RETROGRADE ANALYSIS OF PATTERNS VERSUS METAPROGRAMMING

Tristan Cazenave
Laboratoire d'Intelligence Artificielle,
Département Informatique, Université Paris 8,
2 rue de la Liberté, 93526 Saint Denis, France.

The main objective of this chapter is to present a comparative study of two techniques that automatically generate useful knowledge in games. Retrograde analysis of patterns generates pattern databases, starting with a simple definition of a sub-goal in a game and progressively finding all the pattern of given sizes that fulfill this sub-goal. Metaprogramming is based on similar concepts, but instead of generating fixed size patterns, it generates programs. Programs enable to represent knowledge in a more flexible way. However, they may take more time to use than pattern knowledge. We will describe the application of these two methods to the game of Hex, and compare their behaviors on this game.

1 Introduction

Computer are comparable to humans in many games, even surpassing the world champions in classical games such as Chess or Checkers. However, these excellent results are mainly due to some particularities of these games. In each situation, they is a rather small number of possible moves (8 moves on average for Checkers and 36 moves for Chess), and a situation can be associated to a number that evaluates its goodness with very little computer time. The main problem with these games is to make the computer search the possible moves as fast as possible and numerous and excellent researches have been devoted to

this end. Some other games such as the Game of Go and the game of Hex do not have such nice properties: the average number of moves in each situation is large (250 for 19x19 Go and 90 for 11x11 Hex) and the evaluation of a position is related to abstract and high-level concepts that are hard to capture in a very fast evaluation function.

In the game of Go, a lot of very specific knowledge is required to play well. The best programs for the game of Go are not necessarily the programs that search great deal but are rather those that have the best knowledge of Go. This knowledge is for example a database of patterns to make eyes, or some programs that suggest the interesting moves to capture strings. Hex, which is less difficult to program than Go also requires a great deal of knowledge to build high-level programs.

Some techniques that have proven useful in the game of Go such as automatic generation of interesting pattern databases for some sub-goals, or the automatic creation of programs that select few interesting moves, may also be useful for other games such as Hex.

The goal of this research is to solve the game for greater sizes and to improve the ability of Hex playing programs. It is considered that retrograde analysis of patterns and metaprogramming are both efficient tools to greatly enhance the level of Hex programs, and to solve the game for larger sizes.

The second section describes the game of Hex and introduces the game of Go. The third section shows how pattern databases can be generated for games. The fourth section deals with metaprogramming. The fifth section mentions the use of generated knowledge. The sixth section concludes with experiments in knowledge generation and outlines future work.

2 The game of Hex and the game of Go

2.1 Hex rules

Hex was invented by P. Hein in 1942, and independently rediscovered by John Nash in 1948. The rules are very simple, the initial board is empty and each color (Red and Blue in this paper) alternatively colors an empty hexagon. It is played on an hexagonal grid as shown in Figure 1. It is usually played on 11x11 or larger sizes, and the game is currently solved until the size 7x7. The goal of the game is to connect two opposite borders with hexagons of the same color, each border is given a pre-defined color, and the player that has this color tries to connect the two borders of his color. The players, Red and Blue, take turns putting hexagons of their color on empty hexagons.

An interesting property of Hex is that it cannot end in a draw, because a completely filled board always has two opposite borders connected. This is a property which is not simple to demonstrate. In fact, David Gale demonstrated that this result is equivalent to the Brouwer fixed-point theorem for 2-dimensional squares [8]. It follows that there exist a winning strategy either for the first or the second player. The strategy stealing argument was used by John Nash, to deduce that Hex is a win for the first player: if there were a winning strategy for the second player, then the first player could play a meaningless first move and then apply the strategy to win. The strategy stealing argument works in Hex because the meaningless first move cannot be harmful: coloring an hexagon can never be a disadvantage in Hex. This argument does not work for Chess or Go because in these games, there are moves that make the position worse.

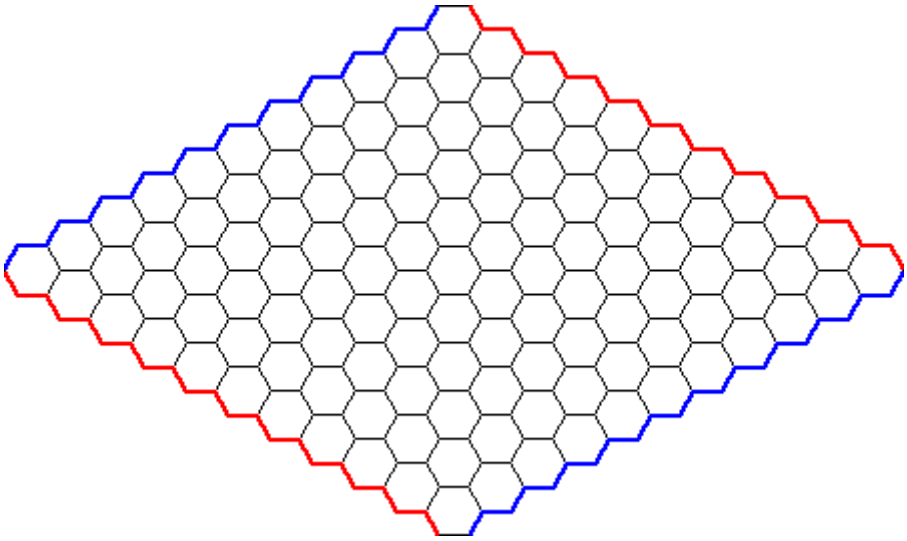

Figure 1. A Hex board.

2.2 Go rules

Go was developed three to four millennia ago in China; it is the oldest and one of the most popular board game in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. In spite of the simplicity of its rules, playing the game of Go is a very complex task. Go is very difficult to program and Computer Go has been recognized as an interesting challenge for Artificial Intelligence [12].

The Go board is made of 19 vertical lines and 19 horizontal lines and therefore 361 *intersections*. At the beginning the board is empty. Each player (Black or White) moves alternatively in adding one *stone* on an empty intersection. Two adjacent stones of the same color are *connected* and they are part of the same *string*. A string is a maximally connected set of adjacent stones of the same color. Empty adjacent intersections of a string are the *liberties* of the string. When a move fills the last liberty of a string, this string is removed from the board. The repetitions of positions are forbidden. According to the possibility of being captured or not, the strings may be *dead* or *alive*. A player *controls* an intersection either when he has an alive stone on it, either when the intersection is empty but adjacent to alive stones. The aim of

the game is to control more intersections than the opponent. The game ends when the two players pass.

A virtual *connection* is a configuration that enables to connect strings whatever the opponent plays. The leftmost diagram of figure 2 gives an example of a ‘Bamboo join’. If the white player plays at A, black plays at B and connects its stones. If white plays at B, then black at A connects. The four stones are virtually connected.

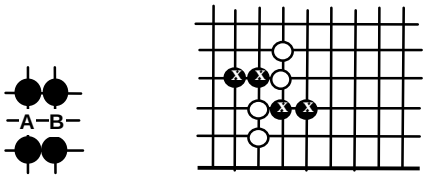

Figure 2. Connection and disconnection in Go.

Go players also reasons with *groups*. A group is a set of intersections that are virtually connected.

2.3 Virtual connections in Hex

Hex has some similarities with the game of Go, virtual connections are very important in both games, but Go is played on a grid and 4-connectivity has mathematical properties that are different from 6-connectivity. Two stones can be separated in Go by non-connected stones as shown in the rightmost diagram of figure 2. Whereas separation of two hexagons implies that the separating hexagons are connected, for example, the two red hexagons of the second diagram of figure 3 can be separated if two connected blue hexagons are played on the two empty separating hexagons. The two blue hexagons that will separate the red ones will be connected. The red hexagons in the figures are marked with a capital R.

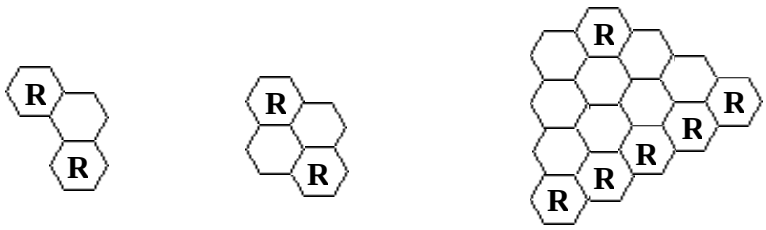

Figure 3**.** Virtual connections in Hex.

Making rotations and symmetries on patterns does not change their properties, a virtual connection is still valid after applying a rotation or a symmetry to a pattern. Applying rotations and symmetries, a pattern can be made equivalent to 12 different patterns, as shown in Figure 4.

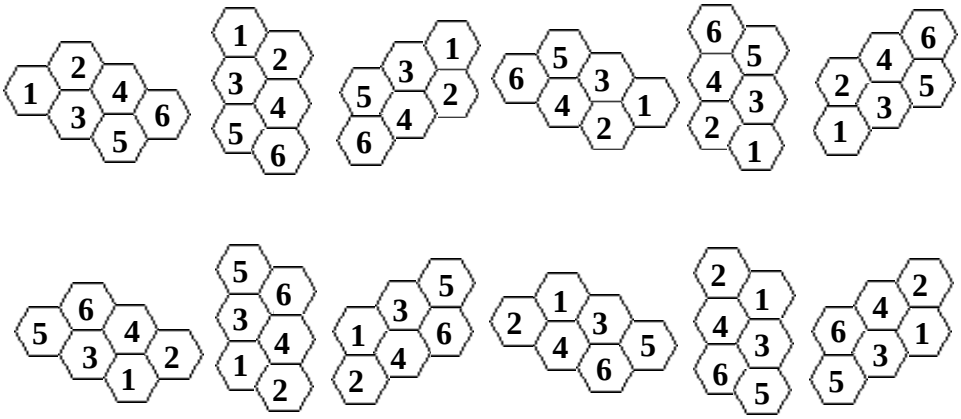

Figure 4. Possible rotations and symmetries of a pattern

With color reversal (switching Red and Blue), a pattern represents 24 different equivalent patterns. Patterns are a convenient and efficient way to represent knowledge on connection. For example the leftmost pattern of figure 3 concludes that the two red hexagons can be connected in on move, the middle one that the two red hexagons are virtually connected, and the right one that all the red hexagons in the pattern are virtually connected. However, representing connection knowledge as programs may also have some advantages as we shall see later.

Other Hex programs are under study [1,14], but they do not use pattern knowledge. We believe that pattern-based knowledge, and more generally knowledge on connection can be very beneficial to Hex programs. Figure 3 gives some very useful connection patterns in Hex that are generated by retrograde analysis. They enable to assess directly the connections between hexagons without any search.

The use of connection knowledge can improve a lot the level of Hex programs. For example, a winning board where the borders are connected through 4 connections that require respectively 2, 4, 5 and 6 moves to be proved. The depth of the Alpha-Beta search [11] to establish that the board is a winning one is therefore 2+4+5+6=17 moves. On this board, the number of possible moves is 100. A naive and brute force approach to solving the board would look at all the possible moves, until two borders are connected. It would approximately need $100^{17}=10^{34}$ nodes in its search tree to solve the problem. In other words, no computer would ever have the time to solve it. Whereas a program knowing how to deduce the different connections directly would see that the board is winning without global Alpha-Beta search. In Hex, as in the game of Go, the notion of decomposability of the whole game into sub-games is essential, moreover knowledge plays a very important role in the computation of the sub-games to reduce the search effort tremendously.

3 Retrograde Analysis of patterns

3.1 Retrograde Analysis

Retrograde analysis consists in analyzing a game backwards. Starting at the terminal positions and undoing moves so as to find positions won many moves ahead. It is very popular for Chess programs [13]. Retrograde analysis has found positions containing six Chess pieces that require up to 262 moves to win. It starts with positions where the king can be captured, and alternatively un-play moves for White and for Black. In Chess, all the 5-pieces endgames have been computed by retrograde analysis, and researchers are now computing the 6-pieces endgames.

3.2 Pattern Databases

A pattern database enumerates in a given game or problem all possible configurations required by any solution, subject to constraints on the pattern size. It is generated with retrograde analysis applied to a few simply defined solutions to the problem. In single agent search , which is related to finding the shortest solution path to a given problem, 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 [7], and to find optimal solutions to Rubik's Cube [10]. Some patterns databases with external conditions (conditions on some properties that are outside the pattern) have also been computed for the game of Go, for the sub-goals of making eyes, making life, connecting two strings and capturing strings [2,6].

3.3 Sub-goals subject to retrograde analysis

The most important sub-goal to analyze is the connection of two hexagons. The hexagon chosen to represent one of the two parts of a connection is taken for all the hexagons that are already connected to it. For the sub-goal of connecting two hexagons, four kinds of patterns are generated: patterns that conclude on virtual connection, virtual connections in one move (i.e. the hexagons can be virtually connected if Red plays one move as in the left pattern of figure 3), virtual connections in two moves (hexagons can be virtually connected if Red plays two moves in a row), and patterns associated to moves to prevent a virtual connection in one move.

The other sub-goal of interest is the connection of empty hexagons: an hexagon is virtually connected to an empty hexagon if they become virtually connected when the player plays on the empty one. Other interesting patterns to generate are the patterns concerning the virtual connection of an empty hexagon in two moves. They are used to try the moves in the search trees related to virtual connection to empty hexagons in one move (see section 5.2 for details).

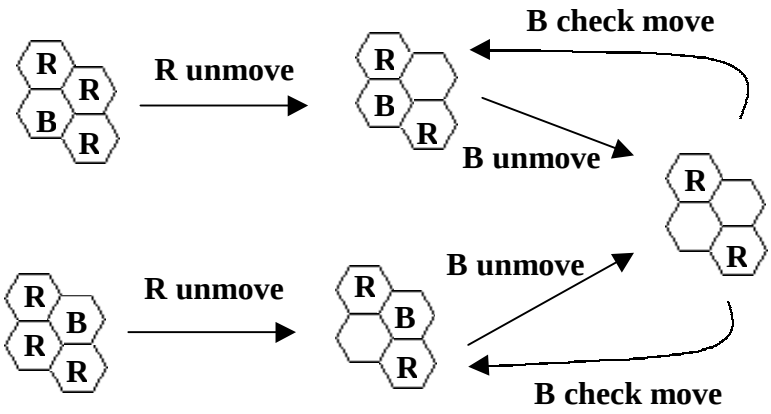

Figure 5. Retrograde analysis of patterns.

Figure 5 shows how the middle pattern of figure 3 is generated using retrograde analysis. Retrograde analysis begins with generating all the patterns that contain connected hexagons. So it generates the two patterns to the left of figure 5, where the red hexagons are connected. Then it un-plays Red moves, leading to the middle patterns of figure 5, which are patterns where Red can connect hexagons in one move. After that, it un-play a Blue move, and verify that all the Blue moves in the un-played pattern lead to pattern where Red can still connect the hexagons. The Blue moves to verify the virtual connection are called the check moves in figure 5. Retrograde analysis iterates this process until no more new patterns can be found.

3.4 The pre-defined shapes for the generation of patterns

The smallest interesting connections shapes are some of the shapes constituted by 3 hexagons. Usually, in Go or in other problems, patterns are represented by rectangular shapes because of the topology of the problem. However the shapes of interesting patterns are more varied in Hex than in Go, due to the hexagonal properties of Hex.

Figure 6 gives a lattice of dependencies between different interesting connection shapes. Each shapes represents the different possible rotations and symmetries that can be deduced from the original. An

arrow between two shapes represent a dependency between the shapes: the pointed shape has to be generated after the smaller shape.

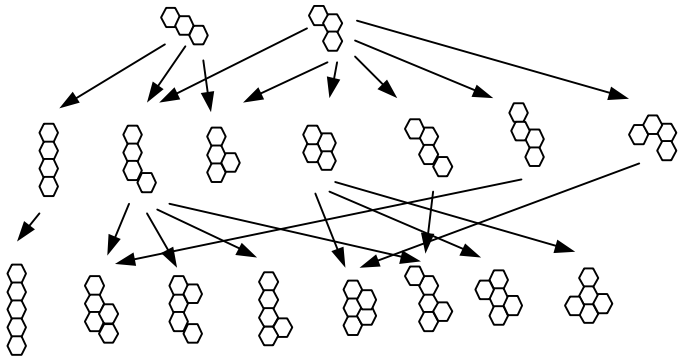

Figure 6. Some shapes dependencies

These dependencies put a partial order on the computations of databases. Small shape databases have to be computed before larger ones. This ensures that many uninteresting large patterns will be discarded. For example in Figure 7, the pattern on the left is more general than the pattern on the right. If the four hexagons pattern databases are computed before the five hexagons ones, then the pattern on the left is discarded because the program can detect that it is a special case of a smaller pattern: every virtual connection found by the larger pattern is also found by the smaller pattern, therefore the larger pattern is useless and can be discarded in order to save the space of storing it and the time to match it.

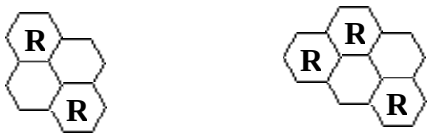

Figure 7. The right pattern is a special case of the left one and is discarded.

In our system, we order the databases calculations by the size of the shape. This ensures that all special rules are discarded, because they are computed after the smaller and more general ones, and can be recognized as special cases of the smaller ones.

Table 1. Number of possible patterns for each pattern size

Number of Hexagons	Possible patterns	Number of Hexagons	Possible patterns
3	27	10	59 049
4	81	11	177 147
5	243	12	531 441
6	729	13	1 594 323
7	2 187	14	4 782 969
8	6 561	15	14 348 907
9	19 683	16	43 046 721

Table 1 gives the number of possible patterns for each number of hexagon. However, out of all these possible patterns, only a few have interesting connection properties that cannot be deduced by smaller patterns. So that databases of large patterns can still fit in memory, because only a few out of all the possible patterns are memorized.

3.5 The use of generated patterns

The main problems associated to the use of automatically generated pattern databases is the cost of recognizing that a pattern is present on the board (i.e. the match cost) and the memory sizes of databases. The match cost is quite low, because generated patterns are sorted and the pattern matching is performed by a binary search of the integer representing the pattern in the array of integers representing a pattern database. The difficult problem is rather the size of the databases that exponentially grows with the number of hexagons.

4 Metaprogramming

Introspect is a logic metaprogramming system that generates rules on worthwhile moves in many games [3,4,5,12]. It writes programs that write other programs that enable to safely cut search trees, therefore enabling great speedups. It is mostly used for games having only two possible outcomes : win or lost. The computation of these games is performed with an AND/OR tree search algorithm. In Hex, the game of connecting two hexagons is such a game. Two kinds of theorems are generated: theorems about moves to reach a tactical goal (at OR nodes),

and theorems that find the complete set of forced moves that prevent the opponent to reach a tactical goal (at AND nodes).

In the first section, we begin with describing the abstract concepts that are used by the generated programs, then, in the second section, we focus on the generation of programs and we detail the kinds of metaprograms used to generate programs.

4.1 Generated programs use abstract concepts

As Introspect generates programs, the generated knowledge can be more abstract and is more flexible than pattern based knowledge. Particularly, abstraction of the board and high level concepts can be easily used. In order to improve generated program efficiency, we now define abstract concepts that have some interest in Hex, and that are used in the generated programs.

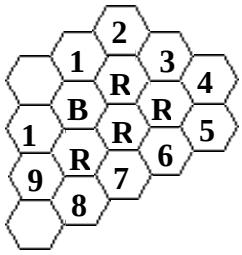

Figure 8. A Red string at Hex.

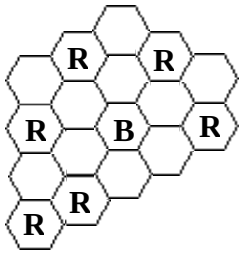

Figure 9. A Red group at Hex.

Figure 8 shows a red string. A string is a maximally connected set of hexagons of the same color. As a parallel to strings in the game of Go, we also define the number of liberties of a string as the number of empty hexagons adjacent to a string. The liberties of the red string in Figure 8 are numbered from 1 to 10. Figure 9 gives an example of a group at Hex: all the red hexagons are independently connected, they form a group.

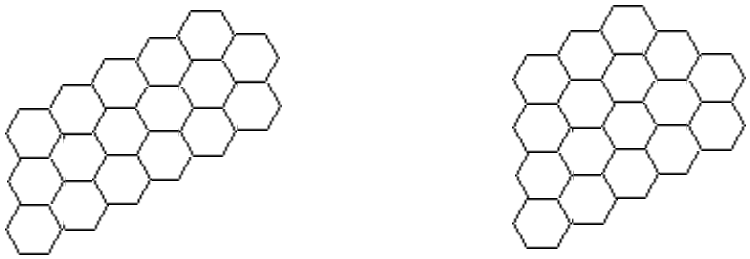

Figure 10. Complicated patterns can be represented as simple programs.

The first pattern of Figure 10 is deduced by the same rule as the second pattern of Figure 3:

```
connected(1,S1,S2,Color):-
        string(S1,Color),string(S2,Color),S1=\=S2
        liberty(H1,S1),liberty(H1,S2),
        liberty(H2,S1),H1=\=H2,liberty(H2,S2).
```

This rule concludes that the strings S1 and S2 of color Color are connected. It verifies that S1 is different from S2, then look at the liberties H1 of S1. It verifies that H1 is also a liberty of S2. When it is the case, it looks at another liberty of S1, named H2, which is different from H1 and which is also a liberty of S2. When it succeeds in finding two such liberties, it can safely conclude that S1 and S2 are connected.

The second pattern of Figure 10 is deduced by a similar generated rule that uses groups instead of strings:

```
same_group(1,G1,G2,Color):-
        group(G1,Color),group(G2,Color),G1=\=G2
        liberty(G1,S1),liberty(G1,S2),
        liberty(G2,S1),H1=\=H2,liberty(G2,S2),
        dependency(G1,Dep1), not_member([H1,H2],Dep1),
        dependency(G2,Dep2), not_member([H1,H2],Dep2).
```

This rules verifies similar things, but it also verifies that the two connecting liberties are not part of the dependencies sets of the groups G1 and G2, in order to avoid interference between the virtual

connections used to build the groups and the connection of the groups. A dependency set is the set of all the empty hexagons that have been tested to establish a virtual connection.

In Hex, as well as in Go, some virtual connections are difficult to capture in a pattern but can be well represented by a program using abstract representations, as shown with the generated rules that apply in figure 10.

4.2 Metaknowledge used for program generation

One of the most important choice in the design of a system that automatically generates programs to solve problems is the representation of the problem. The logic program that represent the problem and the possible actions that can be performed is usually named the domain theory.

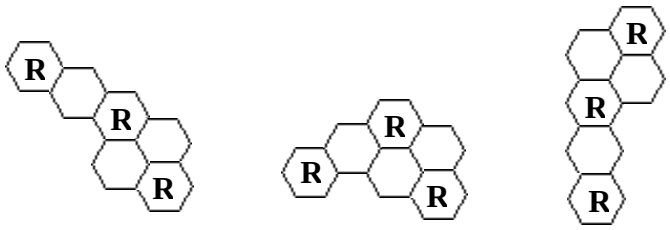

Figure 11. The generality of the generated programs.

Figure 11 shows three different patterns concluding on the possibility of connecting a red hexagon to the others. All these three patterns and many others are represented by only one rule. This is due to the choice to represent all the neighbors of an hexagon by only one predicate in the domain theory: neighbor(H1,H2) is true if H1 and H2 are adjacent hexagons. The rule corresponding to the patterns in Figure 11 is:

```
connected(2,S1,S3,Color):-
        string(S1,Color),string(S2,Color),S1=\=S2,
        liberty(H1,S1),liberty(H1,S2),
        liberty(H2,S1),H1=\=H2,liberty(H2,S2),
        liberty(H3,S2),H3=\=H1,H3=\=H2,
        liberty(H3,S3),S3=\=S1,S3=\=S2.
```

with the following definition for liberty:

```
liberty(H,S):-
        member(H1,S),neighbor(H,H1),empty(H).
```

If each of the six possible directions is explicitly taken into account when designing the domain theory, then the number of generated rules is multiplied by 6^n, n being the number of predicates containing a direction in a generated rule. So for our example, their are 6^6=46 656 different rules to represent only our example rule, the predicate 'neighbor' being replaced by six different predicates, each one corresponding to a different direction, and the predicate 'liberty' being replaced with its definition by unfolding. Naive choices in the design of the domain theory, such as explicitly naming directions, can lead to disastrous metaprogram behaviors.

The basic mechanism of Introspect is unfolding, that is the replacement of some predicates by their possible definitions, taking into account unifications[1]. After each unfolding it uses impossibility and monovaluation metaknowledge. Impossibility metaknowledge is concerned with the removal of generated rules that can never apply. For example if a rule contains the condition '2=\=2', it can be removed because it will never succeed. Monovaluation metaknowledge performs domain dependent unification. For example if a rule contains the conditions 'numberofliberties(S,N)' and 'numberofliberties(S,N1)', it unifies N with N1 because a given string can only have a single number of liberties, its number of liberties is monovaluated.

Another important kind of metaknowledge is the ordering metaknowledge. It finds a good order to verify the conditions of the generated rules. As rules are declarative, their conditions can be matched in any order. However the order in which they are matched has a large influence on the total time used to match the rule. The ordering metaknowledge can be automatically generated by computing statistics

[1] Unification: in a logic program, the simplest form of unification is the equivalence made between two variables. Practically, one of the variables is replaced with the other in the rule. Unifications can also be performed between compound terms.

on the number of some predicates in the set of facts representing a game position.

Metaprograms to generate programs about forced move are also given to Introspect. Forced moves are associated to situation where Red can make a virtual connection in one move. In such situation, the Blue moves that prevent Red from connecting are called forced moves. It is particularly important to find all the forced moves, because forgetting to consider a forced move can lead to false search results.

In Hex this metaprogram is particularly simple, it consists in finding the empty hexagons involved in a generated program concluding on a one-move connection, and append them in the conclusion on forced moves. Some tricks used to metaprogram Hex are very similar to the ones used in Go. For example, in the domain theory, it would be very inefficient to describe all the ways to increase or decrease the number of liberties of a string. A much more convenient knowledge representation is to use the recursive concept liberty_if_move (Liberty,String,MovesList). This hides the complexity of the calculation of liberties in a program that is not unfolded by Introspect, leading to shorter generated programs.

5 Using the generated knowledge

5.1 Dependencies sets

Each object is associated to a dependency set, constituted by a set of hexagons that enable to construct the object. For example, a tree search is associated to the set of hexagons that have been tested during the search, if an unrelated move is played by the opponent and none of the dependency set hexagons has been changed, it ensures that the result of the search is still valid. Dependency sets are associated to objects such as virtual connections and groups. Two objects are independent if there is no common empty hexagon in their dependencies sets.

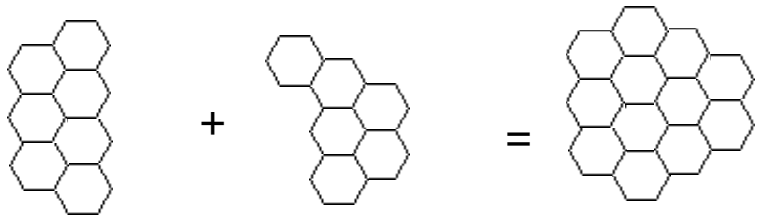

Figure 12. Composition of patterns.

If two patterns do not share empty intersections and both conclude that the same virtual connection can be achieved in one move, then the connection is won. Figure 12 gives an example illustrating this rule. The dependency sets of objects are simple to compute because in Hex, moves are simple and are not dependent on abstract and recursive concepts as in Go for example.

5.2 Tree search with generated knowledge

Two-moves knowledge (either pattern knowledge or generated program knowledge) is used to find the moves at OR nodes, and forced moves knowledge is used at AND nodes. The dependencies of each piece of knowledge used are memorized, and the result of a search is itself associated to a dependency set. Figure 13 gives an example of a search tree in Hex, similar to a ladder in Go. Even numbers are associated to Red moves at the OR nodes of the search tree (Red is trying to reach the goal), and odd numbers to Blue moves at the AND nodes of the tree (Blue is trying to prevent Red from reaching the goal).

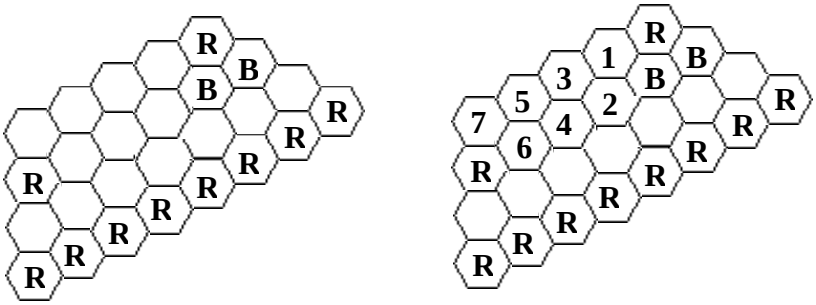

Figure 13. A ladder in Hex

The result of the ladder computation is that the upper right Red hexagon can be virtually connected to the lower border in one move.

5.3 Computing distances and evaluating a board

A stone that is virtually connected to a border is at distance 0 of the border. We also have knowledge on virtual connection to empty hexagons. An empty hexagon that is virtually connected to a string has at most the value of the distance of the string plus one. Given the knowledge on the Red and the Blue distance of all our hexagons, we can compute an evaluation function: Evaluation = Distance between the Friend borders -Distance between the Opponent borders. Of course, whether at a node the distance between two borders equals zero, the evaluation function returns $+\infty$ or $-\infty$ depending on the friendly color and stops searching at this node.

6 Future Work and Conclusion

A promising area of research is the use of partition search [9] to speed-up tree search. As dependencies are easy to compute and maintain in Hex, partition search has good chances to give nice results by recalling already computed properties of some pieces of Hex boards. Therefore saving search time by not computing them again and again.

Another interesting research subject is the compression of the generated pattern databases so as to be able to use even larger shapes and make the program play more accurately, and possibly solving the game for larger sizes.

The use of pattern generated knowledge is not very harmful as large patterns can find connections many moves ahead and as their match cost is quite low. On the contrary generated programs are more time consuming to match as they use abstract concepts, but they enable to recognize better non local connections. So our current choice is to use both form of knowledge so as to be able to get the advantages of the two approaches.

We have shown two methods to automatically generate knowledge for the game of Hex, as well as how the generated knowledge is used in a Hex program. Each method has different properties and they both enable to reduce very significantly the search. These two approaches are based on similar concepts and they are complementary. The knowledge generated by metaprogramming can replace some of the pattern knowledge but at some matching cost, however generated programs perform better in some situations.

References

[1] Anselevich V. (2000), *The Game of Hex: An Automatic Theorem Proving Approach to Game Programming*. AAAI 2000.

[2] Cazenave T. (1993), *Apprentissage de la résolution de problèmes de vie et de mort au jeu de Go*. Rapport du DEA d'Intelligence Artificelle de l'Université Paris 6.

[3] Cazenave T. (1996), Système d'Apprentissage par Auto-Observation. Application au Jeu de Go. Ph.D. diss., University Paris 6.

[4] Cazenave T. (1998), Metaprogramming Forced Moves. Proceedings ECAI98, pp 645-649, Brigthon.

[5] Cazenave T. (1998), Controlled Partial Deduction of Declarative Logic Programs. ACM Computing Surveys, vol. 30, no 3es.

[6] Cazenave T. (2000), Generation of Patterns with External Conditions for the Game of Go. Advances in Conputer Games 9.

[7] Culberson J.C., Schaeffer J. (1998), Pattern Databases. Computational Intelligence.

[8] Gale D.(1986), *The Game of Hex and the Brouwer fixed-point theorem*. American Mathematical Monthly, pp. 818-827.

[9] Ginsberg M. L. (1996), *Partition Search*. AAAI-96.

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

[11] Marsland T. A., Björnsson Y. (2000), From Minimax to Manhattan. Games in AI Research, pp. 5-17. Edited by H.J. van den Herik and H. Iida, Universiteit Maastricht. ISBN 90-621-6416-1.

[12] Pitrat, J. (1998), Games: The Next Challenge. ICCA journal, vol. 21, No. 3, September 1998, pp.147-156.

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

[14] van Rijswijck, J. (2000), Are Bees Better Than Fruitflies ? Experiments with a Hex Playing Program. Canadian conference on AI, 2000.