Generation of Patterns With External Conditions for the Game of Go

Tristan Cazenave[1]

Abstract. Patterns databases are used to improve search in games. We have generated pattern databases for the game of Go. The generated patterns are associated to conditions external to the pattern. This enables the pattern to cover much more positions, but it leads to new problems for pattern generation. We explain how we have managed to solve these problems. We believe that patterns associated to external conditions can be useful in other games.

1 INTRODUCTION

In this paper, we explain how to generate pattern-based knowledge associated to external conditions for the game of Go. This approach has good properties for the game of Go. Tsume-Go is an important problem of the game of Go, it consists in finding if a group is alive (the opponent cannot remove it) or dead. Patterns have also been generated for connections and removing of stones.

In the second section, we relate our work to similar works in other games and especially Chess and Checkers, then explain why associating external conditions to patterns is useful in the game of Go. In the third section, we present how we manage this special pattern based knowledge. The fourth section is devoted to the explanation and the optimization of the algorithm that generates the patterns. The fifth section presents the results obtained with this approach.

In this paper, Black is the friend color and White the enemy color.

2 USEFULNESS OF DATABASES OF PATTERNS WITH EXTERNAL CONDITIONS

2.1 Game Databases

Perfect knowledge game databases are an effective mean for significantly controlling and reducing the search trees in many planning domains. A pattern database enumerates in a given planning domain all possible subgoals required by any solution, subject to constraints on the subgoal size. Work on Chess endgame databases was initiated in (Herik & al. 1985, Thompson 1986). It was pushed further with 6-piece endgames databases by L. Stiller, K. Thompson (Stiller 1996, Thompson 1996). Endgame databases enabled to discover new chess knowledge (Nunn 1993) and to play some endgames better than any human. Another well known application is Chinook's endgame databases for Checkers (Lake & al. 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 1000-fold (Culberson & al. 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 & al. 1998). Some simple raw pattern databases have also been computed for the game of Go (three by three eye patterns in the center) (Cazenave 1996).

2.2 Go Patterns with external conditions

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

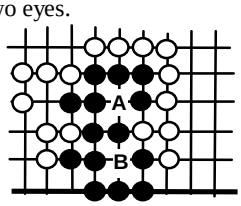


Figure 1

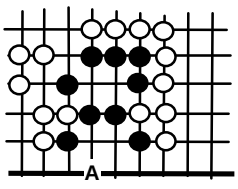


Figure 2

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 a lot of knowledge about eye shapes.

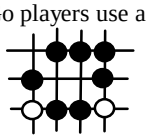


Figure 3

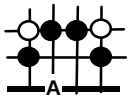


Figure 4

Some eye shapes are used to detect that an eye is done some moves in advance like the shape in the Figure 3 that apply to detect the upper

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

eye in the Figure 2 before it is rawly done. Other eye shapes are used to find the moves to play, as the shape in the Figure 4 that advises the Black move at A to make the lower eye of the Figure 2.

To date, all the pattern databases used to reduce search trees contained pattern with only raw information. The element of the pattern always correspond to occupation of raw elements of the problem.

These kinds of patterns do not take into account some fundamental properties of some domains, as it is the case 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 the 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 the pattern only is given.

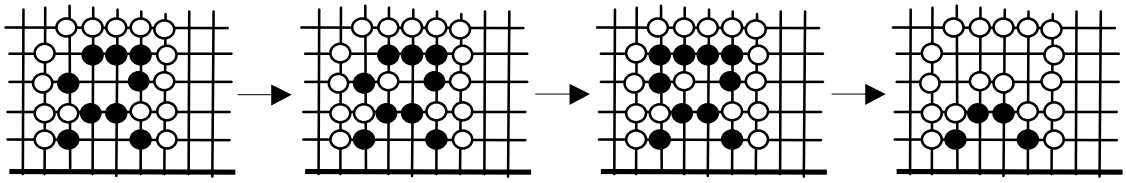


Figure 5

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

Why does the patterns in the Figure 3 and 4 are right for the Figure 2 and not for the Figure 5 ? This is due to properties external to the pattern. The difference between the two positions is that when Black answers White 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 the next move. So in the first position, White cannot remove Black eye whereas he can in the second position.

So as to enable our system to handle such positions we have to add external conditions to the elements of our pattern. The element of a pattern are the string of stones and the empty intersections that it contains. The external conditions associated to the elements of the pattern are conditions on the number of liberties external to the pattern for strings and conditions on the number of liberties external to the pattern if one color plays there, for the empty intersections.

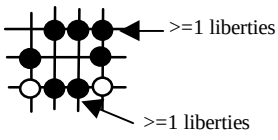


Figure 6

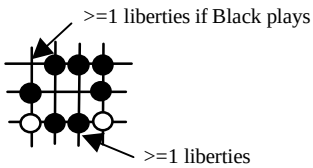


Figure 7

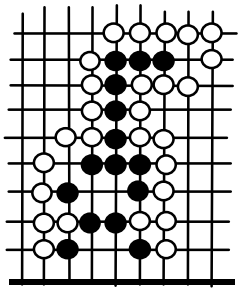


Figure 8

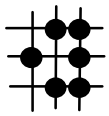


Figure 9

The Figure 6 gives an example of a set of conditions that have to be added to the pattern in the Figure 3 to ensure that it represent 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 answer on the upper left empty intersection as in the Figure 5. So White will not be able to remove the string after Black move, and Black will keep his eye.

However the conditions given in the Figure 6 are not verified for the Figure 2 nor for the Figure 5. But for each raw pattern there may be more than one set of external conditions attached. For example, the Figure 7 gives another set of conditions attached to the pattern of the Figure 3 that ensure a Black eye. This time the set of conditions is verified for the Figure 2 and not for the Figure 5.

One could argue that the need for external conditions associated to pattern can be overcome by extending the pattern to take into account its direct environment. But this method will make use of many more patterns and will cover less cases. The Figure 8 illustrate the large coverage of different situations the external logical information can take into account: The direct environment of the upper eye pattern in the Figure 8 is equivalent to the one of the Figure 5, however the Black string has one more liberty in the upper right corner of the Figure 8. This information is taken into account by the logical condition and could not be taken into account by raw patterns only. Situations involving such slight but vital differences often appear in real game positions. This is why 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, not using external conditions. 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 Tomas 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 professionnal 1-dan Go player). It relies on heavy Alpha-Beta searching and numerous hand-coded and tuned patterns for directing search and 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 but for all goal-based search is that if a rule enable to detect life on move earlier and 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 to detect won goals many moves ahead (10 moves or more), so using our generated rules enables to solve much harder problems than with a simple 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 three possibilities (no conditions,0,1) for the slot MaxNumberOfLibertyIfEnemy, and three possibilities (no condition,1,2) for the slot MinNumberOfLibertyIfFriend. Each string in a pattern leads to three possibilities : no condition, 0 or 1 for the slot MaxNumberOfLiberties if it is an enemy string, and no condition, 1 or 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 the figure 9 has two empty intersection on its side and two stings. So the number of possible rule that can be tested by the pattern generator is 9*9*3*3=743 different rules.

3.2 Possible moves

Possible moves with external conditions.
When checking if a rule is a winning rule, the program has to try all possible Black moves and 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 in a MinNumberOfLiberties condition for the new string containing the played Black stone. The other possible moves for Black are to remove 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 strings only has this empty intersection as liberty in the pattern and no liberties outside, then the White string is remove from the pattern.

Possible moves for White:

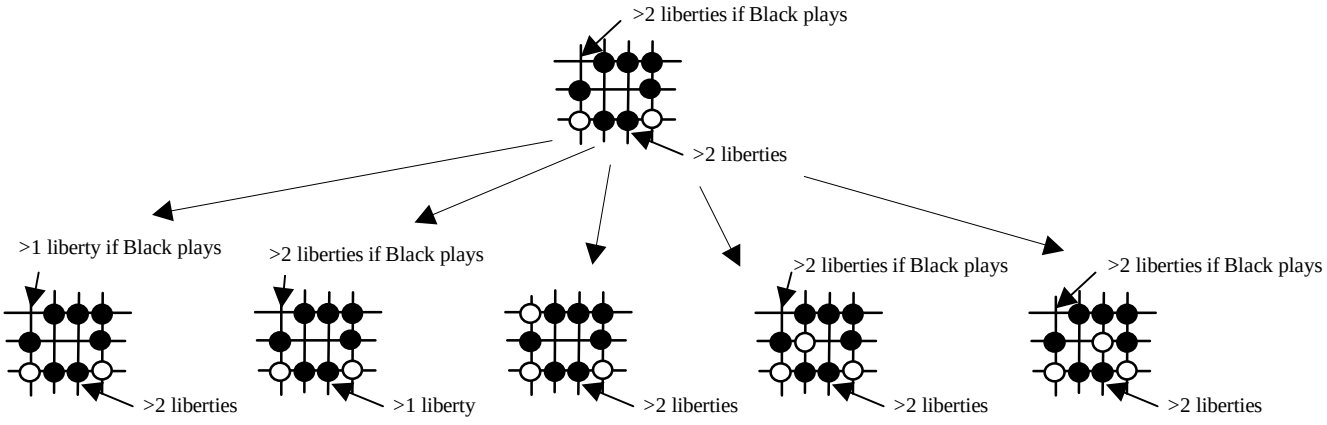


Figure 10

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 approach moves before filling them. The minimum number of moves to remove one liberty to a friend string is one, so the White moves other than putting stones inside the pattern are either decrease by one the number of liberties of a Black string, or decrease by one the number of liberties if Black plays of an empty intersection, or remove an external condition on the maximum number of liberties of a White string, or remove 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 favorable to White and defavorable to Black. So there is no need for consistency checking or verification of the generated patterns (except maybe to find bugs in the generation program, but it has not been done automatically).

Independence of conditions

We make the hypothesis that the external conditions of the generated rules are independent of each other. That 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

Patterns included

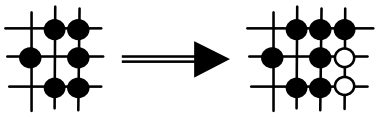


Figure 11

Patterns that contain smaller patterns concluding on the same goals are not memorized. This enables to reduce a lot the number of patterns generated for large patterns as most of the large patterns that can be generated are only small patterns with some useless conditions added. The Figure 11 gives an example of this. The pattern on the left has been generated as a won eye in the center, it is a three by three intersections pattern. The pattern on the left is a four by three intersections pattern in the center, but this pattern can be deduced from the one on the left, so it will not be memorized. 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.

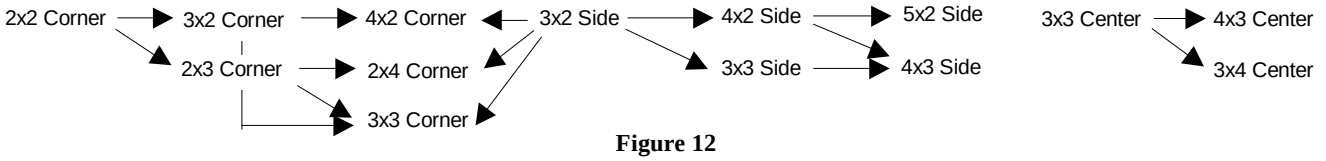


Figure 12

The partial order is given in the 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 can only have a partial parallelism. For example, if we want to compute 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 to be computed.

Patterns 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 the 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, set of conditions for the smaller pattern have to be calculated given the larger pattern and its own set of conditions. There is an example in the Figure 13 where the empty intersection in the center of the 4x3 pattern in the center become a border empty intersection in the 3x3 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 4x3 pattern that is external to the 3x3 pattern.

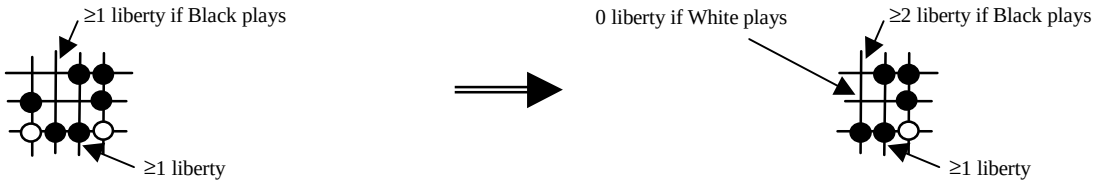


Figure 13

Once the conditions of the sub-pattern are calculated, the program looks for rules that are more general than the sub-pattern and its conditions. For example, if the 3x3 rule in the Figure 14 has already been deduced for the same state and the same goal, the 4x3 rule will be discarded.

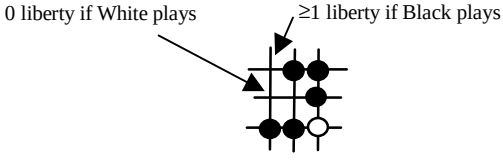


Figure 14

3.4 Number of possible patterns and rules

Size of the pattern	Location	Total number of possible patterns	Total number of possible rules
2x2	Corner	81	5 133
3x2	Corner	729	184 137
4x2	Corner	6 561	6 498 165
3x3	Corner	19 683	23 719 791
5x2	Corner	59 049	228 469 857
4x3	Corner	531 441	3 238 523 049
6x2	Corner	531 441	8 023 996 893
5x3	Corner	14 348 907	464 991 949 659
3x2	Side	729	541 101
4x2	Side	6 561	18 513 177
3x3	Side	19 683	191 890 599
5x2	Side	59 049	631 651 053
4x3	Side	531 441	20 752 761 345
6x2	Side	531 441	21 555 306 681
3x4	Side	531 441	68 094 804 369
5x3	Side	14 348 907	2 353 796 975 871
3x3	Center	19 683	663 693 159
4x3	Center	531 441	239 111 765 601
5x3	Center	14 348 907	59 241 069 331 995

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

According to (Lake & al. 1994), the number of position for the seven pieces Checkers endgame databases is 34 779 531 480 and 406 309 208 481 for the eight pieces Checkers endgame databases. So the number of rectangular rules that contains less than fifteen intersections is much higher than the number of eight pieces endgame positions in Checkers. 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.

4 GENERATION OF GO PATTERNS WITH EXTERNAL CONDITIONS

4.1 Coding patterns

Usually when generating patterns databases, only one or two bits are used per pattern (Lake & al. 1994; Korf 1997; Culberson & al. 1998; Junghanns & al. 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 on a 32 bits 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 and this is easier to associate this superset to an entry in a table of patterns.

For example, if we use one bit per rule, the 5x3 in the center rules for won states would take 7 405 133 666 499 bytes, which is out of 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 bits unsigned integer per won pattern, we only use 1317*4=5268 bytes for patterns and roughly the same memory for associated conditions.

4.2 Simple algorithm

Simple forward algorithm:

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

This simple algorithm looks at all the possible patterns and check if they are won or winning patterns for the desired goal. However, this 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 center is 5x3. There are `59 241 069 331 995` different possible rules for 5x3 patterns in the center. Each time we want to regress rules one move further, this algorithm has to check all this huge number of rules.

4.3 Backward algorithm

Unmove generator.
To improve the forward algorithm, we can write an unmove generator that given a rule gives all the rules that lead to it in one move databases (Lake & 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 pattern 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.

Simple backward algorithm

```
for (Pattern=0; Pattern<NumberOfPatterns(length,height); Pattern++) {
       ForAllPossibleArrangementOfExternalLiberties(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];
        ForAllPossibleArrangementOfExternalLiberties(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];
        ForAllPossibleArrangementOfExternalLiberties(Pattern,Liberties) {
          if (NewWinningPattern(Pattern,Liberties)) {
            AddWinningPattern(Pattern,Liberties); NewPattern=1; } } }
   }
while(NewPattern);
```

4.4 Memorizing last iteration

The next optimization is not to unmove all the patterns but to unmove only the last deduced patterns. This leads to a substantial speedup for large pattern sizes, as a large number of rules are generated, this optimization is mentioned in the paper on Chinook's databases (Lake & 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 reduces a lot the number of rules to unmove at each step.

4.5 Order of test and cut

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

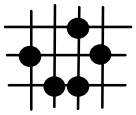
This optimization works particularly well when looking for won states, as one White move is sufficient to disprove the won state. As soon as this move is found, the loop is stopped, even if it is the first arrangement tested: the most favorable for Black. It happens many times that a White unmove leads 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 optimization	Time with constraints optimization
4x2 eyes on the side of the board	7 min.	1 min.
3x3 eyes on the side of the board	1 hour 41 min.	13 min.
5x2 eyes on the side of the board	9 hours 10 min.	55 min.

Table 2: Impact of constraints optimization

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

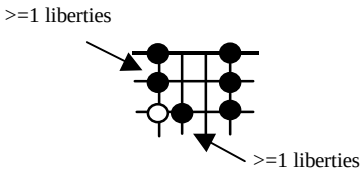
4.6 Rule coverage reductions



If the number of empty intersection 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, that enable to keep the number of possible conditions associated to a pattern low, while keeping a large number of interesting rules. For example, in the next pattern on the right, only the empty intersections in the corners will be associated to conditions.

5 RESULTS

5.1 Eyes on the side



The following table gives the number of generated rules for eyes on the side for each pattern size. The number of generated rules is remarkably low in comparison of the number of possible rules. This is due that many conditions must be fulfilled to make an eye. However these numbers are quite high in comparison of the number of rules used by other Go programs that use hand-written pattern databases. My Go program also used Theatening to make an eye rules, but they are not listed here. The figure 15 gives a won eye on the side rule generated by the system.

Figure 15

Size of the pattern	Location	Number of won rules	Number of winning rules
3x2	Side	11	108
4x2	Side	171	1 081
3x3	Side	727	5 570
5x2	Side	1 661	5 952
4x3	Side	38 909	146 272
3x4	Side	14 966	62 329
6x2	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.

Size of the pattern	Location	Number of won rules	Number of winning rules
4x2	Corner	15	164
3x3	Corner	75	977
5x2	Corner	151	1 172
4x3	Corner	10 305	72 014
6x2	Corner	2 916	19 490
5x3	Corner	93 301	483 519

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

The figure 16 shows some generated rules where Black can live in one move (winning rules).

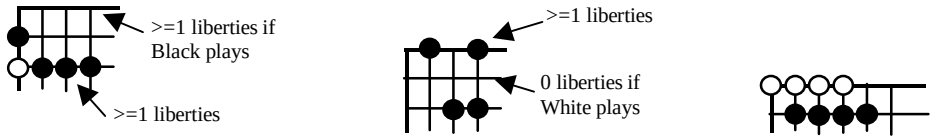


Figure 16

5.3 Life on the side

Size of the pattern	Location	Number of won rules	Number of winning rules
5x2	Side	25	264
4x3	Side	444	5 940
6x2	Side	298	2 808
5x3	Side	30 174	262 541

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

To make life, one needs two eyes, so there are many less life patterns than eye patterns for the same pattern size.

5.4 Other Goals

Pattern were generated for the 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 one 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 experiments scales well. The first book contains 90 Tsume-Go problems, and the second book 127 Tsume-Go problems.

We used Proof-Number (PN) search (Allis & al. 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), there are good heuristics to order moves. For example, the last move of the opponent which won is a good candidate for the looser to try himself before, so the program can learn from terminal leaves of the search tree and therefore depth-first search is appropriate because it reaches the terminal leaves earlier. It is the contrary for open problems where if 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 forever and the correct blocking move 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 to take into account the information on the shape of the search tree given by this last move.

In this experiment we counted the eight equivalent patterns as different patterns, each one counting as one in the number of patterns. The reason is that each different pattern is a different item in our databases as they have different entries. 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 pattern. So one raw pattern having multiple sets of conditions counts for more than one.

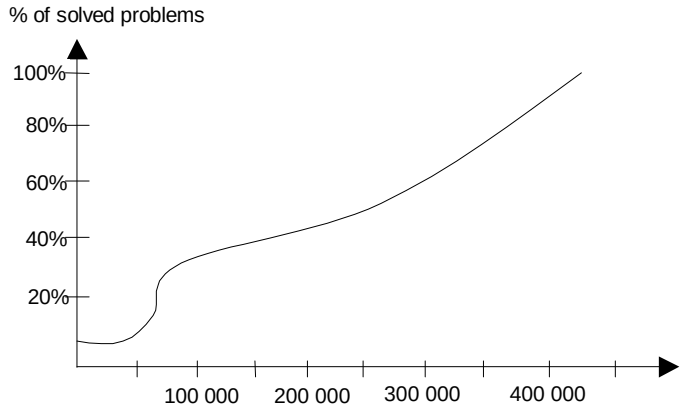


Figure 17

In the figure 17, the horizontal axis represents the number of patterns used to control and stop the search. The vertical axis represents the number of problems solved by this amount of patterns on beginners problems.

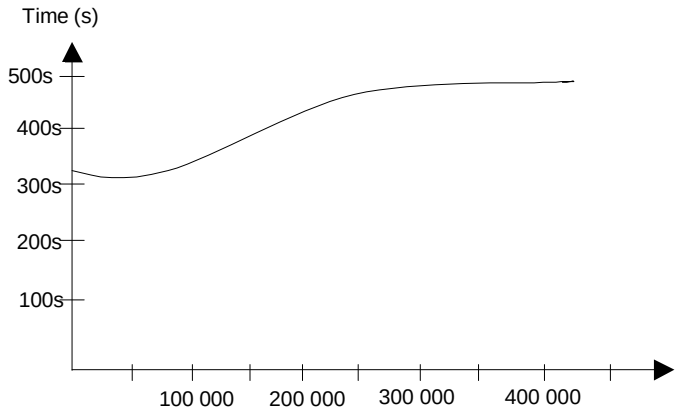


Figure 18

In the figure 18, 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 problems, our system must recognize the groups of stones on the board. So a lot of search is used 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, that way it gives a realistic evaluation of the real difference of time that can be used to evaluate the change in its abilities when it plays real time-limited games.

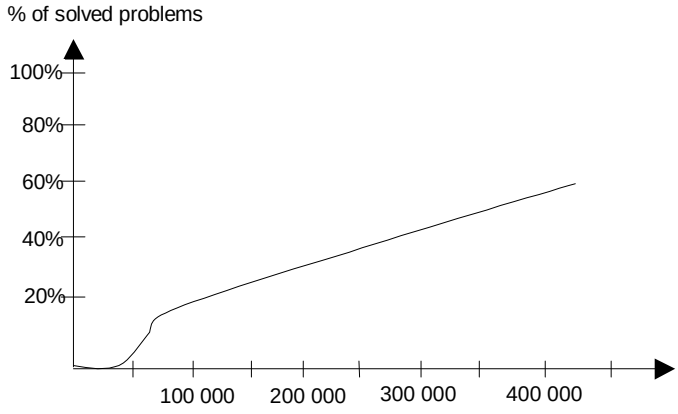


Figure 19

In the figure 19, the horizontal axis represents the number of patterns 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 presented the importance of databases of patterns with external conditions for computer Go. We have shown the importance of logical informations in patterns that take into account external properties of the pattern. 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 informations in Go and to test this approach in other games and single agent search domains.

7 REFERENCES

Allis L. V., van der Meulen M., Jaap van den Herik H. (1994). *Proof-number search.* Artificial Intelligence 66, pp.91-124.

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

Culberson J.C., Schaeffer J. (1998). *Pattern Databases*. Computational Intelligence, 1998.

Gasser R. (1996). *Solving Nine Men's Morris*. In Games of No Chance, R. J. Nowakowski editor, MSRI Publications, Vol. 29, 1996.

Van den Herik, H. J.; Herschberg, I. S. (1985). *The Construction of an Omniscient Endgame Database.* ICCA Journal, Vol. 8, 1985.

Junghanns A., Schaeffer J. (1998). *Single-Agent Search in the Presence of Deadlocks*. AAAI-98.

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 4-906574-47-5.

Korf, R. (1997). *Finding optimal solutions to Rubik's Cube using pattern databases*. AAAI-97, pp. 700-705.

Lake R., Schaeffer J., Lu P. (1994). *Solving Large Retrograde-Analysis Problems Using a Network of Workstations*. Advances in Computer Chess 7, pp. 135-162. University of Limburg, Maastricht, The Netherlands. ISBN 90-6216-1014.

Nunn, J. (1993). *Extracting Information From Endgame Databases*. ICCA journal, December 1993, pp.191-200.

Schaeffer, J. (1997). *One Jump Ahead – Challenging Human Supremacy in Checkers*. Springer Verlag, 1997.

Stiller, L. (1996). *Multilinear Algebra and Chess Endgames*. In Games of No Chances, R. J. Nowakowski editor, MSRI, Vol. 29, 1996.

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

Thompson, K. (1996). *6-Piece Endgames*. ICCA Journal December 1996, pp. 215-226.

Wolf T. (1994). *The program GoTools and its computer-generated tsume-go database*. First Game Programming Workshop in Japan, Hakone, 1994.

Wolf T. (1996). *About problems in generalizing a tsumego program to open positions*. Game Programming Workshop in Japan'96, Hakone, 1996.