 1: Black stones are alive.

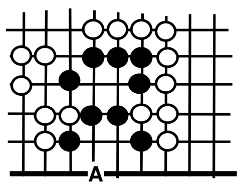


Figure 2: A critical position.

Finding the status of a group (unconditionally alive, alive if friend plays first, dead) and the associated moves is a called a Tsume-Go problem. To solve Tsume-Go problems, Go players use much knowledge about eye shapes.

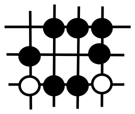


Figure 3: Detecting an eye.

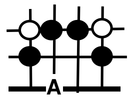


Figure 4: Eye making move.

Some eye shapes are used to foresee that an eye can be made many moves ahead. The shape in Figure 3 is an example. It applies to detect the upper eye in Figure 2 before this is done by a rough estimation. Other eye shapes are used to find moves to play, like the shape in Figure 4 that advises the black move at A making the lower eye of Figure 2.

To date, all the pattern databases used to reduce search trees contain patterns with only raw information. Moreover, a pattern always corresponds to the occupation of raw elements of the problem.

The latter kind of patterns does not take into account the fundamental properties of some domains, as they occur in the game of Go. One essential property of a string of stones is its number of liberties. However, in small patterns like the ones depicted in Figures 3 and 4, some parts of the strings that are present in the pattern are not represented. So, the number of liberties of the strings that border the edge of the pattern (if this edge is not also the edge of the board) cannot be calculated when only the pattern is given.

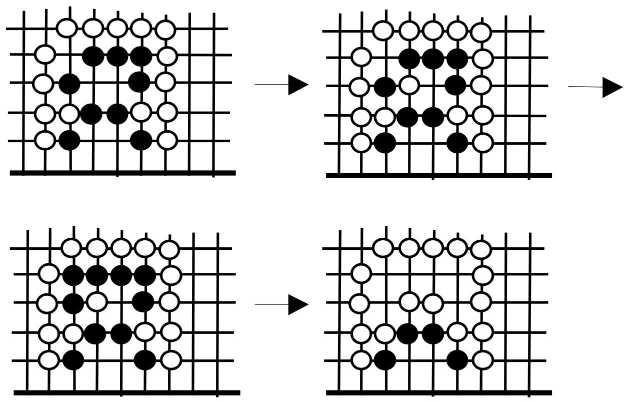


Figure 5: The importance of external liberties.

Figure 5 stresses the importance of the number of liberties of a string. The position at the upper left of Figure 5 has similarities with the position of Figure 2. Moreover the two raw patterns of the Figures 3 and 4 apply to this position as well. But on the contrary of the position of Figure 2, the black group of Figure 5 is dead even if Black plays first. The sequence explaining why the upper enclosed black region is not an eye but a half eye (eye if Black plays first) is given by the sequence of moves following the arrows of Figure 5.

Why are the patterns in the Figures 3 and 4 appropriate for Figure 2 and not for Figure 5? This is due to properties external to the pattern. The difference between the two positions is that when Black answers White's move, the upper black string has two liberties in the first position and only one liberty in the second one. A string with only one liberty is in *Atari*: the opponent can remove it on the next move. So in the first position, White cannot remove a black eye whereas he can in the second position.

To enable our system to handle such positions we have to add external conditions to the elements of our pattern. Elements of a pattern are the string of stones and the empty intersections that it contains. For strings, external conditions associated to the elements of a pattern are conditions on the number of liberties external to the pattern, and for the empty intersections they are conditions on the number of liberties external to the pattern if one colour plays there.

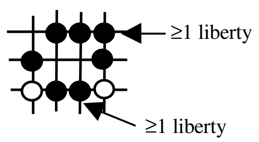


Figure 6: A sample set of conditions.

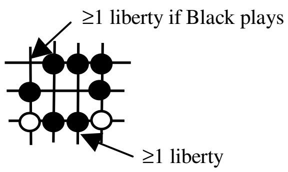


Figure 7: Another set of conditions.

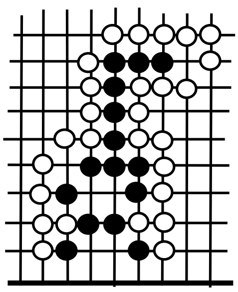


Figure 8: A far and critical external liberty.

Figure 6 gives an example of a set of conditions that has to be added to the pattern in Figure 3, to ensure that it represents an eye whatever is the environment of the pattern: if the upper black string has more than one external liberty, it will have more than one liberty when White puts its stone inside and Black answers on the upper left empty intersection as in Figure 5. So, White will not be able to remove the string after Black's move, and Black will keep his eye.

However, the conditions given in Figure 6 are neither verified for Figure 2 nor for Figure 5. But for each raw pattern there may have more than one set of external conditions attached. For example, Figure 7 gives another set of conditions attached to the pattern of Figure 3 that ensure a Black eye. This time the set of conditions is verified for Figure 2 and not for Figure 5.

One could argue that the need for external conditions associated to a pattern can be overcome by extending the pattern by taking into account its direct environment. But this method makes use of many more patterns and covers less cases. Figure 8 illustrates the large coverage of different positions the external logical information can take into account: the direct environment of the upper eye pattern in Figure 8 is equivalent to the one of Figure 5. However, the black string has one more liberty in the upper right corner of Figure 8. This information is taken into account by the logical condition and could not be taken into account by raw patterns only. Positions involving such slight but vital differences often appear in real games. Therefore, logical external conditions are a convenient, efficient and useful way to represent important knowledge.

Patterns associated to external conditions are used in many Go programs, without them patterns are much less useful. The novelty of our approach is to generate automatically this kind of patterns. We believe that the use of external logical information associated to patterns can improve the use of pattern databases in other domains than Go. Examples of this kind of information could be the existence of a corridor behind an emplacement at Sokoban or the control of a square at chess.

2.3 Computer Tsume-Go

Most Go programs have Tsume-Go problem solvers. Some other programs are specialized in Tsume-Go. The best Tsume-Go problem solver is Thomas Wolf's GOTOOLS (Wolf, 1994). GOTOOLS is a very strong Tsume-Go problem solver. It can solve 5-dan problems (an amateur 5-dan is roughly equivalent to a professional 1-dan Go player). It relies heavily on alpha-beta searching and has numerous hand-coded and well-tuned patterns for directing the search and for evaluating positions. However, GOTOOLS is restricted to completely *enclosed* problems that contain thirteen or less empty intersections (Wolf, 1996) and most of the problems that are to be solved in real games are not enclosed.

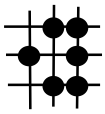
An important assertion that is true for Tsume-Go, and for all goal-based search, is the following. If a rule enables a player to detect life one move earlier and assuming that there is an average of five possible moves at each node of the tree, then finding all the rules that detect life one move earlier reduces the size of the tree by a factor five. Many of our rules enable the detection of won goals many moves ahead (sometimes 10 moves or more), so using our set of generated rules enables us to solve much harder problems than by using a plain problem solver.

3. REPRESENTATION OF GO PATTERNS WITH EXTERNAL CONDITIONS

3.1 Kind of External Conditions

Number of liberties outside the patterns

Each intersection in a pattern can have three values. Each empty intersection on the side of a pattern leads to: (a) three possibilities (no conditions, 0,1) for the slot MaxNumberOfLibertyIfEnemy, and (b) three possibilities (no condition, 1,2) for the slot MinNumberOfLibertyIfFriend. Each string in a pattern leads to three possibilities: (no condition, 0, 1) for the slot MaxNumberOfLiberties if it is an enemy string, and (no condition, 1, 2) for the slot MinNumberOfLiberties if it is a friend string. So each empty intersection on the side of the pattern leads to nine possible choices, and each string in the pattern leads to three possible choices.



For example the pattern in Figure 9 has two empty intersections on its side and two strings. So the number of possible rules that can be tested by the pattern generator is 9×9×3×3=743 different rules.

Figure 9: 743 possible rules are generated using this pattern.

3.2 Possible Moves

Possible moves with external conditions

When checking whether a rule is a winning rule, the program has to try all possible black moves and to find if one leads to a winning rule. The possible black moves are putting black stones on empty intersections. If the intersection has a MinNumberOfLibertiesIfBlackPlays condition, then it is removed and transformed into a MinNumberOfLiberties condition for the new string containing the played black stone. Other possible moves for Black are removing white strings that have no liberties inside the pattern and at most one liberty outside the pattern. If Black plays on an empty intersection in the pattern and if a white string only has this empty intersection as liberty in the pattern and no liberties outside, then the white string is removed from the pattern. Figure 10 gives an example of the possible moves for White.

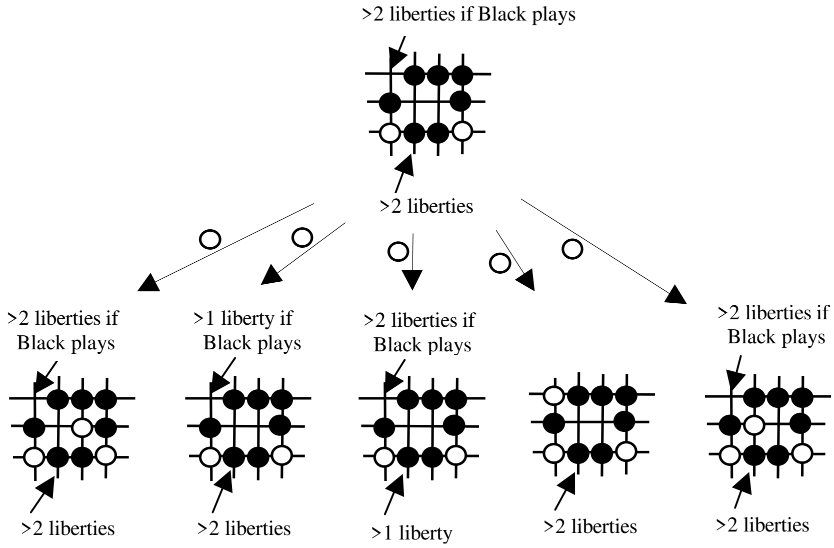


Figure 10: Possible moves for White.

Admissible heuristics on moves

One important property of the game of Go is that a move can remove at most one liberty of a string. Sometimes, liberties are protected and the opponent has to make approaching moves before filling them. The minimum number of moves to remove one liberty to a friend string is one. So, white moves other than putting stones inside a pattern have one of the following consequences: the move (a) decreases the number of liberties of a black string by one, (b) decreases the number of liberties by one if Black plays on an empty intersection, (c) removes an external condition on the maximum number of liberties of a white string, or (d) removes an external condition on the maximum number of liberties if White plays on an empty intersection.

This ensures that we generate rules that enable Black to achieve his goal whatever White does, even if the external environment is completely favourable to White and unfavourable to Black. So there is no need for consistency checking or verification of the generated patterns (except maybe for finding bugs in the generation program, but so far, this has not been done automatically).

Independence of conditions
We hypothesise that the external conditions of the generated rules are independent of each other. This means that the opponent can only modify one of the conditions at each move. This pre-condition has to be verified by the program that uses the generated rules when it matches them.

3.3 Smaller Patterns Included

Included patterns

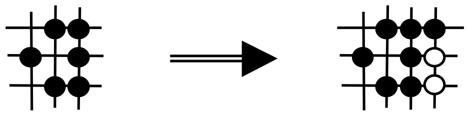


Figure 11: Only the smaller pattern is memorised.

Patterns that contain smaller patterns concluding on the same goals are not memorised. This reduces the number of patterns generated for large patterns considerably since most of the large patterns that can be generated are only small patterns with some useless conditions added. Figure 11 gives an example of the idea. The pattern on the left has been generated as a won eye in the centre, it is a three by three intersections pattern. The pattern on the right is a four by three intersections pattern in the centre, but this pattern can be deduced from the one on the left, so it will not be memorised. The selection of patterns not containing smaller patterns reduces greatly the number of generated patterns. However, it forces to generate pattern sizes using a partial order.

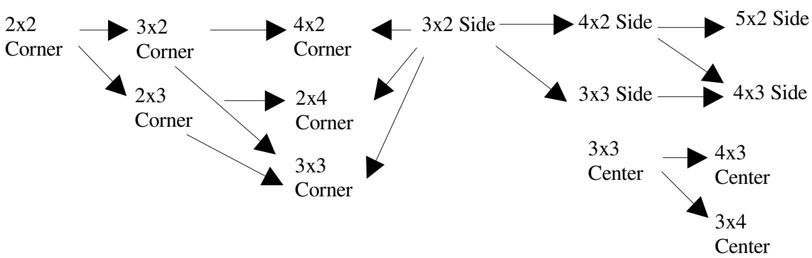


Figure 12: A partial order of pattern sizes.

The partial order is given in Figure 12. Each arrow represents a dependency between a pattern size and another one. The main drawback is that all pattern databases cannot be computed in parallel, we only have a partial parallelism. For example, if we want to compute the four by three intersections on the side eye pattern database, we must wait for the three by two, the four by two and the three by three intersections on the side pattern databases.

Rules are used in two different ways. On one hand new patterns on won eyes or unsettled eyes are used to detect sooner in the proof tree that an eye is made or can be made. On the other hand, patterns that threaten to make an eye and that give forced moves to prevent the opponent to make an eye are used to find appropriate moves to try in the search tree.

Calculating the conditions for inside patterns

When verifying that a smaller pattern is included in a larger one, a set of conditions for the smaller pattern has to be calculated given the larger pattern and its own set of conditions. There is an example in Figure 13 where the empty intersection in the centre of the 4×3 pattern becomes a border empty intersection in the 3×3 sub-pattern. Therefore, we can add a condition that is calculable: if White plays on this empty intersection he will have no external liberties. Similarly, the number of liberties if Black plays on the upper empty intersection is increased by one to take into account the liberty contained in the 4×3 pattern that is external to the 3×3 pattern.

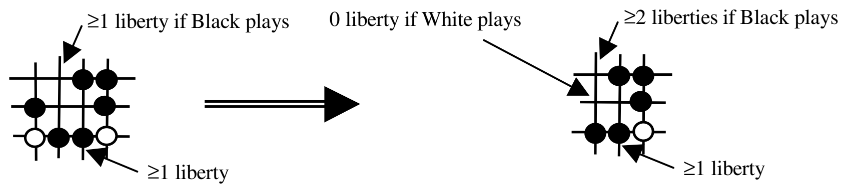


Figure 13: Calculating conditions for inside patterns.

Once the conditions of the subpattern are calculated, the program looks for rules that are more general than the subpattern and its conditions. For example, if the 3×3 rule in the Figure 14 has already been deduced for the same state and the same goal, the 4×3 rule in the Figure 13 will be discarded.

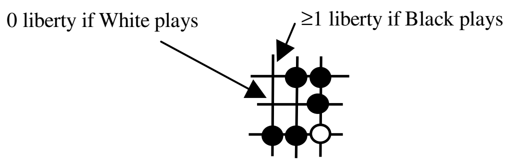


Figure 14: A 3×3 rule more general than the 4×3 rule in the Figure 13.

3.4 Number of Possible Patterns and Rules

In Table 1 the number of possible patterns and rules for different sizes and locations are presented. A pattern is a rectangular shape containing only black, white and empty intersections. A rule is a pattern associated to external conditions. To calculate the number of possible rules, we made a program that generated and counted all of them. All the possible rules we have counted are valid ones and can be matched on some boards.

According to Lake *et al*. (1994), the number of positions for the seven-piece checkers endgame databases is 34,779,531,480 and for the eight-piece databases 406,309,208,481. So, the number of rectangular rules that contains fewer than fifteen intersections is much higher than the number of eight-piece endgame positions in checkers.

Size of the pattern	Location	Total number of possible patterns	Total number of possible rules
2×2	Corner	81	5,133
3×2	Corner	729	184,137
4×2	Corner	6,561	6,498,165
3×3	Corner	19,683	23,719,791
5×2	Corner	59,049	228,469,857
4×3	Corner	531,441	3,238,523,049
6×2	Corner	531,441	8,023,996,893
5×3	Corner	14,348,907	464,991,949,659
3×2	Side	729	541,101
4×2	Side	6,561	18,513,177
3×3	Side	19,683	191,890,599
5×2	Side	59,049	631,651,053
4×3	Side	531,441	20,752,761,345
6×2	Side	531,441	21,555,306,681
3×4	Side	531,441	68,094,804,369
5×3	Side	14,348,907	2,353,796,975,871
3×3	Centre	19,683	663,693,159
4×3	Centre	531,441	239,111,765,601
5×3	Centre	14,348,907	59,241,069,331,995

Table 1: Number of possible patterns and rules for different sizes and locations.

4. GENERATION OF GO PATTERNS WITH EXTERNAL CONDITIONS

4.1 Coding Patterns

Usually when generating pattern databases, only one or two bits are used per pattern (Lake *et al.*, 1994; Korf, 1997; Culberson and Schaeffer, 1998; Junghanns and Schaeffer, 1998). All patterns are associated to one or two bits, sometimes a byte so as to encode the minimal length to the winning position (Thompson, 1986, 1996; Schaeffer, 1997). We do not use this representation. Instead, each pattern is coded as a 32-bit unsigned integer. This representation takes less memory because out of the total number of possible rules for each size and each location, only a few conclude on a won or a winning state. Moreover, different sets of conditions can be associated to a pattern. It is easier to associate this superset to an entry in a table of patterns.

For example, if we use one bit per rule, the 5×3-in-the-centre rules for won states would take 7,405,133,666,499 bytes, which is out of the question for current machines. If instead we allocate a pointer on a superset of set of conditions for each possible pattern, we get 57,395,628 bytes for the pointer table without counting the memory for the sets of conditions. This is still too much. If instead we record only a table of 32-bit unsigned integers per won pattern, we only use 1317×4=5268 bytes for patterns and roughly the same memory for associated conditions.

4.2 Simple Algorithm

A simple forward algorithm for generating rules is given in Figure 15.

```
do {
   NewPattern=0;
   for (Pattern=0; Pattern<NumberOfPatterns(length,height);
        Pattern++) {
       ForAllArrangementsOfExternalLiberties(Pattern,Liberties) {
       if (NewWonPattern(Pattern,Liberties)) {
         AddWonPattern(Pattern,Liberties); NewPattern=1; } } }
   for (Pattern=0; Pattern<NumberOfPatterns(length,height);
        Pattern++) {
       ForAllArrangementsOfExternalLiberties(Pattern,Liberties) {
       if (NewWinningPattern(Pattern,Liberties)) {
         AddWinningPattern(Pattern,Liberties); NewPattern=1; } } }
   }
while(NewPattern);
```

Figure 15: A simple forward algorithm.

The algorithm looks at all the possible patterns and checks if they are won or winning patterns for the desired goal. However, the algorithm cannot be used for the sizes of the patterns we want to generate. For example, the smallest size of pattern for making life in the centre is 5×3. There are 59,241,069,331,995 different possible rules for 5×3 patterns in the centre. Each time we want to regress rules one move further, the algorithm has to check the huge number of rules.

4.3 Backward Algorithm

Unmove generator

To improve the forward algorithm, we wrote an unmove generator that given a rule provides all the rules that lead to it in one-move (Lake *et al.*, 1994; Thompson, 1996; Gasser, 1996). However, writing an unmove generator is a difficult task when dealing with external conditions and different patterns sizes and locations. It is much easier to write an unmove generator for raw patterns without external conditions. So we improved the simple algorithm by unmoving the raw patterns and looking at all the arrangements of external liberties for the unmoved raw patterns (see Figure 16).

```
for (Pattern=0; Pattern<NumberOfPatterns(length,height); Pattern++) {
      ForAllArrangementsOfExternalLiberties(Pattern,Liberties) {
         if (NewWonPattern(Pattern,Liberties)) AddWonPattern
         (Pattern,Liberties); } }
do {
    NewPattern=0;
    for (i=0; i<NumberOfWinningPatterns; i++) {
      NewPatternsToUnmove=Unmove(Enemy,WinningPattern[i]);
      for (j=0; j<NumberOfNewPatternsToUnmove; j++) {
        Pattern=NewPatternsToUnmove [j];
        ForAllArrangementsOfExternalLiberties(Pattern,Liberties) {
          if (NewWonPattern(Pattern,Liberties)) {
            AddWonPattern(Pattern,Liberties); NewPattern=1; } } }
    for (i=0; i<NumberOfWonPatterns; i++) {
      NewPatternsToUnmove=Unmove(Friend,WonPattern[i]);
      for (j=0; j<NumberOfNewPatternsToUnmove; j++) {
        Pattern=NewPatternsToUnmove [j];
        ForAllArrangementsOfExternalLiberties(Pattern,Liberties) {
          if (NewWinningPattern(Pattern,Liberties)) {
            AddWinningPattern(Pattern,Liberties); NewPattern=1; } } } }
while(NewPattern);
```

Figure 16: A simple backward algorithm.

4.4 Memorizing the Last Iteration

The next optimisation is not to unmove all the patterns but to unmove only the last deduced patterns. This leads to a substantial speed-up for large pattern sizes, as a large number of rules are generated. This optimisation is mentioned in Lake *et al*. (1994). Instead of unmoving all the winning rules, we only unmove the winning rules found during the last iteration. We do the same for the won rules. This significantly reduces the number of rules to unmove at each step.

4.5 Order of Test and Cut

Another important optimisation relies on the property that the program does not have to do all the tests in the `ForAllArrangementsOfExternalLiberties` loop. If we begin to test the arrangements with the most favourable ones for Black, then as soon as one arrangement does not lead to a new rule, we can stop looking for an arrangement less favourable for Black: they would not lead to new rules either.

The optimisation works particularly well when looking for won states, since one white move is sufficient to disprove the won state. As soon as this move is found, the loop is stopped, even if it is the first arrangement tested: the most favourable for Black. It happens many times that a white unmove leads to *no won* state, because all white moves have to be disproved for the state to be won. It happens very rarely that a black unmove does not lead to a winning state.

Type of rules	Time without constraints optimisation	Time with constraints optimisation
4×2 eyes on the side of the board	7 min.	1 min.
3×3 eyes on the side of the board	1 hour 41 min.	13 min.
5×2 eyes on the side of the board	9 hours 10 min.	55 min.

Table 2: Impact of constraints optimisation.

Some tests, on a slow workstation, that evaluate the impact of constraints optimisation are given in Table 2.

4.6 Rule-Coverage Reductions

If the number of empty intersections on the side of the pattern is strictly greater than four, the program only keeps intersections at the corner of the pattern. This is a domain-dependent coverage reduction, which enables to keep the number of possible conditions associated to a pattern low, while keeping a large number of interesting rules. For example, in the pattern of Figure 17, only the empty intersections in the corners will be associated to conditions.

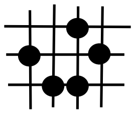


Figure 17: An example of rule-coverage reduction.

5. RESULTS

5.1 Eyes on the Side

Table 3 gives for each pattern size the number of generated rules for eyes on the side. The number of generated rules is remarkably low in comparison with the number of possible rules. This is due to the fact that many conditions must be fulfilled to make an eye. However, these numbers are quite high in comparison with the number of rules used by other Go programs that use hand-written pattern databases. Figure 18 gives a won eye on the side rule generated by the system.

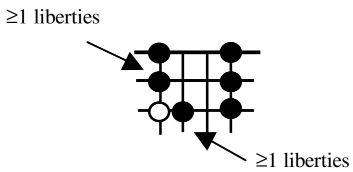


Figure 18: A won eye on the side rule.

Size of the pattern	Location	Number of won rules	Number of winning rules
3×2	Side	11	108
4×2	Side	171	1,081
3×3	Side	727	5,570
5×2	Side	1,661	5,952
4×3	Side	38,909	146,272
3×4	Side	14,966	62,329
6×2	Side	18,194	31,500

Table 3: Number of generated rules for eyes on the side.

5.2 Life in the Corner

Life in the corner of the board is a tricky part of the game of Go. Many patterns gives birth to living strings, and some of them need quite deep and accurate reading for proving life. Table 4 provides some figures on this issue.

Size of the pattern	Location	Number of won rules	Number of winning rules
4×2	Corner	15	164
3×3	Corner	75	977
5×2	Corner	151	1,172
4×3	Corner	10,305	72,014
6×2	Corner	2,916	19,490
5×3	Corner	93,301	483,519

Table 4: Number of generated rules for life in the corner.

Figure 19 shows some generated rules where Black can live in one move (winning rules).

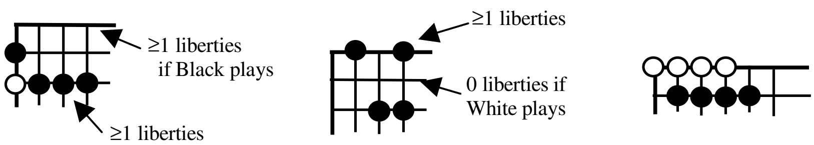


Figure 19: Rules where Black can live in one move.

5.3 Life on the Side

To make life, one needs two eyes, so there are many less life rules than eye rules for the same pattern size. Table 5 provides some numbers.

Size of the pattern	Location	Number of won rules	Number of winning rules
5×2	Side	25	264
4×3	Side	444	5,940
6×2	Side	298	2,808
5×3	Side	30,174	262,541

Table 5: Number of generated rules for won life on the side.

5.4 Other Goals

Rules were generated for the following goals: make an eye, live, connect two strings, connect a string to an empty intersection, connect two empty intersections, remove a string from the board. For each of these goals, large numbers of rules were generated for each of the three possible locations on the board, leading to substantial improvements in the problem-solving abilities of our Go program.

5.5 Using Generated Rules to Solve Problems

To evaluate the impact of new pattern-based search knowledge on the problem-solving performance in Tsume-Go, we used problems from two beginners' books (Kano, 1985a, 1985b). The first book is for beginners and the second for advanced beginners. We used two books of different levels of strength to compare the influence of knowledge on easy problems and harder problems, and to see if our experiment scales well. The first book contains 90 Tsume-Go problems, and the second book 127.

We used proof-number (pn) search (Allis, van der Meulen, and van den Herik, 1994) for our Tsume-Go problems for various reasons. The first reason is that in depth-first search as used in GOTOOLS for completely enclosed problems (Wolf, 1996), good heuristics are available to order moves. For example, the last move of a winning opponent is a good candidate for the loser to try. So the program can learn from terminal leaves of the search tree; therefore depth-first search is appropriate, because it reaches the terminal leaves earlier. It is the contrary for open problems where a wrong move involving a ladder (a ladder is a subgoal of the game consisting in removing some stones of the board) across the board is tried first. The first subtree search may last very long or even forever; the correct blocking move may never be learned and the problem never be solved. So, a best-first search like the one used in pn search is more appropriate for the open Tsume-Go problems our system tries to solve.

The second reason is that we generate control knowledge in the sense that we generate patterns that advise a small number of moves out of the large number of possible moves (meanly 250), but we do not generate ordering knowledge for the selected moves. Correctly ordering the moves to try is very important for the efficient use of the alpha-beta algorithm, and more generally for depth-first search algorithms. The advantage of pn search is that the correct ordering of moves is less important because the interest of each subtree is dynamically evaluated and reconsidered at each move taking into account the information on the shape of the search tree given by the last move.

In this experiment we counted the eight patterns that are equivalent by rotations and symmetries as different patterns. Each rotated and symmetrized pattern is a different item in our database. As we have seen in Section 2, each pattern can have different sets of external conditions attached to it. We counted each different set of conditions as one rule. So one raw pattern having multiple sets of conditions counts for more than one rule.

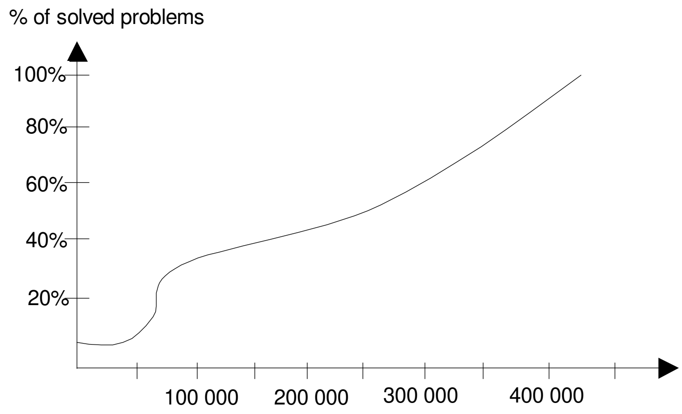


Figure 20: Level increases with generated patterns

In Figure 20, the horizontal axis represents the number of rules used to control and stop the search. The vertical axis represents the number of problems solved by this amount of rules on beginners' problems.

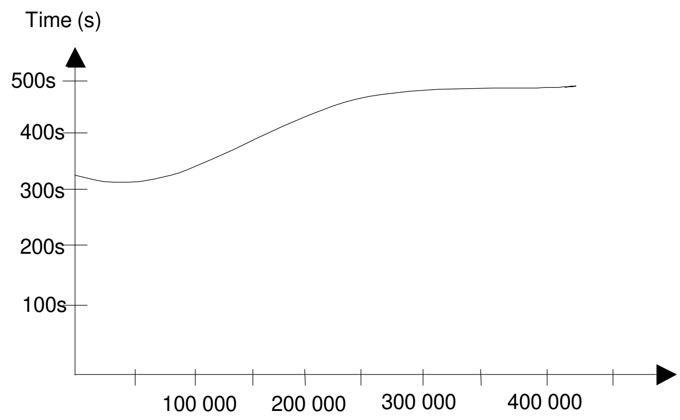


Figure 21: Overhead due to the generated patterns

In Figure 21, the horizontal axis represents the number of patterns used to control and stop the search. The vertical axis represents the time used to search the problems. In order to solve a Tsume-Go problem, our system must recognise the groups of stones on the board. So, much search is performed before solving the Tsume-Go problem at hand: all the subproblems concerning the connection and the capture of stones have to be solved first to build the groups. The system calculated the total time used for building groups and solving Tsume-Go problems. This gives a realistic evaluation of the overhead due to the Tsume-Go solver.

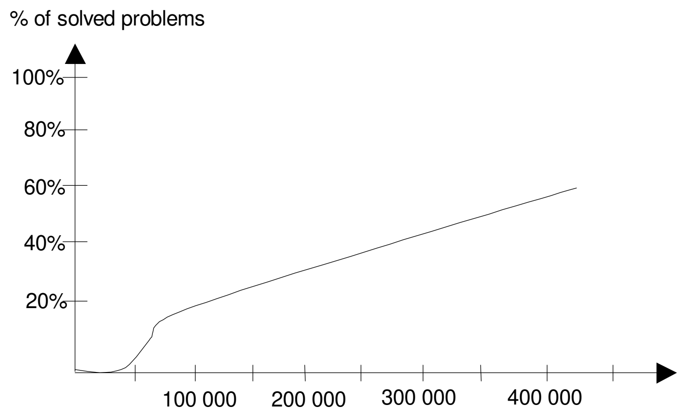


Figure 22: Increase in level on more difficult problems

In Figure 22, the horizontal axis represents the number of rules used to control and stop the search. The vertical axis represents the number of problems solved by this amount of patterns on advanced beginners' problems.

6. CONCLUSION

We have demonstrated the importance of pattern databases equipped with external conditions for computer Go. Moreover, we have shown the importance of logical information in patterns that take into account external properties of the pattern. In summary, we have described the representations and the algorithms used to generate such patterns. The experimental results show that the number of simple problems solved increases well with the number of generated patterns. The additional time used by the playing program to solve problems it could not solve (and not even see as problems) before generating the patterns, is reasonable and involves no time problem for tournament and competitive play. In fact, the mean number of nodes and the mean time used to solve a given problem decreases as the number of pattern increases. When tested on harder problems, the experiments scale well and show a similar increase of the number of problems solved with the number of patterns. Some games played by our system during tournament play show that its pattern databases and search algorithm give it a better understanding of Tsume-Go than the best Go-playing systems on some positions. These experimental results are an encouragement to continue working on pattern databases associated to external logical information in Go and to test this approach in other games and single-agent search domains.

7. REFERENCES

Allis, L.V., van der Meulen, M., and Herik, H.J. (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No.1, pp. 91-124. ISSN 0004-3702.

Cazenave, T. (1996). Automatic Acquisition of Tactical Go Rules. *Game Programming Workshop in Japan '96*, Hakone, Japan.

Culberson, J.C., Schaeffer, J. (1998). Pattern Databases. *Computational Intelligence,* Vol. 14, No. 3, pp. 318-334. ISSN 0824-7935.

Gasser, R. (1996). Solving Nine Men's Morris. *Games of No Chances* (ed. R.J. Nowakowski), Vol. 29, MSRI Publications, Cambridge, MA. ISBN 0-5216-4652-9.

Herik, H.J. van den and Herschberg, I.S. (1985). The Construction of an Omniscient Endgame Database. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87. ISSN 0920-234X.

Junghanns, A. and Schaeffer, J. (1998). Single-Agent Search in the Presence of Deadlock. *Proceedings of the 17th National Conference on Artificial Intelligence* (AAAI-98), pp. 419-424.

Kano, Y. (1985a). Graded Go Problems for Beginners. Volume One. *The Nihon Ki-in*. ISBN 4-8182-0228-2. C2376.

Kano, Y. (1985b). Graded Go Problems for Beginners. Volume Two. *The Nihon Ki-in*. ISBN 4-9065-7447-5.

Korf, R. (1997). Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. *Proceedings of the 16th National Conference on Artificial Intelligence* (AAAI-97), pp. 700-705.

Lake, R., Schaeffer, J. and Lu, P. (1994). Solving Large Retrograde-Analysis Problems Using a Network of Workstations. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 135-162. University of Limburg, Maastricht, The Netherlands. ISBN 90-6216-1014.

Nunn, J. (1993). Extracting Information from Endgame Databases. *ICCA Journal*, Vol. 16, No. 4, pp. 191-200. ISSN 0920-234X.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy at Checkers*. Springer-Verlag, New York, NY. ISBN 0-387-94930-5.

Stiller, L. (1996). Multilinear Algebra and Chess Endgames. *Games of No Chances* (ed. R.J. Nowakowski), Vol. 29, MSRI Publications, Cambridge, MA. ISBN 0-5216-4652-9.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal* Vol. 9, No. 3, pp. 131-139. ISSN 0920-234X.

Thompson, K. (1996). 6-Piece Endgames. *ICCA Journal*, Vol. 19, No. 4, pp. 215-226. ISSN 0920-234X.

Wolf, T. (1994). The Program GoTools and its Computer-Generated Tsume-Go Database. *First Game Programming Workshop in Japan*, Hakone, Japan.

Wolf, T. (1996). About Problems in Generalizing a Tsume-Go Program to Open Positions. *Game Programming Workshop in Japan'96*, Hakone, Japan.