Introduction

This is a short review of "Game Tree Searching by Min/Max approximation" by Ron Rivest

Selecting the paper to review

Being an avid chess player for 30+ years and having witness the evolution of computer chess since the early 80's till now, I rejected the Deep Blue paper, since what it brought to the table for computer chess was just publicity. There are many more interesting chess engines that really played a role in chess programming history, among them **Fruit** chess engine by **Fabien Letouzey** or **Crafty** by **Robert Hyatt**. On the other hand , the selected paper, addresses a problem that is really interesting and is certainly not closed.

Key question

The main question the paper resolves around is:

```
    Given a partial game tree E, how do you select which node you should
"extend"
```

In the classical alpha beta pruning search with iterative deepening we do follow an "expand all" strategy from depth d to depth d+1. The question is can we do better and selectively expand node and if yes, is it efficient?

This is the main subject of the paper. There is also one more issue that arises in the discussion of the paper that will be addressed at the end.

Formulation

A big part of the paper is focused in the precise formulation of the tree, the partial tree, the definition of the evaluation function etc... The key parts of the paper are

- Every leaf node has an evaluation, from a static evaluation function
- Every parent has an evaluation that is either a min or a max of its children
- the evaluation function is strictly positive. For that we may need to add a constant to the initial evaluation function, if it gives positive values for player 1 and negative for player 2
- for every edge, connecting the parent and the child we attach a weight.
- for every parent , we attach as information the index of the best child and call that an arrow
- since we are dealing with partial game trees (full game trees will have all the information provided) the paper calls the **terminal nodes** of the partial tree **tip nodes**
- we are interested on which tip node to expand next
- for every node we define A(c) as the set of all ancestors of c
- We are discussing acyclic game trees (no cycles - no transpositions)
- We introduce the notion of a penalty to every node π(c)
- The penalty of the tip nodes is zero
- the penalty of every parent node is the penalty of its best child + the weight of the edge that connect it
- we can also attach another "penalty" p(c) to every tip node, that is the sum of weights w of all of its

ancestors
- we select to expand the tip node with the least penalty

Selecting the weighting function

- We want to expand that tip node that the root is most "sensitive" to its change (this is formulated mathematically in the paper)
- in the context of min and max functions we are using, its obvious that we would always select the tip node that is the end of the **principal variation** , that is we follow from the root the best move, and we will always expand the best move.
- to avoid that, we use a generalized notion of the min and max functions, borrowed from L^p spaces
- we define a generalized mean for every p as follows :

```
![generalized mean](https://latex.codecogs.com/png.latex?
\bg_white&space;M^p_a&space;=&space;\bigg(\frac{1}
{n}&space;*&space;\sum&space;a_i^p\bigg)^{1/p})
```

- if we let p go to +inf then what we get is the maximum function
- if we let p go to -inf then what we get is the minimum function
- the benefit of this formulation is that we can now have continuous approximations of the min an max functions and this calculate derivatives
- the partial derivate of the new function M on one of its components is:

$$\frac{\partial M_a^p}{\partial a_i} = \frac{1}{n}\left(\frac{a_i}{M_a^p}\right)^{p-1}$$

- if we take the derivative of the max function computed on a 'father' node (denoted as f(c)) against one of its children we have

$$\frac{\partial M^p(f(c))}{\partial c} = \frac{1}{n}\left(\frac{eval(c)}{M^p(f(c))}\right)^{p-1} = \frac{1}{n}\left(\frac{eval(c)}{eval(d)}\right)^{p-1}$$

  where d is the sibling of c with the best evaluation

- if we take logs we have
  - $-[log(1/n) + (p-1) * log(eval(c)) - log(eval(d))]$
  - this last expression quantizes how much a change in node c will affect the parent f(c)
  - this the **negative** of this exact expression we set at the weight w(c) and attach it to the edge from f(c) to c
  - n in this equation is the number of siblings of c

- using the chain rule we can propagate the influence of every tip node to the root (does it remind you of back propagation....??)
- because we took the negative of the derivative and we want the biggest influence , we are searching for the tip node with the least penalty (defined above as the sum of weight of the path leading to c)
- choosing a large value of p, thus approximating the max function will result in selecting the "best" moves to expand , small values of p, will result in a wider tree and more nodes expanding.

Remarks

- The above approach is good not only if we want to find the best move, but if we are interested in actually evaluate the position, which in our case means create a graph that evaluates the most important positions.
- It is a very good approach for creating an "opening book"

Actually this approach has been implemented in a chess program called Aquarium, in a internal function called IDeA (Interactive Deep Analysis), which I had a pleasure of being an alpha tester and contributing to ideas and code.

The basic idea of the program is to create a persistent chess graph, which is expanded automatically (and interactively by user assistance). The program , as in the paper

- uses an static evaluator for every node (an external chess engine that returns an evaluation for the position)
- uses evaluation >0 (internally the evaluations are shifted so the evaluations are always > 0)
- the evaluations are minimaxed to the root(s) of the graph
- the graph, contrary to the paper contains cycles (transpositions), which makes bottom up minimaxing challenging (I am actually very proud of solving that problem)
- in the program , contrary to the paper, not **every** move to a position expanded is added to the tree (this is actually a highly debatable decision)
- it also uses a parameter , like p in the program, to decide on which nodes to expand or not.

In my experience as a chess player at GM level at correspondence chess , this selection algorithm is very helpful when exploring the truth of a position in a complex game without **time** limitations, but is also as good as its static evaluator. Especially positions with sacrifices, where an early evaluation is bad (fail low) will get low priority to expand, so it might be useful to add more intelligent weight functions, depending on the game (for example in chess when there is a king attack, a pawn march, multiple exchanges).

Maybe adding weights based on patterns analyzed by Neural Networks would also help