Learning to Solve Game Trees

David Stern DHS26@CAM.AC.UK
Cambridge University, Cambridge, UK

Ralf Herbrich RHERB@MICROSOFT.COM
Thore Graepel THOREG@MICROSOFT.COM
Microsoft Research Ltd., Cambridge, UK

Abstract

We apply probability theory to the task of proving whether a goal can be achieved by a player in an adversarial game. Such problems are solved by searching the game tree. We view this tree as a graphical model which yields a distribution over the (Boolean) outcome of the search before it terminates. Experiments show that a best-first search algorithm guided by this distribution explores a similar number of nodes as Proof-Number Search to solve Go problems. Knowledge is incorporated into search by using domain-specific models to provide prior distributions over the values of leaf nodes of the game tree. These are surrogate for the unexplored parts of the tree. The parameters of these models can be learned from previous search trees. Experiments on Go show that the speed of problem solving can be increased by orders of magnitude by this technique but care must be taken to avoid over-fitting.

1. Introduction

We address the issue of proving whether goals can be achieved by a player in an adversarial game. The task is to prove that a player (the 'attacker') can achieve the goal whatever the actions of the opponent (the 'defender'). Such problems are solved by searching the state space (the game tree) (Pearl, 1984; Russell & Norvig, 1995).

We assume every node in the game tree has an underlying 'Delphic' value: the Boolean value that would be returned by an oracle with perfect knowledge (Palay, 1985). This value is TRUE for a node if the goal can be provably achieved in the corresponding position and FALSE if the goal cannot be achieved. After fully exploring the tree we can determine by logical deduction the value of the root node. In this case the tree is 'solved' and search terminates. During a search the values of some nodes are not yet determined and we can quantify the uncertainty about these values using probabilities. We assign a probability of 1 to every TRUE node and probability 0 to every FALSE node. All other nodes have some probability between 0 and 1 which represents a degree of belief about whether the node is TRUE (see Section 3).

By placing prior distributions on the values of the leaf nodes of the game tree we can incorporate knowledge into search. These distributions are surrogate for the unexplored parts of the tree. As searches are performed, nodes become proved as TRUE or FALSE and these proofs can be used to update the surrogate distributions so future searches are more efficient.

A number of approaches using probability distributions to guide search in games have been suggested (Palay, 1985; Baum & Smith, 1997; Russell & Wefald, 1991). In these cases distributions are used to model the uncertainty in the real-numbered value of game states. In this work we are concerned with binary (WIN/LOSS) games for which the semantics of a real-numbered state-value are unclear so we assume the underlying Delphic value of a node can only be TRUE or FALSE.

Many individual nodes must be solved recursively in order to solve a game tree so each node represents a search problem in its own right. This means that a complex problem provides a rich source of information about problem solving in general. In Section 4 we apply these ideas to the game of Go. A Go position in conjunction with the rules of the game contains all the information necessary for perfect play but if we have limited computational resources we must take care to

Appearing in *Proceedings of the 24th International Conference on Machine Learning*, Corvallis, OR, 2007. Copyright 2007 by the author(s)/owner(s).

extract only relevant information. The game tree gives the structure required to extract this knowledge.

This type of supervised learning is unusual because it is the agent itself generating the observations. However, as long as we make our inferences based entirely on our probabilistic model (the search tree) and on which nodes are observed to be TRUE or FALSE we receive the full protection of the *likelihood principle* (MacKay, 2003): it is not possible to bias our models by the fact we are selectively exploring the state space because all proofs and disproofs we observe are objective facts about the domain.

2. Search in Games

2.1. AND / OR Trees[1]

Let the set of possible positions in a game be $\mathcal{N}$. A problem is defined by its 'goal', $g : \mathcal{N} \rightarrow$ {TRUE, FALSE, UNKNOWN}. Each position $n \in \mathcal{N}$ has a set of legal successor positions, $\mathcal{L}(n)$, each of which can be generated by an action of a player (a move). Two players, 'attacker' and 'defender', take it in turns to move. We are concerned with proving whether the attacker can reach a state in which the goal is TRUE taking into account all possible actions of 'defender'. The (Boolean) result of this proof is the Delphic value of the node and is denoted $d(n)$ where $d : \mathcal{N} \rightarrow \{\text{TRUE}, \text{FALSE}\}$.

Starting at a root position $r \in \mathcal{N}$ we 'develop' it by generating each legal successor position (its 'children'). In this way we begin to generate a search tree, $\mathcal{T} := \{\mathcal{N}, \mathcal{E}\}$, which represents possible (directed) paths through state space. Each edge $e \in \mathcal{E}$ corresponds to a transition between states (a move). We refer to the set of children of a node n as $\text{ch}(n)$ and the parent of a node c as $\text{pa}(c)$. Once a position is developed it is called an 'internal' node otherwise it is a 'leaf' node. Leaf nodes, l, where $g(l)$ is TRUE or FALSE are called 'terminal' nodes. We iterate the process of developing non-terminal leaf positions to expand the search tree.

The values of previously explored paths through state space are represented as an AND/OR Tree (AOT) (Nilsson, 1971). Each node $n \in \mathcal{N}$ in the AOT is labelled with a value $v(n) \in$ {TRUE, FALSE, UNKNOWN}. If $v(n)$ is TRUE or FALSE then node n is 'solved' and $v(n) = d(n)$. An AOT has two types of nodes: OR nodes and AND

[1]We use the common convention of referring to the search graph as a 'tree'. In fact a game corresponds to a directed acyclic graph because it is possible for the same state to be reached via different paths.

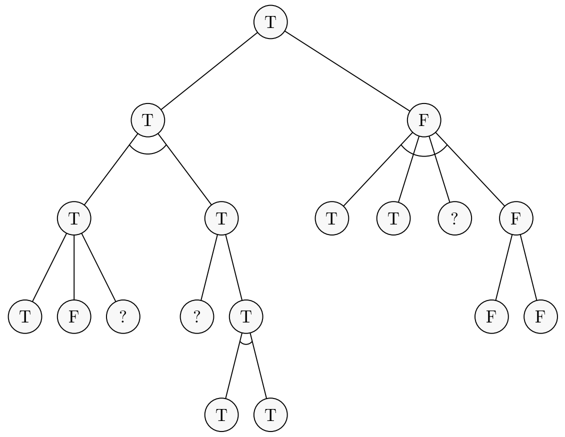


Figure 1. And / Or Tree with truth values of nodes labelled. The arcs underneath some of the nodes indicate that they are AND nodes. The other nodes are OR nodes.

nodes. For a given tree with values assigned to the leaf nodes we determine the values of the internal nodes by:

$$\text{AND node: } v(n) = \bigwedge_{c \in \text{ch}(n)} v(c)$$

$$\text{OR node: } v(n) = \bigvee_{c \in \text{ch}(n)} v(c).$$

The AND operator ($\wedge$) is defined such that if any child of a node is FALSE then the node is FALSE, otherwise if any child is UNKNOWN then the node is UNKNOWN, otherwise it is TRUE. The OR operator ($\vee$) is defined such that if any child of a node is TRUE then the node is TRUE, otherwise if any child is UNKNOWN the node is UNKNOWN, otherwise it is FALSE; see Figure 1 for an example tree.

Each AND node corresponds to a position in which it is the defender's turn to move (because every defender response must be considered to prove that the goal can be achieved). Each OR node corresponds to a position in which it is the attacker's turn to move (because only one working attacker move must be found in each position along the path to the solution). This scheme is equivalent to the minimax algorithm with a binary valued evaluation function (Russell & Norvig, 1995).

If the root has value TRUE or FALSE then the tree is 'solved' and the value of the tree is the value of its root. If a tree has value TRUE it is 'proved', if it has value FALSE it is 'disproved'. If no children can be added to a leaf node (because no legal moves are available) then it has value FALSE if it is an AND node and TRUE if it is an OR node.

In this work we focus on best-first search. At each step in a best-first search the most promising node

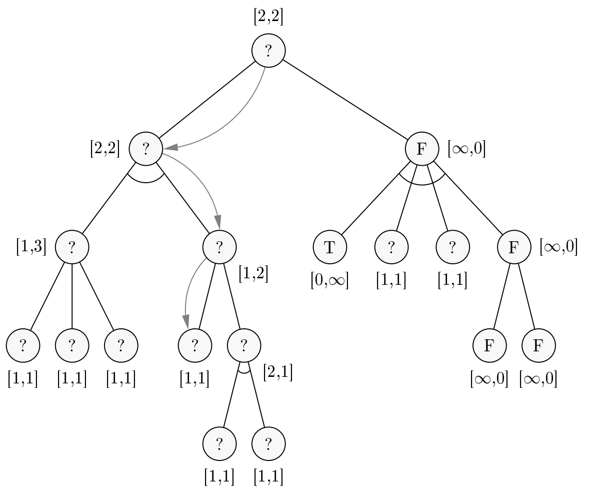


Figure 2. And / Or Tree with proof and disproof numbers labelled as [PN,DN]. The path to the most proving leaf is shown.

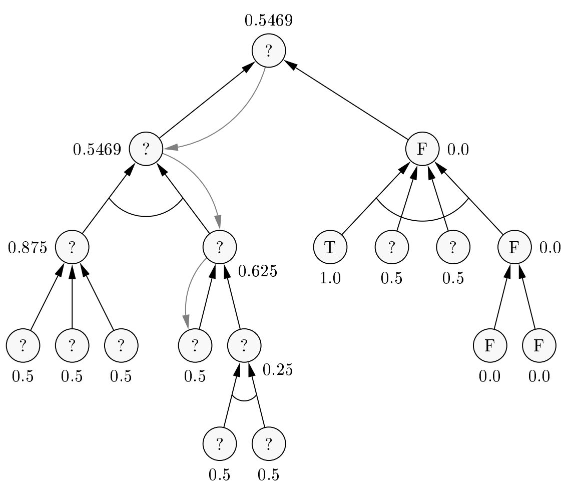


Figure 3. Search tree as Bayesian network. Each node is labelled with its probability of being TRUE. The estimated path of best play is also labelled - notice it is the same as the path followed by PNS (Figure 2).

(according to some criteria) is developed. This is in contrast to depth-first search where the search tree is enumerated up to some fixed depth. In practice depth first search has proved much more successful in game playing applications because of the difficulty of move selection. However, using depth as the criterion for terminating search may result in a great deal of wasted computational effort by not concentrating on important lines of play. Both depth-first and best-first methods suffer from the *horizon effect*: important lines of play may be terminated before they are played out leading to a poor estimation of the value of the root (Palay, 1985).

2.2. Go

AOTs can be used to describe problems in the game of Go[2]. Go is an ancient oriental board game of two players, 'Black' and 'White' (Müller, 2002). The players take turns to place *stones* on the intersections of a grid with the aim of making territory by surrounding areas of the board. All the stones of each player are identical. Once placed, a stone is not moved but may be captured (by being surrounded with opponent stones). We focus on the task of solving a class of Go problems called *tesuji*[3] problems where the goal in each case is to capture a particular stone on the board (Davies, 1975) (Figure 7). For these problems $g(n)$ is TRUE if the goal vertex is empty in position n.

[2] A great deal of information about Go can be found at http://www.gobase.org.

[3]A *tesuji* is the best play in a certain local position. These moves have names such as the 'net', the 'ladder', the 'cranes nest' etc. Tesuji problems are used in teaching a player these standard plays.

2.3. Proof Number Search

We compare our techniques to Proof Number search (PNS) (Allis, 1994), a state-of-the-art best-first search algorithm for finding solutions to problems represented as AOTs. Two numbers are assigned to each node: the proof number (PN) and the disproof number (DN). The PN of a node is defined as the minimum number of nodes that must be developed in order to prove that node.

$$\mathrm{PN}_n = \begin{cases} \sum_{c\in \mathrm{ch}(n)} \mathrm{PN}_c & \text{if internal AND node,} \\ \min_{c\in \mathrm{ch}(n)} \mathrm{PN}_c & \text{if internal OR node,} \\ 0 & \text{if } g(n) = \text{TRUE,} \\ \infty & \text{if } g(n) = \text{FALSE.} \\ 1 & \text{if UNKNOWN leaf node} \end{cases}$$

By symmetry the rules for propagating DNs are the same as the rules for PNs if we exchange OR for AND and TRUE for FALSE. Figure 2 shows an AOT with proof and disproof numbers labelled. Given a search tree the next node to develop is determined by working down the tree from the root, selecting the child with the lowest PN at each OR node and the child with the lowest DN at each AND node. Once a leaf is reached it is developed and then the PNs and DNs are propagated up to the root. This process is repeated until the tree is solved.

3. Search and Inference

Probability propagation (PP) applies the rules of probability to calculating a belief distribution over the Delphic values of nodes in the tree. We assume that the

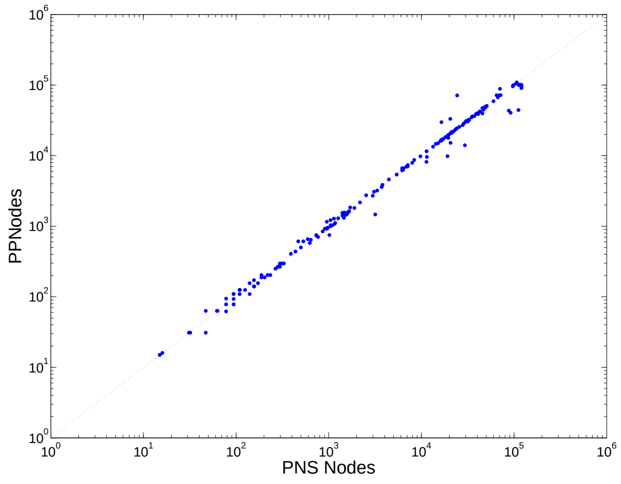


Figure 4. Comparing the number of nodes developed in order to solve a set of tesuji Go problems.

value of each child of a node is distributed independently of the states of its siblings. This assumption may frequently be violated (for example in a game tree the values of siblings are likely to be correlated as once the player is in a strong position there are likely to be many good moves available). This seems more likely to be a problem for global (strategic) search of the game tree rather than local, tactical search, to which the ideas presented here seem more applicable.

For each node, n, we store the probability of it being TRUE: $P_n := P(d(n) = \text{TRUE}), n \in \mathcal{N}$. If a node has value FALSE the probability $P_n = 0$, if it has value TRUE then $P_n = 1$. If the node is UNKNOWN then the probability represents our degree of belief about the value of the node being TRUE. Inference is achieved by simple propagation rules (Pearl, 1984; Chi & Nau, 1988):

$$\text{AND}: P_n = P\left(\bigwedge_{c\in \text{ch}(n)} d(c)\right) = \prod_{c\in \text{ch}(n)} P_c \quad (1)$$

$$\begin{aligned}\text{OR}: P_n &= P\left(\bigvee_{c\in \text{ch}(n)} d(c)\right) = P\left(\neg \bigwedge_{c\in \text{ch}(c)} \neg d(c)\right) \\ &= 1 - \prod_{c\in \text{ch}(c)} (1 - P_c). \quad (2)\end{aligned}$$

The Bayesian network for the model is shown in Figure 3. The joint distribution of the Delphic values of all nodes in the game tree is:

$$P(\mathcal{N}) = \prod_{n\in\mathcal{N}\setminus\mathcal{F}} P\left(d(n)|\left\{d(c)\right\}_{c\in \text{ch}(n)}\right) \prod_{l\in\mathcal{F}} P(d(l)) \quad (3)$$

where $\mathcal{F}$ is the set of leaves (the search *frontier*). For an AND node, $P(d(n)|\{d(c)\}_{c\in\text{ch}(n)}) = \mathbb{I}(d(n) = \bigwedge_{c\in\text{ch}(n)} d(c))$ and for an OR node $P(d(n)|\{d(c)\}_{c\in\text{ch}(n)}) = \mathbb{I}(d(n) = \bigvee_{c\in\text{ch}(n)} d(c))$. For TRUE or FALSE nodes the priors on the leaf values, $P(d(l))$ are set to 1 or 0 respectively. For UNKNOWN leaves the priors represent our prior belief about whether the node is TRUE or FALSE (set to 0.5 in initial experiments).

The Algorithm The game tree is explored by the best-first search procedure described in the algorithm boxes. At each step the best node to expand is selected by starting at the top of the tree and working downwards following the path of best play (according to current beliefs). There are two ways in which new observations can change the planned sequence of actions - by reducing the value of the current plan or by increasing the value other actions so as to make them preferable (Russell & Wefald, 1991). Our method explores the first of these possibilities.

To implement the propagation rules we represent the probabilities as log-odds ratios, logodds$(p) := L(p) := \ln(\frac{p}{1-p})$. This uses the full floating point range to represent $L(p)$, with high precision at both ends of the range (corresponding to probabilities close to 1 or 0). The log-probability domain is not suitable as it has poor accuracy for probabilities close to 1 which are readily generated by the OR rule in large problems.

Algorithm 1 FindBestNode(n)

```
if $n$ is leaf then
  return $n$
else if $n$ is AND node then
  return FindBestNode($\text{argmin}_{c\in\text{ch}(n)}\{P_c\}$)
else
  return FindBestNode($\text{argmax}_{c\in\text{ch}(n)}\{P_c\}$)
end if
```

Algorithm 2 UpdateBeliefs(n)

```
if $n$ is AND node then
  Calculate $P_n$ via (1)
else
  Calculate $P_n$ via (2)
end if
UpdateBeliefs(pa($n$))
```

Experiments (See Section 5.1 and Figure 4) show that PP and PNS must expand roughly the same number of nodes to solve Go problems. Comparing figures 2 and 3 it can be seen that PP and PNS are similar strategies. Both methods avoid exploring branches of the

Algorithm 3 Develop(n)

```
ch($n$) := $\mathcal{L}(n)$
for all $c \in$ ch($n$) do
  if $g(c)$ TRUE then
    $P_c := 1.0$
  else if $g(c)$ FALSE then
    $P_c := 0.0$
  else
    $P_c := p(d(c))$ (prior)
  end if
end for
```

Algorithm 4 Search

```
while $p_{\text{root}} < 1.0$ do
  $n$ = FindBestNode(root)
  Develop($n$)
  UpdateBeliefs($pa(n)$)
end while
```

tree leading to AND nodes with many children (due to the fact that a proof of such a branch would involve proving more nodes in total). PP differs from PNS in that it has an affinity for developing OR nodes with many children (due to the fact that each child of an OR node represents an independent additional chance of finding a proof of the parent). That is, PP tends to explore parts of the search tree where *player* has more moves available and *opponent* has fewer moves available. Thus PP seems to recover an intuitive heuristic: mobility.

4. Search and Knowledge

Search is a process of observation. The search algorithm is initialised with some prior beliefs about the leafs of the tree. These prior distributions are surrogate for the as-yet unexplored parts of the tree. There is a trade-off between search and knowledge: the more accurate the prior beliefs the fewer states the searcher must explore to find a proof. When search terminates the parameters each surrogate prior distribution, $p(d(l))$ can be updated according to the final value of the node, $d(l)$, determined by the search. If the vector $\mathbf{l}$ denotes all of the nodes in all of the searches then the joint likelihood is given by $p(d(\mathbf{l})|\mathbf{q}) = \prod_i p(d(l_i)|\mathbf{q})$. The parameters $\mathbf{q}$ are shared across nodes (depending on which common *patterns* match) so generalisation across different positions and search tasks is possible.

4.1. Pattern Matching

Exact local pattern matching gives a rapid and surprisingly accurate Go move predictor (Stern et al., 2006). In this paper we apply this technique to represent knowledge in search. A pattern is defined as the exact arrangement of stones in a sub-region of the board centred on the empty location of the board where a move is to be made. We define a set of fixed nested 'templates' $T_i \in \mathcal{T}$ (Figure 5). Each template T_i is a mask which determines the sub-region of the board within which the arrangement of stones (of size i) must match for the pattern to be present. Size 1 is the smallest template (just the point at which the move is made) and Size 7 is the largest. Therefore each move in a Go position corresponds to a stack of 8 patterns of nested sizes. Each pattern maps to an efficiently generated hash key such that the patterns are invariant to the 8-fold symmetry of the square. In contrast to our earlier work, here each pattern vertex has five states (attacker stone, defender stone, empty vertex, off-board, goal stone) and the patterns are not invariant to colour reversal.

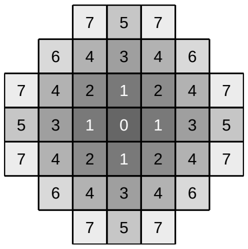
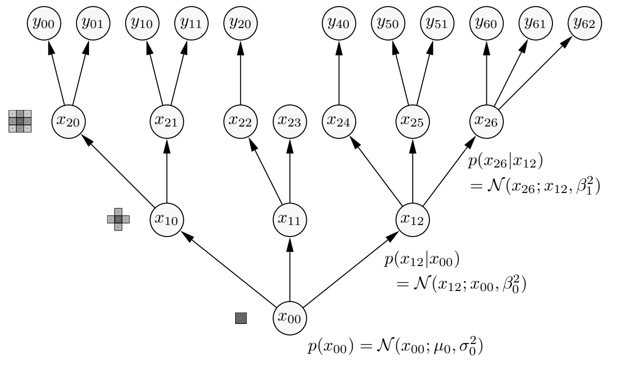


Figure 5. The sequence of nested pattern templates T_i with $i \in \{0, \ldots, 7\}$.

Nodes (positions) are mapped to patterns via the move which generated the node. Let the stack of all patterns which match for a node n be denoted by $\boldsymbol{\pi}(n)$. Each pattern defines a many-to-one mapping from search tree nodes to a look-up table, H, via the hash key. This table can be viewed as a partial-transposition table. A transposition table (TP) is a tool used in most practical game search implementations which contains the values of all board positions that previously appeared in the search so if a position is encountered again the information already gathered about it can be exploited (Plaat et al., 1986). In this work we do not map from positions to TP entries but instead from *patterns* (partial positions) to table entries. This partial matching allows generalisation across different search tasks which is bought at the cost of uncertainty - hence the entries in the partial-TP are probability distributions.

When a node, n, is developed and its children $c \in \mathcal{L}(n)$ are added to the tree then the patterns for each pattern template $T \in \mathcal{T}$ centred on the moves which generate these children are *harvested,* i.e. added to H.



y_{00} y_{01} y_{10} y_{11} y_{20} y_{40} y_{50} y_{51} y_{60} y_{61} y_{62}
x_{20} x_{21} x_{22} x_{23} x_{24} x_{25} x_{26}
$p(x_{26}|x_{12}) = \mathcal{N}(x_{26}; x_{12}, \beta_1^2)$
x_{10} x_{11} x_{12}
$p(x_{12}|x_{00}) = \mathcal{N}(x_{12}; x_{00}, \beta_0^2)$
x_{00}
$p(x_{00}) = \mathcal{N}(x_{00}; \mu_0, \sigma_0^2)$

Figure 6. Hierarchical Pattern Model. In this case there are 11 patterns in total in the hierarchy in 3 levels. The full system has 8 levels in the hierarchy. The diagram shows 11 observations.

4.2. Surrogate Tree Models

Beta Model Let $\hat{\pi}(l)$ denote the largest pattern (which has been observed at least once before) matching for node l. The prior distribution on the value of a leaf node, $p(d(l))$, is distributed according to a Bernoulli distribution, $p(d(l)|q_{\hat{\pi}(l)})=\text{Ber}(d(l); q_{\hat{\pi}(l)})$. The parameter q represents a prior belief about the node being TRUE. We place a Beta prior on $q_{\hat{\pi}(l)}$, $p(q_{\hat{\pi}(l)})=\text{Beta}(q_{\hat{\pi}(l)}; \alpha_{\hat{\pi}(l)}, \beta_{\hat{\pi}(l)})$. This gives the predictive distribution $p(d(l)|\alpha_{\hat{\pi}(l)}, \beta_{\hat{\pi}(l)}) = \frac{\alpha_{\hat{\pi}(l)}}{\alpha_{\hat{\pi}(l)}+\beta_{\hat{\pi}(l)}}$ and the parameters $\alpha_{\hat{\pi}(l)}$ and $\beta_{\hat{\pi}(l)}$ are *pseudo-counts* corresponding to the number of observed proofs and disproofs respectively of all nodes where $\hat{\pi}(l)$ is found to match.

After search termination, the posterior distribution over the parameter $p(q_l|d(l))$ is $\text{Beta}(q_l; \alpha'_{\hat{\pi}(l)}, \beta'_{\hat{\pi}(l)})$ with $\alpha' = \alpha + 1$ if $d(l) =$ TRUE and $\beta' = \beta + 1$ if $d(l) =$ FALSE. This update is applied for *all* elements of $\boldsymbol{\pi}(n)$ for each solved node, n, in the tree.

Hierarchical Gaussian Model We also considered a model which takes account of the entire stack of patterns that match at a vertex. Intuitively the evidence provided by a larger pattern should dominate over the evidence from a smaller pattern at the same location because the larger pattern contains all the information of the smaller pattern plus additional information. However, the smaller patterns should be allowed to influence the value of the larger patterns in cases where the larger patterns have been seen infrequently.

First we define a hierarchical model of the value of Go moves. Let the set of the values of all the (infinite) possible observations of all possible Go moves in all possible positions be $\mathcal{Y}$. Also, let the set of (latent) values of all possible patterns of all sizes be $\mathcal{X}$. Each member of $\mathcal{X}$ shall be denoted x_{ij} where x_{ij} is the value of the jth pattern of size i. The index of the smallest pattern size is 0 so the value of the smallest (zero sized) pattern is x_{00}. The largest pattern size is m (in the experiments here $m = 7$). An observed value, $y_{jk} \in \mathcal{Y}$, is the kth observation of the jth full board position. The joint distribution is:

$$\begin{aligned} p(\mathcal{Y}, \mathcal{X}) &= p(\mathcal{X}) \cdot p(\mathcal{Y}|\mathcal{X}) \\ &= p(x_{00}) \prod_{h=1}^{m} \prod_{k} \prod_{j} p(x_{hj}|x_{(h-1)k}) \\ &\quad \ldots \cdot \prod_{q} \prod_{i} p(y_{qi}|x_{nq}) \end{aligned}$$

where $p(x_{ij}|x_{(i-1)j}) = \mathcal{N}(x_{ij}; x_{(i-1)j}, \beta_{i-1}^2)$, $p(y_{ij}|x_{ni}) = \mathcal{N}(y_{ij}; x_{ni}, 1)$ and $p(x_{00}) = \mathcal{N}(x_{00}; \mu_0, \sigma_0^2)$. Figure 6 shows a corresponding graphical model with $m = 3$. The variance parameters, β_i^2, correspond to the variability of the latent value of patterns of sizes $i + 1$ about the value of the size i pattern that also matches and are estimated empirically. The predictive (Gaussian) distribution over the move-value y_{ij} is determined by belief propagation (MacKay, 2003).

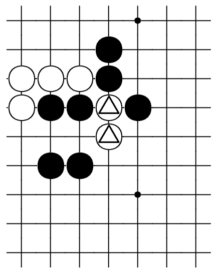
In order to use this model as a prior over the probability of a leaf node, n, being TRUE in the search tree we introduce the conditional distribution $p(d(n)|y) = f_{\text{switch}}(d(n), y) = \mathbb{I}((y > 0) \wedge d(n)) + \mathbb{I}((y < 0) \wedge \neg d(n))$ (we observe the constraint that a TRUE node has positive value and a FALSE node has negative value). This corresponds to letting the probability of a node being proved TRUE be the area under the positive orthant of the Gaussian belief: $p(d(n)|\mu, \sigma) = \text{Ber}(d(n), 1 - \Phi(0; \mu, \sigma^2))$ for $p(y) = \mathcal{N}(y; \mu, \sigma)$. After search termination, for a node with value $d(n)$ and a belief from the hierarchical model of $\mathcal{N}(y; \mu, \sigma^2)$ the update is:

$$p(y|d(n)) = \frac{f_{\text{switch}}(d(n), y) \cdot \mathcal{N}(y; \mu, \sigma^2)}{Z(\mu, \sigma^2, d(n))}.$$

This is non-Gaussian so we approximate it by the Gaussian closest in terms of KL divergence. Following this approximation inference is by belief propagation.

5. Experiments: PNS vs PP

PNS and PP are applied to the task of solving a set of Go problems which we know in advance all have a TRUE solution. A disproof criterion was defined based on the liberty count of the goal stone. The number of *liberties* of a stone is the minimum number of stones that must be played to capture it. The goal function



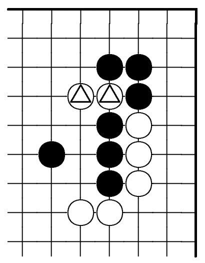


Figure 7. Example tesuji capture problems (Davies, 1975). The goal is to capture the stones marked with a triangle.

was set as

$$g(n) = \begin{cases} \text{TRUE} & \text{if goal vertex empty,} \\ \text{TRUE} & \text{if a ladder works,} \\ \text{FALSE} & \text{if liberties of goal } > L, \\ \text{UNKNOWN} & \text{otherwise} \end{cases}$$

If search terminates with value FALSE then L is incremented and the search repeated. This process is repeated until the search terminates with value TRUE.

5.1. No Knowledge

Firstly the performance of PNS and PP with an ignorant prior ($p(d(l)) = 0.5, l \in \mathcal{F}$) are compared on a set of 192 Go capture problems[4] (see Figure 4). In most problems the two algorithms perform similarly as discussed in Section 3.

5.2. Learning

In a second set of experiments we include additional problems that could not be solved by PNS. These are randomly divided into a training set (289 problems) and a test set (145 problems). The pattern table, H, is initially empty. During learning the searcher harvests patterns continuously. The prior distribution for each new node is assigned using one of the surrogate models described in Section 4.2, using the patterns to determine the parameters of the models. Learning is carried out by updating the posteriors over the parameters of the surrogate models after search termination for moves leading to TRUE and FALSE nodes.

A time-out of 120s is set for each problem. The searcher iterates over the training set re-attempting problems that previously failed until as many problems as possible are solved. The result plots are generated by timing the solver on the test set and comparing these times with the time taken by PNS. Time is used for comparison so additional cost associated with pattern matching and inference is taken account of. Problems that are solved after learning but could not be solved by PNS are labelled as crosses on the plots. Problems which could be solved by PNS but not PP are labelled as circles on the plots. The dots correspond to problems that could be solved by both algorithms. Problems which could be solved by neither algorithm are omitted.

The Beta model is tested (Figure 8 top) with two different initial settings for the prior parameters (α and β). Learning improves the speed of problem solving, sometimes by orders of magnitude. However, there are a number of problems which fail after learning but were able to be solved before learning suggesting over-fitting. This problem is ameliorated by using a peaked prior (Figure 8 right). The hierarchical model performs somewhat better with less over-fitting (Figure 8 bottom). The left plot corresponds to an initial setting of zero previous observations. The plot on the right was generated by assigning prior beliefs as if a number of previous observations of $p(d(l)) = 0.5$ had been made (compare with initialising the pseudo counts of a Beta distribution).

[4]These are also available at http://t-t.dk/madlab thanks to Thomas Thomsen.

6. Conclusions

Probability propagation as a means to solve Go problems appears to perform similarly to best-first proof-number search. Experiments suggest it is possible to learn from previous searches how to search faster by orders of magnitude. However, great care must be taken to prevent over-fitting.

Acknowledgements

We would like to thank David MacKay, Tom Minka and Thomas Thomsen for interesting discussions. This work was supported by a grant from Microsoft Research.

References

Allis, V. L. (1994). *Searching for solutions in games and artificial intelligence.* Doctoral dissertation, University of Limburg.

Baum, E. B., & Smith, W. D. (1997). A bayesian approach to relevance in game playing. *Artificial Intelligence.*

Chi, P., & Nau, D. (1988). Comparison of the minimax and product back-up rules in a variety of games. *Search in Artificial Intelligence.*

Davies, J. (1975). *Tesuji.* Kiseido Publishing Company.

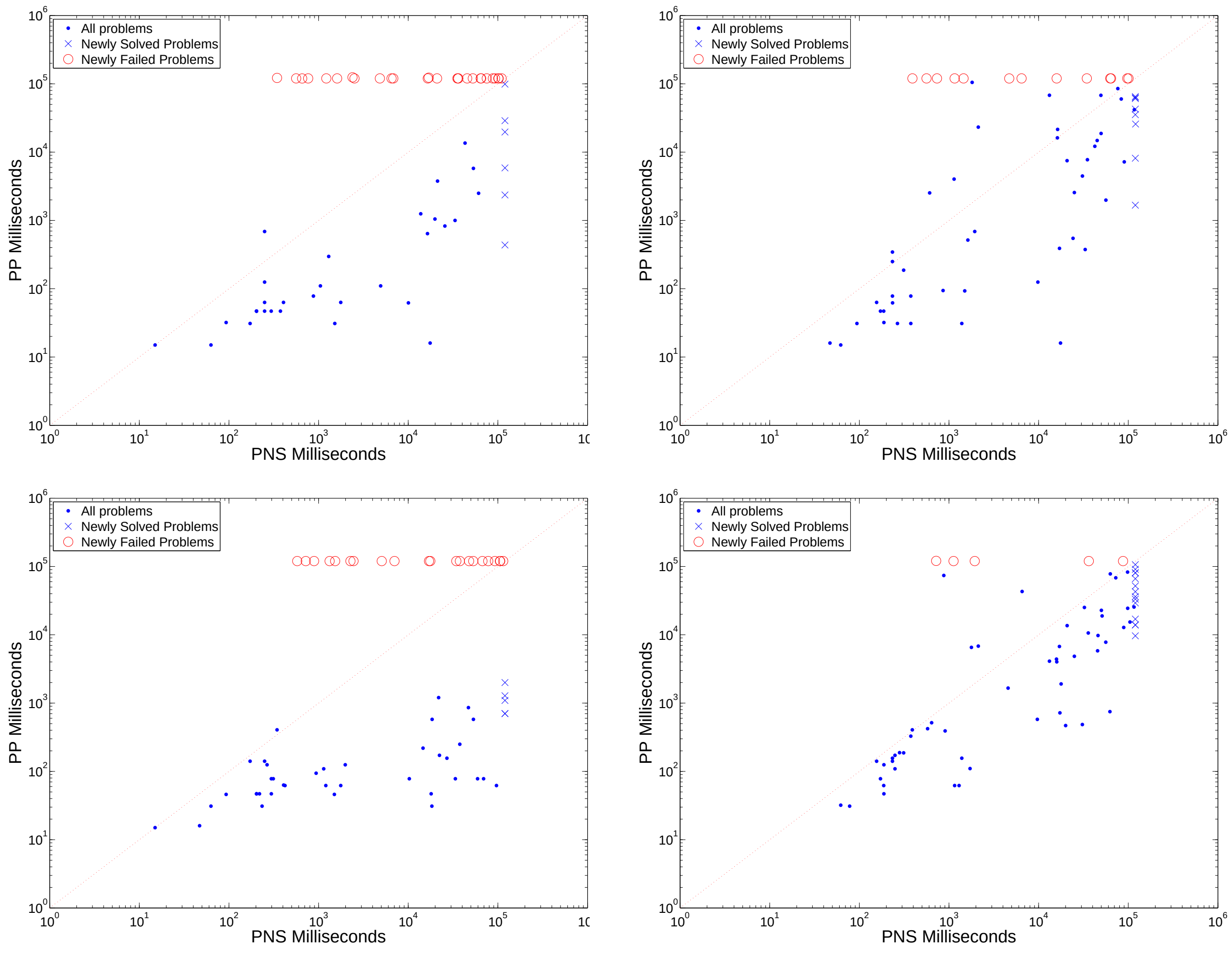


Figure 8. Time taken to solve Go problems - PNS vs PP. **Top Left**: Beta model. prior $\alpha_0 = 1, \beta_0 = 1$ (27 problems newly failed, 6 problems newly solved, mean speedup factor: 45.9); **Top Right**: Beta model, prior $\alpha_0 = 1000, \beta_0 = 1000$ (13 problems newly failed, 8 newly solved, speed, mean speedup factor: 30.7). **Bottom Left**: Hierarchical , uniform prior (21 newly failed, 5 newly solved, mean speedup factor: 135.5); **Bottom Right**: Hierarchical model, prior hand tuned to reduce learning rate (5 newly failed, 14 newly solved, mean speedup factor: 8.8).

MacKay, D. J. C. (2003). *Information theory, inference and learning algorithms*. Cambridge University Press.

Müller, M. (2002). Computer Go. *Artificial Intelligence, 134*, 145–179.

Nilsson, N. J. (1971). *Problem solving in artificial intelligence*. McGraw-Hill.

Palay, A. J. (1985). *Searching with probabilities*. Pitman Publishing Ltd.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving*. Assison-Wesley.

Plaat, A., Schaeffer, J., Pijls, W., & de Bruin, A. (1986). Exploiting graph properties of game trees. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (pp. 234–239). Portland, OR.

Russell, S., & Norvig, P. (1995). *Artificial intelligence: A modern approach*. Prentice Hall.

Russell, S., & Wefald, E. (1991). *Do the right thing: Studies in limited rationality*. The MIT Press.

Stern, D., Herbrich, R., & Graepel, T. (2006). Bayesian pattern ranking for move prediction in the game of Go. *ICML '06: Proceedings of the 23rd international conference on Machine learning* (pp. 873–880). New York, NY, USA: ACM Press.