Automatic Ordering of Predicates by
Game of Go. Efficiently matching these rules is crucial to a Go program. The more it can match rules, the more it understands the position and the better it plays.

Moreover, my program is a learning Go program. Thus the rules it learns do not have a good ordering of predicates. It is vital for him to reason about itself so has to order itself the predicates involved in its rules. Table 1 gives an example of a rule learned by my system.

```
( premises 
    present ( Couleur ( ?c ) )
    present ( Couleur_opposees ( ?c1 ?c ) )
    present ( Couleur ( ?c1 ) )
    present ( Nombre_voisines_couleur_avant ( ?i + 4 ) )
    present ( Nombre_voisines ( ?i 4 ) )
    present ( Nombre_Blocs_voisins_avant ( ?i0 ) )
    present ( Couleur_intersection_avant ( ?i + ) )
    present ( Voisine ( ?i ?i4 ) )
    present ( Couleur_intersection_avant ( ?i4 + ) )
    present ( Voisine ( ?i ?i6 ) )
    present ( Couleur_intersection_avant ( ?i6 + ) )
    intersections_differentes ( ?i4 ?i6 )
    present ( Voisine ( ?i ?i2 ) )
    present ( Couleur_intersection_avant ( ?i2 + ) )
    intersections_differentes ( ?i6 ?i2 )
    intersections_differentes ( ?i4 ?i2 )
    present ( Voisine ( ?i2 ?i1 ) )
    present ( Couleur_intersection_avant ( ?i1 + ) )
    present ( Nombre_voisines ( ?i1 4 ) )
    present ( Nombre_voisines_couleur_avant ( ?i1 ?i0 ) )
    present ( Voisine ( ?i4 ?i1 ) )
    intersections_differentes ( ?i1 ?i1 )
    present ( Nombre_Blocs_voisins_couleur_avant ( ?i1 ?c 0 ) )
    present ( Voisine ( ?i1 ?i7 ) )
    present ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i3 ?i GIII ) )
    present ( Couleur_intersection_avant ( ?i7 + ) )
    intersections_differentes ( ?i4 ?i7 )
    intersections_differentes ( ?i6 ?i7 )
    intersections_differentes ( ?i2 ?i7 )
    present ( Voisine ( ?i7 ?i8 ) )
    present ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i3 ?i GIII ) )
    present ( Couleur_intersection_avant ( ?i8 + ) )
    intersections_differentes ( ?i4 ?i8 )
    intersections_differentes ( ?i2 ?i8 )
    present ( Couleur_intersection_avant ( ?i8 ?i2 ) )
    present ( Voisine ( ?i1 ?i3 ) )
    present ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i3 ?i GIII ) )
    present ( Couleur_intersection_avant ( ?i3 + ) )
    intersections_differentes ( ?i4 ?i3 )
    intersections_differentes ( ?i8 ?i3 )
    intersections_differentes ( ?i2 ?i3 )
    present ( Voisine ( ?i4 ?i5 ) )
    present ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i1 ?i GIII ) )
    present ( Couleur_intersection_avant ( ?i5 + ) )
    intersections_differentes ( ?i1 ?i5 )
    intersections_differentes ( ?i1 ?i5 )
    )

( conclusions 
    ajoute ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i1 ?i GIII ) )
    ajoute ( Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i1 ?i GIII ) )
)
```

Table 1
2.2 The Metarules

My system uses various kinds of metarules. I give in Table 2 two examples of metarules used to give a priority to predicates inside a list of premises. The metapredicate ‘regle’ instanciates in ?r all the rules of a list of rules. The metapredicate ‘condition’ look if its second argument unifies with a premise of the rule in its first argument, ?var and ?var1 are metavariables which can be instanciated in any variable. The metapredicate ‘priorite’ assigns to its third argument the priority corresponding to the premises of the rule in its first argument which unify with the premise in its second argument. The function ‘superieur_reel’ verifies than it first real argument is greater than its second real argument.

```
( nom ( Metaregle_ordonne_1 ) )
premises ( regle (?r)
            condition (?r present ( Voisine (?var ?var1 )))
            instancee (?var)
            non_instancee (?var1)
            priorite (?r present ( Voisine (?var ?var1 )) ?reel)
            superieur_reel (?reel 3.79)
          )
conclusions ( affecte_priorite (?r present ( Voisine (?var ?var1 )) 3.79)
              )
)
```

```
( nom ( Metaregle_ordonne_2 ) )
premises ( regle (?r)
            condition (?r present ( Nombre_voisines (?var 2 )))
            instancee (?var)
            priorite (?r present ( Nombre_voisines (?var 2 )) ?reel)
            superieur_reel (?reel 0.01)
          )
conclusions ( affecte_priorite (?r present ( Nombre_voisines (?var 2 )) 0.01)
              )
)
```

Table 2

The information contained in these rules are about the repartition of the facts in the working memory. They give the average number of instanciations of a variable when the premise instanciates variables. They can also give the probability of unifying a fact when all its arguments are instanciated.

3 Predicate Ordering

3.1 Gathering Informations on Unification

My system has the possibility to observe its behavior when unifying rules. It can collect the number of times it unifies a predicates. The information gathered on a 9x9 Go board working memory is given in Table 3. Each premise and conclusion is followed by the number of time it has been unified.
Table 3

3.2 Example of a learned rule

Table 4 gives an example of a rule learned by my system before ordering. The number of nodes involved in unifying the rule makes it impossible to unify it in reasonable times.
3.3 Rule after ordering

Ordering the premises of the rule makes the unification much faster. Table 5 gives the number of nodes involved. It allows the rule to be unified rapidly. Thus my system is able to unify much more rules. Moreover another mechanism is used so as not to deduce many times the same conclusion using different paths in the unification graph. It consists in verifying the conclusion has not been already deduced when instanciating new variables. This is done by inserting ‘absent’ premises after premises instanciating variables. A priority is given to the instanciation of the variables present in conclusion in order to instanciate them as soon as
possible in the unification of the rule. The sooner they are instanciated in the rules, the more savings are done.

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>111289</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of new facts deduced</td>
<td>104</td>
</tr>
</tbody>
</table>

Table 5

The insertion of ‘absent’ premises approximately doubles the speed of the unification. The unification costs 111,289 node in Table 5 without the absent optimization. It only costs 55315 in Table 3 with the ‘absent’ premises inserted.

4 Conclusion

I have shown how to order predicates using metarules and metapredicates. This rules and metarules are used in a Go learning program playing at an international level [Pettersen 1994].
The method described in this paper allows speedups of 14,000 when matching rules on a working memory representing a 9x9 Go board. It can give even better speedups on larger working memories. This technique can be reused in domains where we know a priori the repartition of the facts in the working memory. This is the case for many domains. I actually apply predicate ordering to rules about other games and about the management of a firm. This work is a part of a longer goal which is to create autonomous self programming systems [Pitrat 1990]. The efficient unification of a great number of rules containing many condition is vital for deductive learning systems. The use of metarules for compiling rules offers a good way to enhance it greatly.

Bibliography


