The Mathematical Game

Marc Pierre, Quentin Cohen-Solal, and Tristan Cazenave

LAMSADE, Université Paris Dauphine - PSL, CNRS, Paris, France

Abstract. Monte Carlo Tree Search can be used for automated theorem proving. Holophrasm is a neural theorem prover using MCTS combined with neural networks for the policy and the evaluation. In this paper we propose to improve the performance of the Holophrasm theorem prover using other game tree search algorithms.

1 Introduction

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [2]. It was used to build superhuman game playing programs such as AlphaGo [7], AlphaZero [8] and Katago [10]. It has been recently used to discover new fast matrix multiplication algorithms [3]. It is also used for automated theorem proving such as in the Holophrasm theorem prover [9]. In this paper, we propose to replace the MCTS used by Holophrasm by other game tree search algorithms. We will start by briefly explaining how metamath works, as well as the Holophrasm interface. Then we will move on to an application of existing tree search algorithms modified for this context, such as Minimax, PUCT or Product Propagation. Finally, we propose a new algorithm, which is an association of existing ones, and apply it in the context of Holophrasm.

2 Holophrasm and Metamath

2.1 Metamath

Metamath is a mathematics language [5]. Its main function is based on the principle of logical substitution. For example, let's imagine that we are at a certain step of a theorem's proof, and to move on to the next step we need to apply a proposition. To do this, we need to change the variables of the proposition we wish to apply, so that its hypotheses correspond to the current state of the proof. For more details, one can check the metamath's book ([5]). Holophrasm transforms this structure of theorem's proof into an "AND/OR" tree. OR nodes represents the current state of the proof. They are considered proven if one of their child is proven or if they are one of the initial hypotheses of the theorem. Their children are AND nodes representing a proposition to be applied to prove the OR father node. The children of an AND node are the set of hypotheses to be proven for the proposition to be true. A hypothesis is modeled by an OR node and an AND node is considered proven if all its children are proven. In

Holophrasm, the theorem is proved by working backward. The root being the conclusion of the proof modeled by an OR node, each of its children is an AND node which is a proposition with its substitution of variables. Concerning the theorems on which we will test our algorithms, the benchmarks are made up of a list of theorems and are provided by the Holophrasm interface. However, due to time constraints, we will only test on the first 200 theorems in the list.

2.2 Classical Holophrasm

Now that we have seen the structure of proof trees, we will explain the search used by Holophrasm [9] through this trees.

Algorithm The algorithm visits the root using $VisitNodeOR$ as long as this root has not been proven or the number of his visits has not reached a certain threshold. The value of the root is, as for all OR nodes, a probability calculated by the payout neural network modeling the chance of proving the OR node with the hypotheses it contains. When visiting an OR node, the objective is then to visit an AND node (proposition), which has the highest chance of being provable. Note that for an AND node, its probability is given by the prediction neural network and the substitution is given by the generative network. Furthermore, when an AND node is created, all its children are also created and visited once. Initial values are given by two different networks, depending on the nature of the node. The payout network takes an OR node as an argument and outputs the probability that this node is provable with the assumptions it contains. The prediction network gives a softmax on the set of propositions applicable to an OR node. Another feature we would like to detail is the interface adding an AND node to an OR node. Holophrasm uses a heap in which all potential candidates are stored. A visit to an OR node may not add an AND node, if the candidate is not valid. An OR node will be considered disproven if all its children are disproof, or if it has been visited enough times and the heap is empty. An AND node is disproven if one of its childs is disproven. Disproven AND node are cut from the tree.

Algorithm 1: Research Function of Holophrasm

```
1 Function Holophrasm(node, maxpasses):
2     passes = 0;
3     while (not node.proven) or passes<maxpasses do
4         VisitNodeOR(node);
5         UpdateProven(node, "OR");
6         UpdateValueOR(node);
7     end
```

Algorithm 2: Visit of an "AND" node by Holophrasm

```
Function VisitNodeAND(node):
    if len(node.children) > 0 then
        VisiteNodeOR(node.children[argmin([child.value/child.visit for child in
          node.children])]);
    end
    UpdateProven(node, "AND");
    UpdateValueAND(node);
```

Algorithm 3: Update of an "AND" node by Holophrasm

```
Function UpdateValueAND(node):
    if len(node.children) > 0 then
        badchild = node.children[argmin([child.value/child.visit for child in
          node.children])];
        node.visit = badchild.visit;
        node.value = badchild.value;
    end
return 0;
```

Algorithm 4: Visit of an "OR" node by Holophrasm

```
Fonction VisitNodeOR(node):
    if node.heap is empty then
        Use the Holophrasm Interface to fill node.heap of all compatible
          proposition;
    end
    if node.visit/6 + 0.01 > len(self.children) + node.childlessvisit then
        Try to add an "AND" node from the heap, if it fail
          node.childlessvisit+=1;
        return 0;
    end
    if len(node.children) > 0 then
        value = 0;
        nextchild = None;
        for child in node.children do
            if valuation-function(node.visit, child) > value then
                value = valuation-function(node.visit, child);
                nextchild = child;
            end
            VisiteNodeAND(nextchild);
        end
    end
    UpdateProven(node, "OR");
    UpdateValueOR(node);
```

Algorithm 5: Update of an "OR" node by Holophrasm

```
Function UpdateValueAND(node):
    if len(node.children) > 0 then
        node.visit = sum(child.visit for child in node.children) +
          node.childlessvisit + 1;
        node.value = node.networkpayout + sum(child.value for child in
          node.children);
    end
```

Algorithm 6: Update if a node is proven

```
Function UpdateProven(node, type):
    if type == "OR" then
        if len(node.children) == 0 and node.visit> 1 and node.heap is empty
          then
            node.dead = True ;
        end
        for child in node.children do
            if child.proven then
                node.proven = True;
            end
        end
    end
    if type == "AND" then
        node.proven = True;
        for child in node.children do
            if not child.proven then
                node.proven = False;
                break;
            end
        end
        if any ([child.dead for child in node.children]) then
            CutNodeFromTree(node);
            return 0;
        end
    end
```

Algorithm 7: Evaluation Function for guiding the exploration in Holophrasm

```
Function valuation-function(fathervisit, nodeAND):
    return nodeAND.value/(nodeAND.visit+1) + 0.5 * nodeAND.probabilitynet/(1+nodeAND.visit) + sqrt(log(fathervisit)/(1+nodeAND.visit));
```

Results On the first 200 theorems of the Holophrasm's Test set and with the parameter 10 for the BeamSearch used by the generative network, we obtain Figure 1.

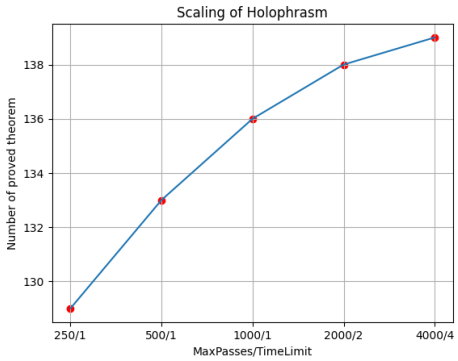

Fig. 1. Results Holophrasm

We are not going to make a detailed analysis of the results, which we will use mainly to compare with the other algorithms we are going to test.

3 Classical Tree Search Algorithm for Metamath Theorem Proving

In this section we will take a look at a number of well-known algorithms applied in this context by changing the Holophrasm search. We will start by analysing Minimax and its results, then we will study more selective algorithms such as PUCT.

3.1 Minimax

Now that we have tested the search algorithm provided with the Holophrasm interface, we are going to test a more traditional search: Minimax. The aim is to compare a more conventional search algorithm with the Holophrasm's results, and highlight the importance of progressive widening (the expansion of OR node's child).

Algorithm We launch a Minimax search from the root by setting the depth. However, the problem is that an OR node can have an infinite number of children. To overcome this problem, we limit the number of possible children to a fixed breadth. The children of the OR node are chosen according to their probability

given by the prediction neural network. The depth is reduced by 1 each time we go from an AND node to an OR node, and the algorithm ends on OR nodes. With regard to node initialization, when an AND node is created, all its OR child nodes are created and checked if they are an initial hypothesis of the problem. The point of testing several breadths is to evaluate the efficiency of the network in finding the next proposition to apply to an OR node.

Results Our test set consists of the first 200 Holophrasm theorems. The test conditions are 1 pass in the maximum tree and a parameter of 10 for the BeamSearch used in the Holophrasm interface.

	depth = 2	depth = 3
breadth = 2	78/200	79/200
breadth = 3	94/200	95/200
breadth = 4	97/200	98/200
breadth = 5	103/200	108/200
breadth = 7	112/200	113/200
breadth = 9	117/200	.../200

From the results, we can see that the networks are often wrong, and that we need to go to wider breadths to get better results. This underlines the importance of the progressive widening used in Holophrasm.

3.2 PUCT

We now test other conventional algorithms such as PUCT [7] , Product Propagation [6] and Proof Number Search [1]. These tree search methods are more recent than Minimax and they allows one to find a solution without exploring the whole tree. Some of these approaches seams to not use networks, but in fact they are used implicitly by the interface when it attributes an And node to an Or node.

Algorithm The idea behind PUCT is to supervise the search when choosing the next AND node to visit. This algorithm is inspired by the Bandit literature. To implement PUCT from Holophrasm's research, we will just change the Holophrasm bandit by changing the valuation-function.

Algorithm 8: Evaluation Function for PUCT

Result: PUCT

Function `valuation-function(`*fathervisit, nodeAND*`):`

Return $\frac{nodeAND.value}{nodeAND.visit} + C * \frac{nodeAND.prediction\_net}{1+nodeAND.visit} * \sqrt{fathervisit}$;

Results First, we will determine the best constant to use in PUCT. With the same BeamSearch setting as before and on the parameters (1000 passes, 1 min), we obtain:

C	0.1	0.2	0.3	0.4	0.5
Results	136/200	136/200	136/200	136/200	135/200

We will therefore test the Scaling of PUCT with $C = 0.2$, the results are described in Figure 2.

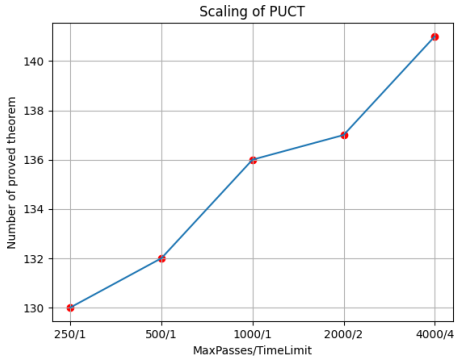

Fig. 2. Results PUCT for C=0.2

In comparison with Holophrasm, the results are quite similar except for the parameters (4000,4). So, with less time constraint, PUCT does better than Holophrasm. It is difficult to explain this behavior further. We will see below that there is a kind of ceiling around 140, whatever the algorithm used.

3.3 Product Propagation

Since PUCT is an algorithm that affects exploration, we will now look at Product Propagation (PP) [6], which mainly changes the value and update of nodes.

Algorithm The idea is to change the visit and the values attributed to the nodes during the search. The value is seen as a probability and the children are assumed to be independent. Thus the value of an AND node is the product of the values of its children and the value of an OR node is calculated using the same principle but with the additional probability. In an OR node we explore the best valued child, and in an AND node the worst. The value of Leaf can be initialized with 1 or by the payout network given by Holophrasm, but in our case we used the payout network for better results.

Results With the the same settings as before we obtain Figure 3. The results show, in this context, that PP is more efficient than PUCT on parameters where the number of passes is limiting. However, for the parameter (4000,4), we achieve

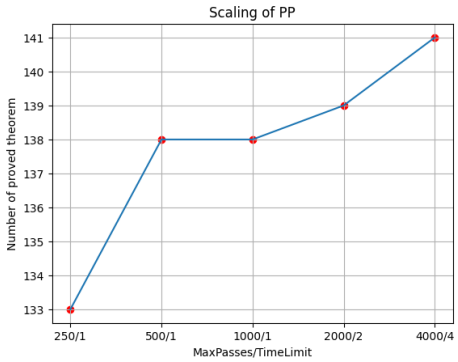

Fig. 3. Results Product Propagation

the same result as PUCT. PP seems to be more efficient, proving 138 propositions in just 1 minute and 500 passages maximum.

3.4 Proof Number Search

Similar to PP, Proof Number Search (PNS) [1] is based on node values and updates. However, this time there are two values per node to take into account.

Algorithm The idea is to count the number of leaves left to explore either to prove the node or to disprove it. During the node initialization, a non proven node is initialized with a Proof Number and a Disproof Number. In the original algorithm, for a non-proven or proven node, the PN and DPN are initialized to 1. For a proven node, the PN is set to 0 and the DPN to infinity, and vice versa for a proven node. Then during the visit, in an OR node we choose the AND node with the lowest DPN and in an AND node we choose the OR node with the lowest PN. One might think that neural networks are not used in this algorithm, but they are used indirectly in AND node assignment.

Results With the same settings as before, we obtain Figure 4. The results are weaker than Product Propagation and all its variants (which we will see later), but in this test we are not using the value given by the "payout" neural network. We tried to improve PNS, but we could not find any approach that improve the results presented above, either by initializing PN and DPN with the networks, or by using a PUCT in the search, so we haven't included these approaches in this article.

3.5 HyperTree Proof Search

We are now going to test an algorithm which is different in his approach from the previous ones. This algorithm try to obtain a broader perspective by selecting wider sub-tree instead of doing descent.

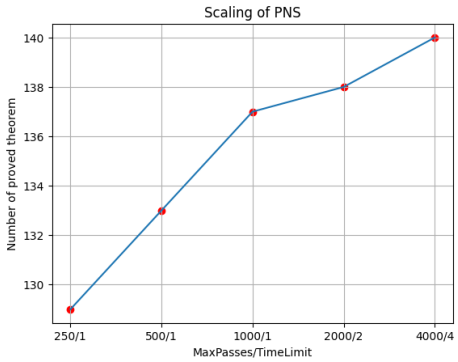

Fig. 4. Results PNS

Algorithm The goal is to draw inspiration from the search algorithm presented in [4], in order to compare the results of different approaches in this context. However, an adaptation is necessary because the interfaces used are different. The idea is to select a sub-tree of the proof tree, expand the leaves, then update only the values of the AND nodes of this sub-tree using Product Propagation. Transposed to our interface, this is like using PUCT to select AND nodes, but once in an AND node, expand all its OR children.

Results With 10 for the interface BeamSearch, we obtain the results described in Figure 5.

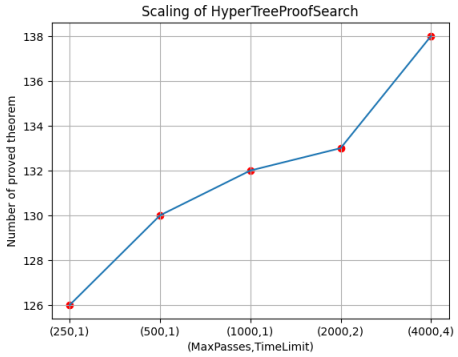

Fig. 5. Results HyperTree Proof Search

The results, in this context, are inferior to those of the previous algorithm. Extending all OR nodes that are children of an AND node is time-consuming. This explains the jump in performance from 2 to 4 minutes. This may be the

algorithm with the worst results, but when you look at the evolution of the curve, it is the one that improves the most as a function of time.

4 New Tree Search Algorithm for Metamath Theorem Proving

Now that we have seen the more traditional approaches, we are going to use them as an inspiration to create new ones.

4.1 Production Propagation Combined with PUCT

Algorithm Given the performance of Product Propagation and PUCT, the idea is to combine the two approaches. In fact, these two algorithms should work well together, since Product Propagation does not rely on a bandit's part. We'll present the two combinations we've tested. For the first algorithm, the Product Propagation value is used, along with the PUCT visit and bandit. In the case of the second approach, we change the PUCT bandit (see algorithm 9). The aim was to keep the first Holophrasm bandit approach, but combine it with PUCT. This is useful especially when the policy given by the payout network is greatly underestimating a son. In this case the part coming from UCB compensates.

Algorithm 9: Modified Evaluation function for PUCT

Function `modified-valuation-function(`*fathervisit, nodeAND*`):`
  Return $nodeAND.value + a * node.probabilitynet * \frac{\sqrt{fathervisit}}{nodeAND.visit} + b * \sqrt{\frac{\log(fathervisit)}{nodeAND.visit}}$;

Results With 10 for the interface BeamSearch and the modified validation function we obtain :

	a = 0.1	a = 0.4	a = 0.6	a = 0.8	a = 1.0	a = 1.1
b = 0.1	...	134/200	...	...	...	...
b = 0.4	133/200	135/200	136/200		137/200	...
b = 0.5	...	...	...	138/200	137/200	136/200
b = 0.6	...	130/200	136/200		136/200	...
b = 1.0	...	...	134/200		135/200	...

In the case of the PUCT bandit (Figure 6), we can see that combining the two approaches results in a slight increase in performance. However, we are still unable to break the 141-theorem barrier.

There are two interesting points to note about the modified PUCT bandit (Figure 7). Firstly, the algorithm performs particularly well with parameters (4000,4), where it proves 144 theorems (and thus breaks the 141-theorem barrier). It is therefore the best on these criteria. Note that, this bandit exploration term has been used with other search algorithms, including Holophrasm, but its

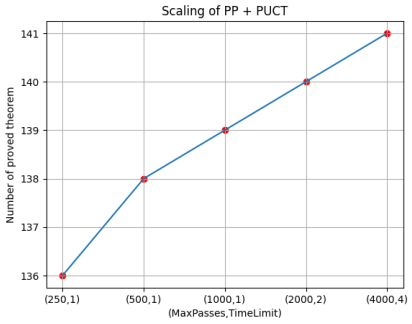

Fig. 6. Results PP + PUCT c=0.2

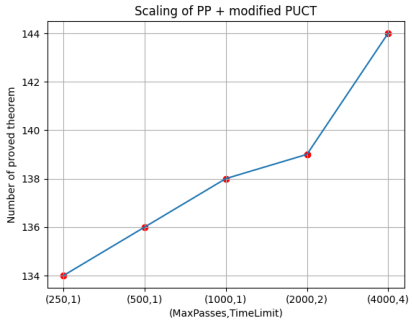

Fig. 7. Results PP + PUCT with modified validation function a=0.8 b=0.5

results were not convincing. The final point is that if we remove the principle of node unprovability (see the UpdateProven 24 code line 3-4 and 20-22), this algorithm is the one with the best results.

5 Conclusion

We studied different algorithms applied to theorem proving in Metamath. Minimax algorithm highlighted the importance of progressive widening. Algorithms such as PUCT, PNS or PP obtained better results than the search used by Holophrasm. Finally, we propose a new algorithm by combining the idea of PUCT and PP and obtain better results.

Our initial goal was to improve the search used by the Holophrasm interface. The different algorithms we proposed, as well as the different solutions we provided, enabled this improvement. Although the improvement is slight (see Figure 8), our different approaches seem more promising in terms of parameter-dependent changes in performance. In future work we plan to test this hypothesis by applying the algorithms on the whole dataset.

References

1. Allis, L.V., van der Meulen, M., van den Henrik, H.: Proof-number search **1** (2003)
2. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4**(1), 1–43 (Mar 2012). https://doi.org/10.1109/TCIAIG.2012.2186810
3. Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F.J., Schrittwieser, J., Swirszcz, G., et al.: Discovering faster matrix multiplication algorithms with reinforcement learning. Nature **610**(7930), 47–53 (2022)
4. Lample, G., Lachaux, M.A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., Rodriguez, A., Lacroix, T.: Hypertree proof search for neural theorem proving. Neural Information Processing Systems (NeurIPS) (2022)

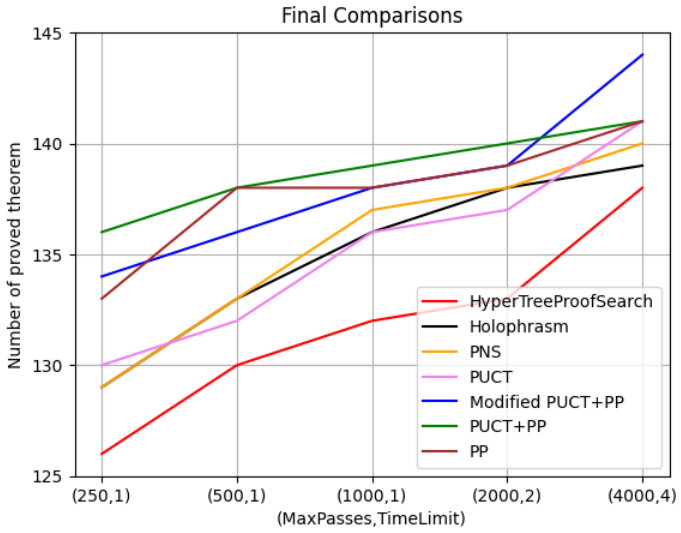

Fig. 8. Final Comparisons

5. Megill, N.D.: Metamath: A Computer Language for Mathematical Proofs. Lulu Press, Morrisville, North Carolina (2019)
6. Saffidine, A., Cazenave, T.: Developments on product propagation. In: CG 2013. pp. 100–109. Springer (2014)
7. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (2016)
8. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
9. Whalen, D.: Holophrasm: a neural automated theorem prover for higher-order logic. arXiv preprint arXiv:1608.02644 (2016)
10. Wu, D.J.: Accelerating self-play learning in go. arXiv preprint arXiv:1902.10565 (2019)