Pattern Matching in Explorer

Extended Abstract

Martin Müller, ETH Zürich

Patterns are a simple yet powerful way of encoding Go knowledge. We describe how pattern matching is implemented in our program Explorer.

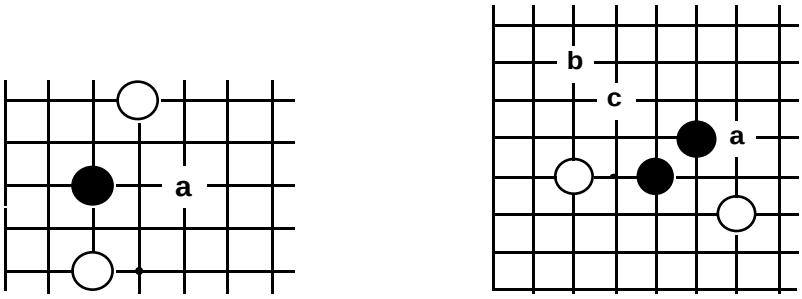
Patterns in Computer Go

We define a Go pattern as a set of points in a rectangle (1≤x≤width,1≤y≤height), where each point(x,y) inside the rectangle has a state *Empty, Black, White,* or *DontCare*.

The state of a point on a go board, denoted by *Board*(x,y), is *Empty, Black*, or *White*.

A pattern *matches* a go board at location (lx,ly) if the following condition holds: For all (x,y) with (1≤x≤width, 1≤y≤height):

(state(x,y) = *DontCare*) OR (state(x,y) = *Board*(lx+x,ly+y))



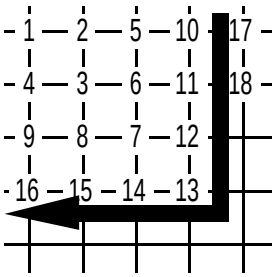
Sample Go patterns

Given a set of patterns and a board, we want to find all pairs of patterns p and locations (x,y) such that p matches the board at (x,y).

Most Go programs use some hashing scheme for pattern matching. This is efficient if all patterns have the same small size [Boon]. We describe a method for matching patterns of variable size that uses *patricia trees* [Gonnet].

A Text Searching Algorithm for Pattern Matching

The patricia tree structure has been used for searching large text databases like the Oxford English Dictionary. We apply this search mechanism to pattern matching in Go:



A linear order for points in a pattern

- Define a linear order of all points in the pattern (like in the picture). Points outside the pattern are assigned the state *DontCare*. Now we can transform the pattern into a *semi-infinite string (sistring).*
- Build a patricia tree index for all pattern sistrings.
- For all points (x,y) on the go board: Match the Board at location (x,y) with the patricia tree. (Use the same linear order for calculating the nth point on the board starting from (x,y)). There may be more than one matching pattern at a location.

Incremental Update of Patterns

One optimisation is very important for playing a game: Matches at a location (px,py) typically depend on a small part of the board only. After a move, an average 93% of all patterns stay valid. To exploit this behaviour we keep a *dependency rectangle* for each point. After each move we must update only the ≈7% of matches with changes inside their dependency rectangle.

Which patterns are relevant?

We have three types of patterns:

- *Center* patterns can occur anywhere on the board. Each pattern can appear in up to 8 symmetric cases.
- *Edge* patterns are fixed to the 4 edges. They can be shifted along an edge. Edge patterns have 2 symmetric cases on each edge.
- *Corner* patterns (e.g. Joseki) are limited to the corners. There are two symmetric cases in each corner.

Each pattern can appear with reversed color. This doubles the number of symmetries.

A typical middle game situation produces about 500 matches from a database of 2500 patterns. Not all patterns are useful. We select by the following criteria:

- We need all patterns that define basic data structures like connections.
- Corner patterns are better than Edge patterns, Edge patterns are better than Center patterns. (A move that works in the middle of the board is often bad near the edge.) We implement this by ignoring moves from Edge patterns inside a Corner pattern etc.
- Big patterns are better than small patterns. Often there is a small pattern describing the general case, but a bigger pattern can take the surroundings into account and propose better moves.

Dia.1 *Dia.2*

To cut or not to cut…

Example: Dia.1 shows the basic crosscut pattern. But if the surrounding position is like Dia. 2 the crosscut pattern is not good anymore. Note that in the same position both patterns will match.

We have developed an algorithm to filter those patterns that are completely covered by some bigger pattern. The original algorithm works for arbitrary patterns. For simplicity we present a version for square patterns only.

Algorithm for finding completely covered square patterns

This algorithm marks all starting points of patterns that would be covered by bigger ones. The crucial observation is that one pattern of size n at location (x,y) covers four patterns of size n-1 at locations (x,y), (x+1,y), (x,y+1) and (x+1,y+1).

```
"Sort patterns according to size MinSize..MaxSize"
Covered := EmptySet; (* the biggest patterns cannot be covered *)

FOR size := MaxSize TO MinSize BY -1 DO
    CoveredThisSize := EmptySet;
    FOREACH pattern IN Patterns[size] DO
        (* if pattern is covered, do not use it *)
        IF NOT(pattern.start IN Covered) THEN
           AddToPatterns(pattern) (* use pattern *)
           INCL(CoveredThisSize, pattern.start);
        END(*IF*);
    END(*FOREACH*);
    Covered := Covered + CoveredThisSize;

    (* Iteration step: n -> n-1 *)
    Covered := Covered+Shift(Covered,1,0)+Shift(Covered,0,1)
               +Shift(Covered,1,1);
END(*FOR*)
```

Summary

We described a pattern matching system for Computer Go. The main advantages of our approach are: No restrictions on number and shape of patterns, search time is logarithmic in number of patterns. We briefly discussed the problems of using patterns in a Go program and gave an algorithm for detecting patterns that are completely covered by a bigger pattern.

References:

[Boon] Boon, M. A Pattern Matcher for Goliath. Computer Go 13, p.12-23.

[Gonnet] Gonnet, Gaston H. Efficient Searching of Text and Pictures, Extended Abstract, UW Center for the New Oxford English Dictionary, June 1988