

# Pattern Matching in GO with DFA

Tanguy Urvoy\* and gnugo team†

## Motivation

An important part of knowledge in Go programs is based on two dimensional patterns. When these patterns are used as heuristic in reading algorithms the pattern matching can become time critical. We present a pattern matching algorithm based on Deterministic Finite State Automata. This algorithm and its incremental extension are detailed in gnugo documentation [3].

## 1 Pattern Representation

We use the symbols '+' for *empty*, 'O' for *white* and 'X' for *black*. A *fixed* Go pattern is a two dimensional grid of values in  $\{+, O, X\}$ . More generally, a Go pattern  $p$  is an application from  $\mathbb{Z} \times \mathbb{Z}$  to the subsets of  $\{+, O, X\}$  describing the allowed colors at a given relative position on the board. The combination  $p(\delta_x, \delta_y) = \{+, O\}$  for example, meaning *no black stone* at  $(\delta_x, \delta_y)$ , is not possible with fixed patterns.

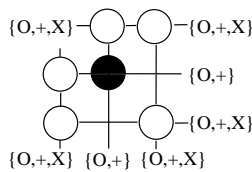


Figure 1: A geta pattern.

A pattern matches the board on a given position  $(x, y)$  iff the value of any intersection  $(x + \delta_x, y + \delta_y)$  of the board is in  $p(\delta_x, \delta_y)$ .

## 2 Pattern Matching in Go

The simplest way to find the set of patterns matching on the board is to check for

\*<http://www.irisa.fr/prive/Tanguy.Urvoy>  
†FSF <http://www.gnu.org/software/gnugo>

each board intersection and each pattern if it matches. The worst case complexity of this method is  $o(ds^2m)$  where  $s$  is the size of the board,  $d$  is the number of patterns and  $m$  the max number of elements by pattern. Under the hypotheses of a database containing only fixed patterns of the same size, [1] gives an average  $o(d\frac{s^2}{\sqrt{m}})$  algorithm. Many optimizations of the brute force algorithm with hash tables or bit parallelism [2],[3] are also possible.

The approach used in *explorer* program [5], is robust and allows different patterns sizes: instead of checking individually each pattern, an arbitrary path is chosen to scan the board thus reducing the two dimensional pattern matching to a linear text searching problem (See Figure 2). The system used by [5] is a *patricia tree*. We propose here an enhancement of this method based on DFA.

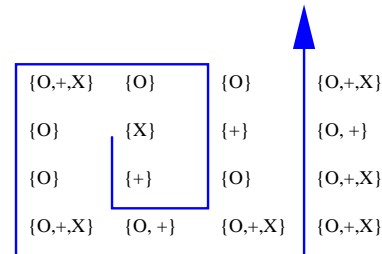


Figure 2: The pattern of Figure 1 scanned following a 'spiral' path gives the following string:  $X+O+OO?OO?\{O,+\}???\{O,+\}$ , where ? stands for  $\{O,+,X\}$  (don't care).

## 3 DFA Pattern Matcher

The use of DFA is common in compilers design. It is also used in well known text searching algorithms like **Knuth-Morris-Pratt** or **Boyer-Moore**. For a complete introduction see [4].

### 3.1 Finite Automata

A *finite automaton* is constituted by a finite set of *states* and by a set of transitions between these states. A particular state is the *initial state*. In Go context we use board values to label transitions and each state is associated to a set of patterns. A pattern is *recognized* when there is a path in the DFA from the initial state to a state containing it. A state is *deterministic* if it admits at most one out-transition by label. An automaton is *deterministic* when all its states are *deterministic*.

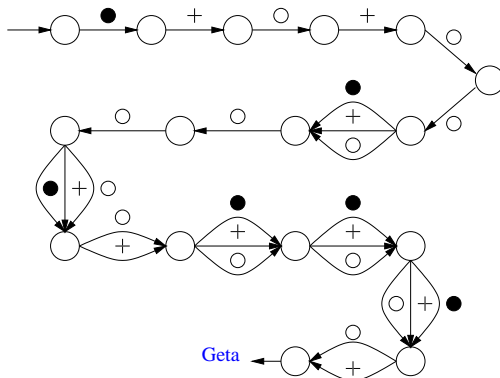


Figure 3: The minimal DFA recognizing the pattern of Figure 2.

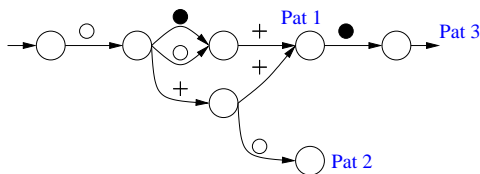


Figure 4: A DFA recognizing both *Pat1*:  $O?+$ , *Pat2*:  $O+O$  and *Pat3*:  $O?+X$ .

The pattern matching with a DFA is simple (and thus fast): start from the initial state and jump from a state to the next following the (only) out-transition labelled by the same value as the one read on the board. Collect all matching patterns on the way and continue until the *error* state is reached. The worst case complexity of this algorithm is  $o(s^2m)$ .

### 3.2 Building the DFA

The delicate point is the insertion of a new pattern in an already existing DFA. The prin-

ciple is to build the minimal DFA recognizing the new pattern and replace the original DFA with its *synchronised product* by the new one. The algorithm performing this product is detailed in [3] and [6]. The worst case size of the obtained DFA is exponential in the number of patterns elements  $dm$ , but experiments shows that this pathologic situation is never reached.

## 4 Experiments and Conclusions

This algorithm is implemented in *gnugo-3.0* program. At level 10, *gnugo* uses approximately 30% of its time in pattern matching and the main speedup is already of 15%. The size of the DFA remain reasonable but a compression method as described in [6] could be used. An incremental version of this algorithm should be faster enough to allow intensive use of pattern matching in deep reading.

## References

- [1] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. In R. Karlsson and J. Gilbert, editors, *SWAT'90*, volume LNCS 447, 1990.
- [2] D. Fotland. Knowledge representation in the many faces of go. available on <ftp://igs.nuri.net>, 1993.
- [3] FSF, [http://www.gnu.org/software/gnugo.GnuGo Reference Manual](http://www.gnu.org/software/gnugo.GnuGo%20Reference%20Manual).
- [4] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C*. Addison-Wesley, 1991. (second edition).
- [5] M. Müller. Pattern matching in explorer. In *In Proceedings of the Game Playing System Workshop*, Tokyo, Japan, 1991. ICOT.
- [6] J.D. Ullman V. Aho, Ravi Sethi. *COMPILERS: Principles, Techniques and Tools*, chapter Optimization of DFA-based pattern matchers. Addison-Wesley, 1979.