Learning a Go heuristic with TILDE

Jan Ramon Tom Francis Hendrik Blockeel

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Jan.Ramon,Hendrik.Blockeel}@cs.kuleuven.ac.be

Abstract

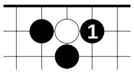
In Go, an important factor that hinders search is the large branching factor, even in local problems. Human players are strong at recognizing frequently occurring shapes and vital points. This allows them to select the most promising moves and to prune the search tree. In this paper we argue that many of these shapes can be represented as relational concepts. We present an application of the relational learner TILDE in which we learn a heuristic that gives values to candidate-moves in tsume-go (life and death) problems. Such a heuristic can be used to limit the number of evaluated moves. Even if all moves are evaluated, alpha-beta search can be sped up considerably when the candidate-moves are approximately ordered from good to bad. We validate our approach with experiments.

1 Introduction

In the board game Go, an important factor that hinders search is the large branching factor. Even in local problems, this can severely limit the ability of an algorithm to read far ahead and find a good solution. Human players are strong at deep reading of Go positions because they easily learn and recognize frequently occurring shapes and vital points, which allows to select the most promising moves at each level and examine only these more thoroughly. This motivates us to consider the task of learning a heuristic for playing Go that is based on the recognition of important patterns. In this paper we more specifically consider the problem of learning a good heuristic in the tsume-go domain.

Most approaches learn patterns that describe a part of the board by assigning pieces to specific positions. This works well for many games, and also for describing certain elementary shapes in Go. However, we argue that many concepts in Go are relational, and hence such propositional patterns are unsuitable for describing them. We will give an example of both attribute-value and relational representations of some concepts and discuss the advantages of both.

The remainder of this paper is organised as follows. In Section 2 we introduce the relational representation language we use. In Section 3 we describe our experiments and analyse the results. We present conclusions and ideas for further work in Section 4.



IF (-3,1)=empty, (-2,0)=black, (-2,1)=empty, (-2, 2)=edge, (-1,-1)=black, (-1,0)=white, (-1,1)=empty, (0,1)=empty, (1,0)=empty THEN black plays on (0,0)

Figure 1: A pattern and a propositional rule using it

2 A relational representation language

We briefly illustrate rather than extensively introduce the first-order logic concepts used in the rest of the paper; more theoretic foundations of Inductive Logic Programming are in [5].

Most machine learning systems use a propositional or attribute-value setting, where hypotheses can be expressed as propositional formulae with propositions of the form "attribute=value"; the attributes could e.g. be board positions and the values *black stone*, *white stone*, *empty*, *edge* (i.e., "off the board") or *don't care*. Such a pattern is given in Figure 1. When this pattern is applicable, the white stone on the second line can be captured by black by playing to the right of it on the intersection labeled '1'. Note that with this approach (e.g. [8], [3] and [4]), patterns have a fixed size (including the don't care positions) and a fixed orientation. Sometimes the representation language is made more expressive by allowing the pattern to have other shapes (e.g. the flexible variant in [4]) or by using graph-based patterns (but then without board-matching part) (see [8], [3]), but not in a systematic way. Another way to increase expressiveness of the language is the use of feature generation, which is frequently applied when using propositional learners or neural networks (see e.g. [3]). These approaches preprocess the data, generating some set of features which are then used in the learning step, though the learner itself is still unable to construct new features.

However, in Go these approaches are often not natural. It is difficult to represent well-known shapes such as 'oi-otoshi', 'double atari', ... in a propositional representation language. Also, common tasks in playing Go such as counting liberties are difficult to implement (if possible at all) using only a propositional language. Therefore, both the board and the relations between groups (and between groups and empty spaces) need to be taken into account.

We propose the relational representation in Figure 2. Every intersection on the board belongs to a group indicated by the BOARD relation. A group can be one empty intersection or can be a string of stones. This is defined by the GROUP relation which specifies the color of a group. Also, for each group the list of adjacent groups is stored in the relation LINK.

These relations give a view at both the board and the intergroup relations. On top of this additional implicit relations, called background knowledge, can be defined. Figure 3 gives some of these relations in PROLOG. For instance, the liberty_cnt predicate contains a *findall* predicate, which makes a list of all

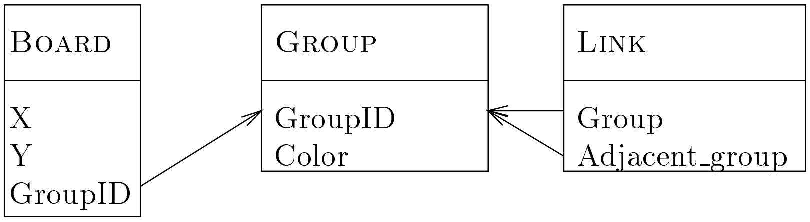


Figure 2: A relational representation

```
liberty(GroupID,Liberty) :- link(GroupID,Liberty), group(Liberty,empty).
liberty_cnt(GroupID,Nr_of_liberties) :-
     findall(Lib,liberty(GroupID,Lib),L),
     length(L,Nr_Of_liberties).
atari(GroupID) :- liberty_cnt(GroupID,1).
```

Figure 3: Some background predicates

solutions of the call $liberty(GroupID, Lib)$. Since $GroupID$ is instantiated at the moment of this call, all liberties of the group $GroupID$ are collected in the list L. The length of the list is then the number of liberties. Note that the liberty predicate is also defined for a group consisting of one empty intersection. A liberty of an empty intersection is an adjacent empty intersection.

Similarly other relevant predicates can be defined; Table 1 gives an overview of the ones we have used. In practice, some of these predicates have extra arguments that allow us to deal with invariance due to the symmetry of Go with respect to mirror, rotation and color swap, but for the sake of simplicity we make abstraction of this in this text.

It could now be hoped that a machine learning algorithm could learn simple concepts such as *double_atari* and *geta_2*, of which definitions are given in Figure 4 and examples in Figure 5 (the point labeled '1' is where black should move). Note that these definitions are short and natural: e.g., the definition for *geta_2* says: a group A of color $Color$ could be captured in a geta by playing at $(0,0)$ if it has only two liberties, and those liberties $L1$ and $L2$ are at $(1,0)$ and $(0,1)$, and playing on them does not connect to another group, i.e. $L3$ and $L4$ are empty. It would be very difficult to describe these concepts with propositional patterns.[1]

The representation of the data and the language bias presented here is not unique; e.g. the Link relation is redundant and could be computed from the Board relation. As usual, there is a trade-off between time and space complexity; our representation is chosen to have a good compromise between both.

[1] If this is tried anyway, several complexity problems arise which may make learning infeasible; [2] contains a thorough discussion of this.

Predicate	Meaning
liberty(G,L)	L is a liberty of the group G
liberty_cnt(G,N)	N is the number of liberties of G
stone_cnt(G,N)	N is the number of stones in the group G
group_on_pos(Pos,Displacement,G)	G is the group on the intersection Pos+Displacement
distance_to_edge(Pos,Dir,Dist)	Dist is the distance from Pos to the edge in the direction Dir
X < Y, X =< Y, X=Y	Comparison operators

Table 1: Background theory for Go problems

```
double_atari((X,Y),A,B) :-
     liberty_cnt(A,2), liberty_cnt(B,2),
     liberty(A,L),liberty(B,L), board(X,Y,L).
geta_2((X,Y),Color,A) :-
     group_on_pos((X,Y),(1,1),A),group(A,Color),
     liberty_cnt(A,2),liberty(A,L1),liberty(A,L2),
     group_on_pos((X,Y),(1,0),L1),
     group_on_pos((X,Y),(0,1),L2),
     group_on_pos((X,Y),(1,-1),L3),group(L3,empty),
     group_on_pos((X,Y),(-1,1),L4),group(L4,empty).
```

Figure 4: Definition of *double_atari* and *geta_2*

3 Experiments

This section describes some experimental results of our approach. For the experiments we used the inductive logic programming system TILDE [1] which is an extension of the well-known TDIDT algorithm [6] that learns first order logical decision trees (which can be transformed into executable Prolog programs). The learning task performed by TILDE could be specified as follows: given a set of labeled examples and a logical background theory, find a first order logical decision tree that together with the background theory assigns a label to unlabelled examples, minimizing some error function (e.g., the percentage of wrongly classified instances or the mean squared error). TILDE is able to produce both classification and regression trees, i.e., it can be used to induce Prolog programs that predict symbolic or numeric values.

In a first experiment, we illustrate the feasibility of this approach on a database of nakade shapes. A nakade shape is a big eyespace which can be reduced to one eye if the attacker plays on the vital point. Nakade shapes are important as elementary shapes in life- and death problems. Some nakade shapes are shown in Figure 6(a). In each case a triangled white stone is placed at the vital point.[2] We created a database of 166 nakade examples; each example consists of

[2] The square shape in the upper right is not a real nakade shape because it is dead whatever black does and therefore this shape has no vital point.

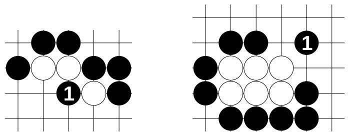
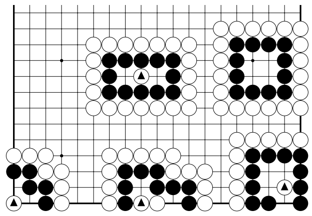


Figure 5: Example of *double_atari* and *geta_2*



```
move((X,Y)),board((X,Y),G),
libertycnt(G,Libs), Libs<2 ?
+--yes:group_on_pos((X,Y),(1,0),D),group(D,defender) ?
|       +--yes: group_on_pos((X,Y),(0,1),E),group(E,defender) ?
|       |       +--yes: negative
|       |       +--no: liberty(B,H,T) ?
|       |               +--yes: group(E,attacker) ?
|       |               |       +--yes: positive
|       |               |       +--no: negative
|       |               +--no: negative
|       +--no: negative
+--no: group_on_pos(G,(1,1),F),group(F,empty) ?
        +--yes: libertycnt(G,Libs), Libs=3 ?
        |       +--yes: positive
        |       +--no: negative
        +--no: positive
```

Figure 6: (a) some nakade shapes; (b) a tree describing vital moves for such shapes

a (shape,move) couple and is positive if the move is good (on the vital point), negative otherwise. A tenfold crossvalidation on this database yielded a predictive accuracy of 100%, confirming our expection that nakade forms are easy to learn using our representation. An inspection of the resulting trees (Figure 6(b) shows an example) showed that trees are simple and interpretable; e.g., the test in the top node is recognized as an important criterion by human Go players (the vital point is usually the one with the most liberties).

Finding vital points in nakade shapes is easy; more interesting are the tsume-go problems. A tsume-go problem consists of deciding whether a group of stones is alive (cannot be captured by the opponent if the player plays optimally) or dead (is bound to be captured by the opponent if the opponent plays optimally). The tsume-go database we use for these experiments contains 3600 tsume-go problems generated by the GoTools program [9], of which we use 2600 for training and 1000 for testing. Each problem contains a board situation and one or more solutions ordered from high to low quality. A solution contains the result (dead/alive) and a representative sequence of moves. The result info could allow us to determine if the task is to make a group alive or to kill an enemy group, but we do not use this information: the learned theory should determine the goal itself, as would be necessary in a real game where it is not known if it is possible to kill the enemy.

A problem can have several correct answers, which all solve the problem (live or kill) but perhaps differ a little (e.g. in endgame possibilities). In that case the answers are awarded 1, 0.9, 0.8, ... points in descending order of optimality. Wrong moves are awarded 0 points. Because in the original database certain kinds

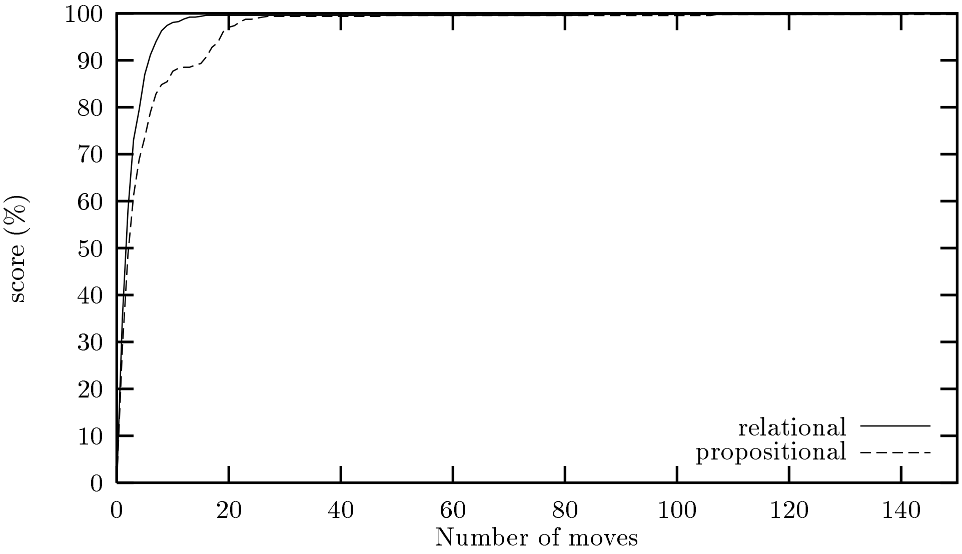


Figure 7: Curves learned on the full training set

of obviously bad moves (e.g., putting a stone in a totally different area of the board than where the action is going one) are never mentioned, which makes it hard to learn that this kind of moves should never be played. Therefore we added to the database for each problem a random move and gave it reward 0.

Running TILDE in regression mode on the 2600 training examples produces a logical decision tree that can be used to assign expected rewards to candidate moves in new situations. We evaluate our results as follows. The induced tree is used to predict the expected reward of all possible moves in the 1000 test problems. These moves are ordered from high to low expected reward. Then, a curve is drawn giving for each number n the total effective reward (given in the answers of the test problems) of the n first moves from the heuristic ordering in each problem.

Using this evaluation criterion, we compared the performance of the trees for two different settings. In the first setting, only propositional patterns are considered. This means we allow only propositional tests of the form $group\_on\_pos(Move, displacement, Group)$, $group(Group, color)$ in the tree where $displacement$ and $color$ are constants. This approach generates rules similar to those considered in [4]. In the second setting, we use a relational language. All relations and background predicates given in Table 1 are allowed.

Several experiments were run, varying e.g. the size of the data set. Figure 7 shows the results when learning on the full training set; it is clear that the propositional setting performs worse than the relational setting. A similar result was obtained for the other experiments.

For the relational tree learned on the full training set, every good move is found on average on position 2.44 (this also applies when there are several good moves), which suggests that the quality of our learned heuristic is comparable to the handcrafted heuristic in GoTools (see [9]).

System	Testset(size)	1	2	3	4	5
Propositional decision tree	Gotools(1000)	29%	49%	61%	69%	73%
Relational decision tree	Gotools(1000)	35%	58%	73%	79%	87%
Flexible rules with Weights	Basic(100)	36%	51%	63%	73%	79%
Flexible rules with Weights	3 dan(100)	31%	57%	67%	74%	80%
Flexible rules with Weights	5 dan(100)	26%	46%	55%	69%	77%
Neural network	Subset Basic(61)	41%	60%	63%	68%	75%
Neural network	Subset 3 dan(43)	27%	56%	79%	81%	83%
Neural network	Subset 5 dan(47)	21%	43%	55%	60%	62%
Neural network	A-E (1000)	35%	50%	59%	65%	69%

Table 2: Score of some systems for the first five moves

[4] reports having found 79% of all good moves after five moves in the heuristic ordering, also for 'basic' problems, using weighted patterns; [7] reports a similar performance on a subset of the same dataset with neural networks. We summarize their and our results in Table 2. The table reconfirms that relational learning works better than propositional learning and suggests that our results (87% after five moves) are at least comparable to the results mentioned there.

Besides these percentages, we also want to estimate the effect of these results on the performance of computer Go players. The complexity of searching in a minimax-tree b^k is exponential in the depth k of the tree, with b the average effective branching factor. While the exponentiality cannot be avoided, optimizations such as alpha-beta pruning and a heuristic can reduce the value of b.

How good are our results in terms of reducing this average branching factor? A more detailed study of our experimental results shows that if only the best 5 moves are examined, 10% of the good moves are missed; when examining the best 10 moves this goes to 1%. Comparing this with about 150 available moves on the 13x13 problem boards or 30 moves available inside the tsume-go problem, it is obvious that good branching factor reductions become possible. In addition, one can reorder the branches according to their expected value. It is known that for perfect ordering an effective average branching factor of $\sqrt{b}$ is obtained when b is the original branching factor. In the case of non-perfect ordering, it is possible to derive an upper bound on the effective branching factor based on the probability of the best move occurring in a certain position. Due to space restrictions we cannot include this analysis here, but the results of the analysis on our dataset are that the average branching factor can be further reduced from 10 (5) to 5.7 (3.6) moves. For the propositional approach the corresponding numbers are 25 (16) to 12.4 (8.4), which gives an idea of the actual difference in performance between the relational heuristic and the propositional one.

4 Conclusions and further work

We have presented the application of the logical decision tree learner TILDE on Go and discussed the relation to other learning algorithms. We have argued that it is useful to use a relational representation language to describe the learned concepts, and presented experiments comparing this approach with existing propositional ones both in terms of accuracy and in terms of alpha-beta search complexity. The results support our claim that more expressive representations are superior to propositional ones. Moreover, the performance of our learning technique is comparable to that of other learning approaches and to handcrafted heuristics.

For further work we see several lines of possible research. First, the heuristic could be extended to be able to give predictions for all moves in a tsume-go sequence, including the final stage of effectively capturing or timely defending when dames are filled. This would enhance the usefulness of it in a real playing program. Next, the same techniques could also be tried in other parts of the game, such as endgame or opening patterns.

Acknowledgements

Jan Ramon is supported by the Flemish Institute for the Promotion of Science and Technological Research in Industry (IWT). Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders. We thank Thomas Wolf for making available his dataset of tsume-go problems.

References

[1] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.

[2] L. De Raedt. Attribute-value learning versus inductive logic programming: the missing links. In *Proc. 8th Int'l Conf. on Inductive Logic Programming*, pages 1–8. Springer-Verlag, 1998.

[3] M. Enzenberger. Integration of a priori knowledge into a go playing neural network, 1996. `http://home.t-online.de/home/markus.enzenberger/neurogo.html`.

[4] T. Kojima and A. Yoshikawa. A two-step model of pattern acquisition: Application to tsume-go. In *Proc. 1st Int'l Conf. on Computers and Games*, pages 146–166. Springer-Verlag, 1998.

[5] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of inductive logic programming*. Springer-Verlag Inc., New York, NY, USA, 1997.

[6] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[7] N. Sasaki, Y. Sawada, and J. Yoshimura. A neural network program of tsume-go. In *Proc. 1st Int'l Conf. on Computers and Games*, pages 167–182. Springer-Verlag, 1998.

[8] D. Stoutamire. Machine learning applied to go. Master's thesis, Case Western Reserve University, 1991.

[9] T. Wolf. The program GoTools and its computer-generated tsume go database. Report, School of Mathematical Sciences, Mile End Road London E1 4NS, November 1996. http://www.qmw.ac.uk/˜ugah006/gotools/.