Expected Work Search: Combining Win Rate and Proof Size Estimation

Owen Randall, **Martin Müller**, **Ting-Han Wei** and **Ryan Hayward**
University of Alberta
{davidowe, mmueller, tinghan, hayward}@ualberta.ca

Abstract

We propose Expected Work Search (EWS), a new game solving algorithm. EWS combines win rate estimation, as used in Monte Carlo Tree Search, with proof size estimation, as used in Proof Number Search. The search efficiency of EWS stems from minimizing a novel notion of Expected Work, which predicts the expected computation required to solve a position. EWS outperforms traditional solving algorithms on the games of Go and Hex. For Go, we present the first solution to the empty 5×5 board with the commonly used positional superko ruleset. For Hex, our algorithm solves the empty 8×8 board in under 4 minutes. Experiments show that EWS succeeds both with and without extensive domain-specific knowledge.

1 Introduction

Efficient search algorithms are critical for finding solutions in large problem spaces. For example, Monte Carlo Tree Search was a key component in finding faster matrix multiplication algorithms [Fawzi *et al.*, 2022]. Search has also been shown to improve reasoning and problem solving capabilities of large language models [Yao *et al.*, 2023].

Expected Work Search (EWS) is a novel search algorithm for two player zero sum games with perfect information. EWS utilizes estimates of both win rate and proof size in order to predict the expected amount of work that it will take to solve a given position[1]. The Expected Work (EW) heuristic is used to direct the search towards smaller, less expensive solutions, and minimize the time wasted searching more complex wins, or losing variations. We evaluate EWS by solving small-board positions from the two player perfect information games of Go and Hex. This involves proving which player wins with optimal play, and creates a complete winning strategy which has a response for every possible opponent move.

EWS combines and extends elements of two popular algorithms, Proof Number Search (PNS) [Allis *et al.*, 1994] and Monte Carlo Tree Search (MCTS) [Browne *et al.*, 2012]. These algorithms use dynamically updated heuristics to efficiently find solutions, which can give them an edge over more traditional minimax-based programs such as $\alpha\beta$ solvers [Knuth and Moore, 1975]. MCTS uses its win rate estimation to deeply search lines of play following the best moves found so far. PNS estimates proof sizes, and seeks to minimizes proof or disproof numbers in order to quickly find a solution. Both algorithms have strengths and weaknesses. MCTS makes no effort to find an easy solution in a short amount of time, while PNS can spend much time on searching deep forcing lines that do not work.

Our results show that combining win rate and proof size estimation into a single Expected Work heuristic allows EWS to outperforms existing search algorithms in both Go and Hex. EWS is able to solve the empty 8×8 Hex board in under 4 minutes on modest hardware (a first player win), and is the first program to solve the empty 5×5 Go board using positional superko rules (a first player full-board win).

[1]We call the complete game state a *position*, including information such as the board configuration of a Go game and the move history where needed. Furthermore, we identify nodes in the search tree with the positions they represent.

2 Background

$\alpha\beta$ pruning [Knuth and Moore, 1975] enhances the basic minimax algorithm by using upper and lower score bounds α and β to prune irrelevant lines of play. Iterative deepening $\alpha\beta$ is an enhancement to the algorithm which allows it to find wins in the minimum number of moves [Slate and Atkin, 1977]. Each iteration executes a depth-limited $\alpha\beta$ search, and non-terminal positions at the depth limit are evaluated by a heuristic. The best moves found in the previous iteration are prioritized so that strong lines of play are searched first. With an accurate heuristic, this results in more $\alpha\beta$ pruning, saving computation.

Proof Number Search [Allis *et al.*, 1994] uses proof and disproof numbers as optimistic heuristic estimates of the remaining cost to complete a proof or disproof. The algorithm constructs an in-memory proof/disproof tree by iteratively expanding most promising nodes, and updating proof/disproof numbers in the tree. PNS searches for solutions in a best-first manner, in contrast with iterative deepening $\alpha\beta$. PNS can quickly find narrow but deep solutions containing long forced lines of play, with few branches [Kishimoto *et al.*, 2012]. However, this depth-seeking behavior of PNS can also cause the algorithm to waste much computation on deep lines of play that do not contribute to the final proof.

Monte Carlo Tree Search [Browne *et al.*, 2012] is a popular search framework for heuristic game play and single agent search for optimization. The MCTS Solver extension [Winands *et al.*, 2008] adds minimax backpropagation of solved positions in order to solve games. Like PNS, MCTS incrementally builds a tree of explored nodes in memory. However MCTS uses an empirical win rate instead of proof numbers, to guide its move selection according to a formula such as UCT [Kocsis and Szepesvári, 2006] or PUCT [Rosin, 2010]. MCTS balances exploitation of moves with the highest win rates against exploration of less visited ones. In each iteration of MCTS, the search tree is expanded with a new leaf node, which is evaluated using either a simulation or a neural network evaluation. Win/loss statistics are propagated along the path back to the root.

$\alpha\beta$ and MCTS can be adapted to find scalar-valued outcomes, while PNS is specialized for binary (win/loss) results. There are techniques to efficiently specialize $\alpha\beta$ for binary results, and PNS variants for solving scalar-valued games [Saffidine and Cazenave, 2012]. For simplicity, our implementation of EWS solves binary outcome problems. Multi-outcome problems could be solved using multiple searches or extended bookkeeping.

The game of Go is a classic test bed for search algorithm research as it is a popular game with simple rules, yet good play requires complex strategies. In Go, the two players Black and White take turns placing a stone of their color onto the game board, until both players pass and the game ends. Adjacent stones of the same color are part of the same *block*. A block losing its last adjacent empty point is captured and its stones are removed. Static safety is the process of recognizing without search when points on the board can be guaranteed to count towards one player's score. Such domain-specific knowledge can greatly speed up the search [Randall *et al.*, 2023].

Superko rules in Go prevent infinite loops during games. Positional superko forbids repeating any previously played board position, and is a commonly used ruleset by Go players. In situational superko, the player to move also has to match in order to repeat a position. Japanese style Go rules address only restricted types of ko, and declare the game as no-result in more complex cases of repetition. Positional or situational superko rules make solving Go more challenging due to the Graph History Interaction (GHI) problem [Campbell, 1985]. The game history must be considered, which makes it more difficult to reuse previously solved positions. Previous work in solving Go by van der Werf et al. [van der Werf *et al.*, 2003] [van der Werf and Winands, 2009] used a version of Japanese rules. In their MIGOS program, using situational superko instead of Japanese style rules increased the time for solving 4×4 Go from 14.8 seconds to 1.1 hours, more than two orders of magnitude.

The game of Hex is played on a rhombus board with hexagonal cells. As in Go, two players Black and White take turns placing their stones. The goal of the first player (Black) is to connect the top of the board to the bottom using their stones, and the second player (White) tries to connect the left edge to the right edge. Hex has been solved up to the 10×10 board using a Depth-First PNS algorithm (DFPN) [Nagai, 2002], with many Hex specific improvements, in 63 days of parallel computation [Pawlewicz and Hayward, 2013].

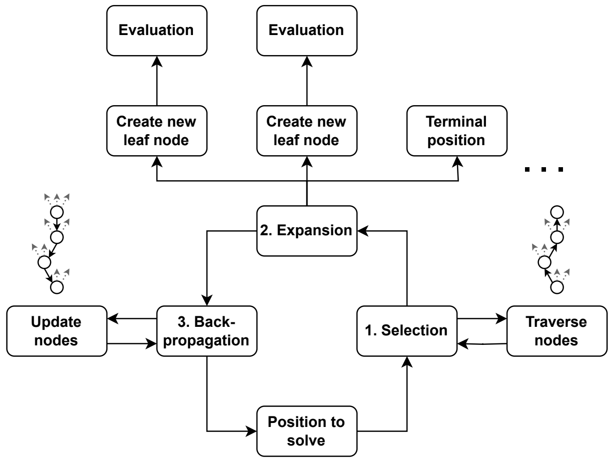


Figure 1: Basic diagram of the structure of Expected Work Search.

3 Expected Work Search

EWS uses a MCTS-like framework for search. It is designed to minimize a new notion of *Expected Work* (EW), in contrast to algorithms with move selection formulas such as UCT. Figure 1 shows the stages of an iteration of EWS: Selection, Expansion, and Backpropagation. EWS uses a negamax formulation with wins and losses defined from the point of view of the player to move.

As in MCTS Solver, repeating these stages grows a search tree of nodes which represent intermediate positions until a complete winning strategy is found for the initial position. The selection stage traverses the search tree to choose a leaf node X for expansion. Expansion adds *all* non-terminal children of X to the search tree and evaluates them (for example by simulation) to initialize their win rate and Expected Work estimates. Backpropagation updates the proofs, child ordering, win rates, and Expected Work of all nodes on the path back to the root.

EWS requires the following domain-specific functions: getting all legal moves, making and undoing moves, and recognizing terminal positions and their win/loss outcomes. Everything else in the EWS algorithm was designed to be domain agnostic so that it can be easily applied to many different problems.

3.1 Expected Work

EWS computes the Expected Work of positions to estimate how much computation they will take to solve. The Expected Work of a position depends on whether it will be proven to be winning or losing. For a winning position, only a single winning move needs to be solved, while in a losing position all moves must be solved. Therefore, EWS computes separate Expected Work estimates for both cases for each position. Let X be a position with n children $C_1, ..., C_n$, arranged in the order in which they will be searched (C_1 first, then C_2 etc.). Let $WR(Y)$ be the estimated win rate of Y. Then the Expected Work for losing and winning of X is defined recursively as:

$$EW_{loss}(X) := \sum_{i=1}^{n} EW_{win}(C_i) \tag{1}$$

$$EW_{win}(X) := \sum_{i=1}^{n} (EW_{loss}(C_i) \cdot \prod_{j=1}^{i-1} WR(C_j)) \tag{2}$$

The estimated win rate of X is interpreted as the probability that X is a win. As a base case, a leaf node in the current tree is initialized with a EW heuristic. initialisation options are discussed in Section 4.2.

If X is losing, then all its children must be searched and proven to be wins for the opponent. Equation (1) defines $EW_{loss}(X)$ as the sum of EW_{win} of its children. This corresponds to proof numbers in PNS AND nodes, which are defined as the sum of child proof numbers.

If X is winning, then EWS searches its children **in order** (first C_1, then C_2, etc.) until the first opponent loss C_i is found, which proves a win for X. Equation (2) defines $EW_{win}(X)$ as the sum of all children's EW_{loss}, each weighted by the estimated probability that the search reaches this child while solving X. For example, suppose C_1 has a 80% win rate and C_2 has a 75% win rate. Then $0.8 \cdot 0.75 = 0.6$ is the estimated joint probability that neither C_1 nor C_2 were solved as losses for the opponent. This is the case when C_3 must still be searched, so the EW of C_3 is weighted by this probability. PNS taking the minimum proof number in OR nodes would correspond to the special case of $WR(C_1) = 0$, which would make Equation 2 equal $EW_{loss}(C_1)$, which is a minimal child EW.

The EW computation is based on three simplifying assumptions. First, that win rates correlate with the position actually being winning or losing. EWS does not prescribe how to estimate win rates. A default option using domain-agnostic random simulations is described in Section 4.1. Second, the definition of Expected Work assumes that the current move ordering among children will remain unchanged. However, in practice EWS reorders children continually as the search updates EW. The third assumption is that the win rates and Expected Work of siblings are independent. This is a strong assumption, and violated for example in a DAG. However, our empirical tests in Section 5 demonstrate that the above EW definition leads to an efficient search algorithm, validating the usefulness of this framework.

3.2 Selection

The pseudo code for selection can be seen in Algorithm 1, lines 1-6. As in MCTS, the selection stage proceeds from the root of the search tree down a path to a leaf. Unlike MCTS, this path simply follows the first-ordered child in the tree according to the current ordering until a leaf node is reached. The child ordering is determined during backpropagation, as described in Section 3.4. Once a leaf node is found, it is expanded as described in Section 3.3.

As the in-memory tree is traversed, moves are made accordingly to update the current position. Each node stores a win rate, both Expected Work values, whether it has been expanded yet, the last move which lead to its position, and pointers to its unsolved children.

Algorithm 1 SelectBackpropagate X: returns whether X is solved and if so whether X is winning

```
 1: C := X.children[0]                                  ▷ Selection
 2: Make move C.move
 3: if C.expanded then
 4:     isSolved, isWinning = SelectBackpropagate(C)
 5: else
 6:     isSolved, isWinning = Expand(C)
 7: Undo move C.move                                    ▷ Backpropagation
 8: if isSolved and isWinning then
 9:     Remove C from X.children
10:     if X.children is empty then
11:         return true, false                          ▷ Solved loss
12: else if isSolved and not isWinning then
13:     return true, true                               ▷ Solved win
14: Sort X.children with (3)
15: Update X.winRate
16: Update X.expectedWorkLoss with (1)
17: Update X.expectedWorkWin with (2)
18: return false, false                                 ▷ Unsolved
```

3.3 Expansion

Algorithm 2 Expand X: returns whether X is solved and if so whether X is winning

```
 1: X.expanded := true
 2: for all legal moves m do
 3:     if m is a terminal winning move for X then
 4:         return true, true                           ▷ Solved win
 5:     else if m is not a terminal losing move for X then
 6:         Create node C
 7:         C.expanded = false
 8:         C.move := m
 9:         Evaluate C.winRate
10:         Evaluate C.expectedWorkLoss
11:         Evaluate C.expectedWorkWin
12:         Add C to X.children
13: if X.children is empty then
14:     return true, false                              ▷ Solved loss
15: else
16:     return false, false                             ▷ Unsolved
```

When the selection stage reaches a leaf node for position p, it is expanded according to Algorithm 2. Expand returns two booleans: whether the node was solved, and if so whether p was winning. All legal moves of the position being expanded are checked to see if they lead to a terminal position. If a move leads to a terminal position which is winning for the player to move in p, then the node is solved as a win and the function returns. If a move leads to a losing terminal position, no new node is added. If p has no unsolved children left, it is solved as a loss, otherwise p remains unsolved.

For a move which leads to a non-terminal position, a new leaf node is added as a child of p's node, and its win rate and Expected Work values are initialized by heuristics. This is further described in Sections 4.1 and 4.2.

3.4 Backpropagation

Lines 7-18 in Algorithm 1 show the pseudo code for EWS backprogagation. After the selected child C of node X has finished expanding or backpropagating (lines 4, 6), backpropagation for X proceeds as follows: If C was solved as a win, it is losing for X and is removed from the (unsolved) children. If X has no more unsolved children, its result is a solved losing position. If C was solved as a loss, through backpropagation X becomes a solved win.

For X that remain unsolved, its unsolved children are re-sorted in ascending order according to the ordering:

$$\begin{gathered} A < B \\ \iff \\ EW_{loss}(A)/(1-WR(A)) < EW_{loss}(B)/(1-WR(B)) \end{gathered} \tag{3}$$

Here, A and B are any two children of X. This child ordering minimizes the EW of nodes, a formal proof of this is provided in the appendix of our arxiv submission [Randall *et al.*, 2024]. EWS achieves exploration by optimistic initialisation of EW, as described in Section 4.2. Move ordering assumes that the node is winning, as move ordering is irrelevant for losing nodes since every child needs to be solved. Therefore EW_{loss} is used for the ordering formula.

The win rate of X is updated after its children are reordered. Details are given in Section 4.1. Expected Work values are updated according to Equations (1) and (2). Finally, the function return indicates that X is still unsolved.

4 Implementation

This section describes choices made for our first implementation of EWS within the algorithm framework of Section 3.

4.1 Win Rate Estimation

Our implementation of EWS performs win rate estimation with random simulations (a.k.a. rollouts). A position is simulated by playing random moves until a terminal position is reached, at which point the winner of that simulation is known. Each node has win and visit counters which are updated during expansion and backpropagation. Every simulation increments the visit counter of all nodes along the in-tree path chosen in the selection stage, and the win counter for nodes of the simulation winner. The win rate of a node is simply wins divided by visits. Wins are initialized to 1, and visits to 2, in order to avoid win rates of exactly 0 or 1. This is required for computing Expected Work (2) and the child ordering relation (3).

It is possible to use other methods of win rate estimation with EWS such as a heuristic policy to guide simulations [Silver *et al.*, 2016], or a machine learned evaluation function [Silver *et al.*, 2018] [Wu *et al.*, 2022].

4.2 Expected Work initialisation

EWS does not use an explicit exploration term as in MCTS. It uses optimistic initialisation [Sutton and Barto,] of EW to achieve exploration. This corresponds to the optimistic initialisation of proof and disproof numbers in PNS [Allis *et al.*, 1994]. In optimistic initialisation, underestimating the EW of new nodes leads to it growing during the initial visits. In two otherwise similar nodes, the one with fewer visits tends to have a smaller EW estimate and is therefore prioritized in the search. Typically, a position's optimistic EW initialisation is smaller than the sum of its children's EW.

Our implementation of EWS takes advantage of "free" information from the simulations used for win rate estimation to also initialize the Expected Work of new nodes. EW is initialized as follows:

$$EW(X) := \sum_{i=1}^{m-1} b(P_i) \tag{4}$$

where X is the new node being initialized, $P_1, ..., P_m$ are the m positions visited during the random simulation, and $b(P_i)$ is the number of legal moves (the branching factor) of P_i. The final mth position is terminal, and therefore is not included in the sum. This initialisation method is domain agnostic, gives information about the size of the proof tree required to solve the node, and it reuses simulations used for win rate estimation. Other plausible initialisation methods for EW include simply initializing EW as 1, and using a learned model to predict proof size [Wu *et al.*, 2022].

4.3 Generic Refinements

We improve the performance of our EWS implementation with the following three game-independent methods:

1. Transposition Reduction
2. Symmetry Reduction
3. Relevancy Zones

A transposition is an equivalent position that has already been encountered during the search. We use a transposition table to store the outcomes of solved positions to avoid re-computing such positions. The transposition table is indexed using hashes that compactly represent positions modulo the table's size. We use Zobrist hashing [Zobrist, 1990] to compute hashes efficiently. We increase the transposition table hit rate using enhanced transposition cutoffs [Schaeffer and Plaat, 1996], which involves performing a 1-ply search for existing transposition entries upon the creation of a new node. The complete strategy for solutions found in EWS is stored in the transposition table.

The Graph History Interaction (GHI) problem arises in games such as Go where a position's outcome depends on the game history. In such domains, this problem occurs when positions that share the same board configuration lead to different game outcomes. If positions with different histories are treated as identical via a transposition, the search result might become incorrect. To remedy this issue, we implement the GHI solution proposed by Kishimoto et al. [Kishimoto and Müller, 2004]. Transposition entries are simulated to ensure that the stored outcomes can be reached legally within the rules of the game before they are used to replace search. These simulations incur a memory and computational overhead, but the savings from being able to use a transposition

Program	Av. solve time (s)	# Faster than EWS	# Timeouts
EWS	2.32	-	-
Go-Solver	22.63	31 (5.2%)	11 (1.8%)
EWS-WR	19.70	96 (16.0%)	11 (1.8%)
EWS-PS	19.58	171 (28.5%)	14 (2.3%)

Table 1: Summary statistics of EWS vs. comparison algorithms on 600 tested 6×6 Go positions.

table and re-using a proof rather than needing to rediscover it by search makes up for the overhead.

In many domains, symmetries of positions can be used to improve search efficiency. If a position can be mirrored or rotated without affecting the outcome, then solving a position means that all of its symmetrically equivalent positions are also solved. This allows more positions to be treated as transpositions, increasing the transposition table hit rate and reducing the number of nodes that must be searched. We also consider symmetries upon creation of new nodes, by omitting any symmetrically equivalent sibling.

EWS uses relevancy zones [Shih *et al.*, 2021] to further reduce the number of nodes required to find proofs. Relevancy zones limit the number of moves that must be considered by discarding any moves which cannot be a refutation to a known winning strategy for the opponent. Relevancy zones are defined for terminal positions in a domain-specific way. The propagation to earlier positions is domain-agnostic and is handled by EWS in the backpropagation stage. See [Shih *et al.*, 2021] for a detailed explanation with concrete examples of relevancy zones, and a proof of correctness.

Two other improvements are used in our implementation: First, if the winner of a starting position is conjectured a priori, then each node's expected win/loss outcome can be set accordingly. This allows the search to only compute one EW, by alternating Equations (1) and (2). This leads to an efficiency gain. Second, if the search is observed to oscillate rapidly among a small number of children for the same position, its focus can be improved using the $1 + \epsilon$ trick [Pawlewicz and Lew, 2006].

5 Experiments

5.1 The 6×6Go Dataset

To broadly evaluate EWS, we created dataset *6×6Go* by selecting 600 positions - one from each of 600 6×6 Go games - that are likely to be encountered in a 6×6 Go proof tree. One player is a strong heuristics-guided agent, representing the good moves made by the winning side. The opponent plays randomly, representing a fair sample of all legal moves which must be analyzed for the losing side. The heuristic player uses six Proof Cost Networks (PCN) [Wu *et al.*, 2022] with different training parameters. PCN are trained to predict minimal proof size, as opposed to the win rate used in AlphaZero [Silver *et al.*, 2018]. We generate 100 games with each PCN to create a dataset with diverse lines of play. The games in the test set last between 29 and 124 moves, with a median of 40, for a total of 26,885 positions. In each game, the earliest position EWS was able to solve in at most 10 seconds was used as a data point in 6×6Go. All algorithms are tested with a 5 minute limit per position. Experiments were run on an Intel i7-10510U CPU with 16 GB of RAM.

5.2 Go Results

To evaluate EWS, we compare it against three baselines: Go-Solver, EWS-WR (EWS without win rate estimation), and EWS-PS (EWS without proof size estimation). Go-Solver is an iterative deepening $\alpha\beta$ algorithm built on the open source Fuego framework [Enzenberger *et al.*, 2011]. Go-Solver is the strongest existing Go solving program using positional superko rules we are aware of, apart from EWS. For these experiments, Go-Solver uses the exact same static safety Go knowledge implementation as EWS, so it statically recognizes the same wins and losses.

In EWS-WR we replace Equation (2) with $EW_{win} := \min(EW_{loss}(C_i))$ to take the minimum child EW instead of a weighted average, and set all node's WR to 0. This results in a negamax PNS algorithm which prioritizes positions closest to being solved. In EWS-PS we order children of each node according to the UCT formula [Kocsis and Szepesvári, 2006]. This algorithm is similar to MCTS solver [Winands *et al.*, 2008], with minor implementation differences.

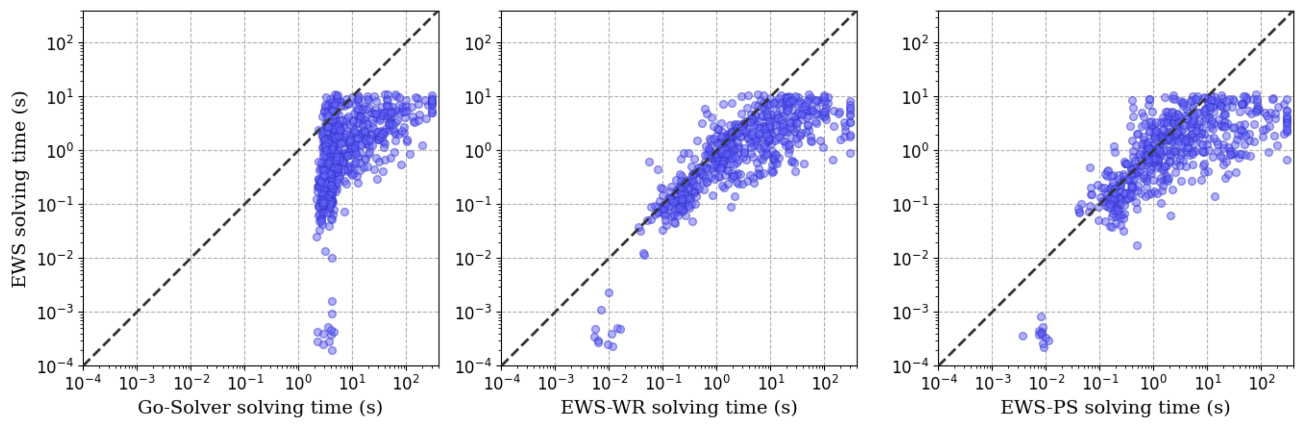
Figure 2 compares EWS against the three baselines on the 6×6Go dataset. Each scatter plot uses doubly-logarithmic scales, with EWS solving time on the y-axis, and the other algorithm's solving time on the x-axis. For data points below the diagonal, EWS is faster. Table 1 shows the summary statistics of these experiments. Go-Solver has a one second initialisation overhead, and was generally slower for harder positions. Go-Solver's average solving time is 9.5× larger than the average time it took EWS to solve the same positions.

EWS-WR is a respectable solving algorithm even without win rate estimation, with performance comparable to Go-Solver. On average EWS-WR took 8.5× longer than EWS to solve the same positions. EWS-PS was faster than EWS-WR and Go-Solver on average, and solved the most positions faster than EWS. However, EWS-PS also exceeded the 5 minute time out period on the most positions and took 8.4× longer than EWS on average. This experiment shows that the performance of EWS degrades when either one of its two main attributes is removed from the search.

Solving Small Square Go Boards

We compared EWS, EWS-WR, EWS-PS, and Go-Solver against the state of the art by solving the empty square Go boards of size 3×3 up to 5×5, shown in Table 2. State of the art is represented by MIGOS, which is accepted as the strongest previously published Go solver [van der Werf *et al.*, 2003] [van der Werf and Winands, 2009].

While the strongest results for MIGOS are for Japanese style rules, it was also tested using situational superko (SSK) rules up to the empty 4×4 board, shown in Table 2. Our Go implementation uses positional superko rules, which differ slightly from SSK as they forbid any positional repetition regardless of the player to move. Both rulesets lead to



(a) EWS vs. Go-Solver (b) EWS vs. EWS-WR (c) EWS vs. EWS-PS

Figure 2: Comparing EWS against Go-Solver, EWS without win rate estimation (EWS-WR), and EWS without proof size estimation (EWS-PS). The y-axis is a log scale of the time EWS took to solve the positions in seconds, and the x-axis is the time it took the other algorithms to solve the same positions. Data points below the diagonal dashed line took less time for EWS to solve than the comparison algorithm.

Program	3×3 Time (s)	3×3 Nodes	4×4 Time (s)	4×4 Nodes	5×5 Time (h)	5×5 Nodes
EWS	0.053	161	13.626	495,494	5.252	2,605,781,360
EWS-WR	0.074	796	40.396	1,562,718	> 24	-
EWS-PS	0.070	232	18.320	744,169	> 24	-
Go-Solver	1.299	1,628	51.522	799,607	> 24	-
MIGOS SSK	< 3.3	$\sim$ 25,118	$\sim$3,960	$\sim$3,162,277,660	-	-

Table 2: The results from solving empty n×n Go boards with EWS, EWS without win rate estimation (EWS-WR), EWS without proof size estimation (EWS-PS) and the $\alpha\beta$ Go-Solver program. 5×5 Time is in hours. MIGOS times and node counts are approximate. All programs use positional superko other than MIGOS which uses situational superko. 3×3 has 8.5 komi, 4×4 has 1.5 komi, 5×5 has 24.5 komi.

Program	4×4 Time (s)	4×4 Nodes	5×5 Time (s)	5×5 Nodes	6×6 Time (h)	6×6 Nodes
EWS	0.002	283	0.253	37,034	0.422	93,963,192
Enhanced AB	0.004	1,673	3.935	698,402	> 24	-
Morat MCTS	0.035	4,644	29.669	3,554,546	> 24	-
Morat PNS	0.054	8,871	758.102	154,539,591	> 24	-

Table 3: The results from solving the empty n×n Hex board without extensive domain-specific knowledge with EWS, an enhanced $\alpha\beta$ program, Morat MCTS, and Morat PNS. 6×6 Time is in hours.

Program	6×6 Time (s)	6×6 Nodes	7×7 Time (s)	7×7 Nodes	8×8 Time (s)	8×8 Nodes
EWS with knowledge	0.005	26	0.588	2,318	234.449	555,158

Table 4: The results from solving the empty n×n Hex board using domain-specific knowledge with EWS.

GHI problems [Campbell, 1985], making them comparable for solving.

In our experiments EWS outperforms the other tested algorithms and was the only program to solve 5×5 Go, with 24.5 komi, within 24 hours. For the first time, 5×5 Go has been solved with the positional superko ruleset, finding a full-board first player win. We naively parallelized the programs when solving 5×5 Go by running 6 parallel processes to solve the positions resulting from the center opening move for Black and the 6 symmetrically unique responses from White. No other experiments reported here used any parallel computing.

5.3 Hex

To show the generality of EWS, we also evaluated it on the game of Hex. Table 3 shows the results of solving empty n×n Hex boards using little Hex-specific knowledge. We compare EWS against the publicly available Morat program [Ewalds, 2012] which includes PNS and MCTS algorithms to play and solve Hex. Morat PNS implements the Depth First Proof Number search variant of PNS, along with refinements such as the $1 + \epsilon$ trick [Pawlewicz and Lew, 2006]. We also implemented an enhanced $\alpha\beta$ algorithm to solve Hex. All programs tested in the results shown in Table 3 have the same limited Hex specific knowledge.

The results in Table 3 show that EWS is faster and creates smaller proof trees than any of the other tested algorithms. For solving 5×5 Hex, EWS was 15.5× faster than enhanced $\alpha\beta$, 117.3× faster than Morat MCTS, and 2996.5× faster than Morat PNS. EWS was the only program to solve 6×6 Hex in the 24 hour time limit allotted.

6×6 Hex is a significantly smaller problem than the largest solved board size, 10×10 Hex [Pawlewicz and Hayward, 2013]. To show the importance of domain-specific knowledge in Hex, we implemented Hex specific knowledge for EWS as shown in Table 4: basic virtual connections which identify terminal positions earlier, and fill-in / inferior cell patterns which prune dominated moves [Hayward *et al.*, 2004].

Our results emphasize the importance of domain-specific knowledge for solving large Hex problems. In Table 3, EWS explored over 93 million nodes to solve 6×6 Hex with no domain knowledge. With basic Hex-specific knowledge, EWS takes 26 nodes, and solves 8×8 Hex in less than 4 minutes. Compared to [Pawlewicz and Hayward, 2013], our current Hex knowledge implementation lacks the more complicated virtual connections and fill-in / inferior cell patterns among other techniques, limiting our ability to efficiently solve larger Hex boards.

6 Related Work

The PN-MCTS algorithm [Doe *et al.*, 2022] is a game playing program which has shown favourable results compared to unenhanced MCTS on a number of games. PN-MCTS is MCTS with a modified UCT formula that uses proof and disproof numbers to help guide move selection. Proof and disproof numbers are used to rank children, which is used in a normalized term added to the UCT formula which influences move selection. All information in the proof and disproof numbers other than the relative ordering of children's proof numbers is discarded.

A major difference from EWS is that PN-MCTS keeps its win rate and proof number-based heuristics separate, and only combines them through adding terms to UCT during move selection. In contrast, EW integrates win rate and proof number-like information into a single Expected Work estimate, which is computed recursively. EWS is also able to consider the estimated proof sizes of positions rather than just considering children's relative ordering. Another main difference between PN-MCTS and EWS is that PN-MCTS was designed to be a game-playing program, while EWS is a game solver. Since the published PN-MCTS algorithm is not currently able to solve games, we cannot make any empirical comparisons against this work.

Product propagation [Saffidine and Cazenave, 2013] modifies PNS by backing up heuristic estimates of the likelihood of winning rather than proof and disproof numbers. This modification replaces the normal proof size estimation which occurs in PNS, with the goal of achieving a stronger heuristic for the search. Product propagation search was empirically shown to outperform $\alpha\beta$ search and PNS in some of the games tested.

Previous work by van der Werf et al. on solving Go has solved rectangular boards up to 5×6 and 4×7 [van der Werf *et al.*, 2003] [van der Werf and Winands, 2009] using Japanese style rules. Their MIGOS solver uses iterative deepening $\alpha\beta$ with handcrafted heuristics and Go-specific knowledge. Japanese style rules were used in order to avoid the Graph History Interaction problem [Campbell, 1985]. With situational superko, MIGOS was able to solve up to the empty 4×4 board.

The first computer solution of 7×7 Hex by Hayward et al. relied on advances in the Hex-specific knowledge of virtual connections and move domination [Hayward *et al.*, 2004]. Improvements on inferior cell analysis and pattern decomposition allowed 8×8 Hex to be solved [Henderson *et al.*, 2009]. Increasing the strength of virtual connections with captured cell knowledge and switching to a DFPN algorithm allowed solving 9×9 Hex [Arneson *et al.*, 2010]. Finally, a large-scale parallel computation solved 10×10 Hex [Pawlewicz and Hayward, 2013].

7 Conclusions and Future Work

EWS successfully combines win rate and proof size estimation in an efficient search algorithm. Our experiments show that EWS can outperform $\alpha\beta$, PNS, and MCTS based programs in Go and Hex. We present the new result of solving 5×5 Go with positional superko, and demonstrate that EWS can solve 8×8 Hex when given basic Hex-specific knowledge.

For future work, we plan to test machined learned evaluation functions for win rate estimation [Silver *et al.*, 2016] [Silver *et al.*, 2018] and EW initialisation [Wu *et al.*, 2022]. We will continue to develop our Hex specific knowledge with the goal of obtaining new results in solving Hex. Finally, we hope to investigate translating EW into a concrete prediction of the resources it will take to solve problems.

References

[Allis *et al.*, 1994] L.Victor Allis, Maarten van der Meulen, and H.Jaap van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.

[Arneson *et al.*, 2010] Broderick Arneson, Ryan Hayward, and Philip Henderson. Solving Hex: Beyond Humans. In *Computers and Games*, pages 1–10, 2010.

[Browne *et al.*, 2012] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[Campbell, 1985] Murray Campbell. The Graph-History Interaction: On Ignoring Position History. In *ACM Annual Conference on the Range of Computing*, pages 278–280, 1985.

[Doe *et al.*, 2022] Elliot Doe, Mark H. M. Winands, Jakub Kowalski, Dennis J. N. J. Soemers, Daniel Górski, and Cameron Browne. Proof Number Based Monte-Carlo Tree Search. In *IEEE Conference on Games*, pages 206–212, 2022.

[Enzenberger *et al.*, 2011] Markus Enzenberger, Marcus Müller, Broderick Arneson, and Richard Segal. Fuego—An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:259–270, 2011.

[Ewalds, 2012] Timo V Ewalds. Playing and Solving Havannah, 2012. *University of Alberta Master's Thesis* (2012), https://github.com/tewalds/morat.

[Fawzi *et al.*, 2022] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning. *Nature*, pages 47–53, 2022.

[Hayward *et al.*, 2004] R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, and J. van Rijswijck. Solving 7×7 Hex: Virtual Connections and Game-State Reduction. *Advances in Computer Games: Many Games, Many Challenges*, pages 261–278, 2004.

[Henderson *et al.*, 2009] Philip Henderson, Broderick Arneson, and Ryan Hayward. Solving 8x8 Hex. *IJCAI International Joint Conference on Artificial Intelligence*, pages 505–510, 2009.

[Kishimoto and Müller, 2004] Akihiro Kishimoto and Martin Müller. A General Solution to the Graph History Interaction Problem. In *Proceedings of the 19th National Conference on Artifical Intelligence*, pages 644–649, 2004.

[Kishimoto *et al.*, 2012] Akihiro Kishimoto, Mark Winands, Martin Müller, and Jahn Takeshi Saito. Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal*, 35:131–156, 2012.

[Knuth and Moore, 1975] Donald E. Knuth and Ronald W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML*, pages 282–293, 2006.

[Nagai, 2002] A. Nagai. DF-PN Algorithm for Searching AND/OR Trees and its Applications. *University of Tokyo Ph.D. Thesis*, 2002.

[Pawlewicz and Hayward, 2013] Jakub Pawlewicz and Ryan B. Hayward. Scalable Parallel DFPN Search. In *Computers and Games*, pages 138–150, 2013.

[Pawlewicz and Lew, 2006] Jakub Pawlewicz and Lukasz Lew. Improving Depth-First PN-Search: 1 + Epsilon Trick. In *Proceedings of the 5th International Conference on Computers and Games*, pages 160–171, 2006.

[Randall *et al.*, 2023] Owen Randall, Ting-Han Wei, Ryan Hayward, and Martin Müller. Improving Search in Go Using Bounded Static Safety. In *Computers and Games*, pages 14–23, 2023.

[Randall *et al.*, 2024] Owen Randall, Martin Müller, Ting Han Wei, and Ryan Hayward. Expected work search: Combining win rate and proof size estimation, 2024. (https://arxiv.org/abs/2405.05594).

[Rosin, 2010] Christopher Rosin. Multi-armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2010.

[Saffidine and Cazenave, 2012] A. Saffidine and T. Cazenave. Multiple-outcome proof number search. In *ECAI*, pages 708–713. IOS Press, 2012.

[Saffidine and Cazenave, 2013] Abdallah Saffidine and Tristan Cazenave. Developments on Product Propagation. In *Computers and Games*, pages 100–109, 2013.

[Schaeffer and Plaat, 1996] Jonathan Schaeffer and Aske Plaat. New Advances in Alpha-Beta Searching. In *Proceedings of the 1996 ACM 24th annual Conference on Computer Science*, pages 124–130, 1996.

[Shih *et al.*, 2021] Chung-Chin Shih, Ti-Rong Wu, Ting Han Wei, and I-Chen Wu. A Novel Approach to Solving Goal-Achieving Problems for Board Games. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, pages 10362–10369, 2021.

[Silver *et al.*, 2016] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529:484–489, 2016.

[Silver *et al.*, 2018] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science*, 362(6419):1140–1144, 2018.

[Slate and Atkin, 1977] David Slate and Larry Atkin. Chess Skill in Man and Machine. In *Springer-Verlag*, pages 82–118, 1977.

[Sutton and Barto,] R. S. Sutton and A. G. Barto. Reinforcement Learning: An Introduction. *MIT Press* (1998), 39-40.

[van der Werf and Winands, 2009] Erik C. D. van der Werf and Mark H. M. Winands. Solving Go for Rectangular Boards. *ICGA Journal*, 32(2):77–88, 2009.

[van der Werf *et al.*, 2003] Erik van der Werf, H. Herik, and Jos Uiterwijk. Solving Go on Small Boards. *ICGA Journal*, 26:92–107, 2003.

[Winands *et al.*, 2008] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search Solver. In *Computers and Games*, pages 25–36, 2008.

[Wu *et al.*, 2022] Ti-Rong Wu, Chung-Chin Shih, Ting-Han Wei, Meng-Yu Tsai, Wei-Yuan Hsu, and I-Chen Wu. AlphaZero-based Proof Cost Network to Aid Game Solving. In *International Conference on Learning Representations*, 2022.

[Yao *et al.*, 2023] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, (2023).

[Zobrist, 1990] Albert Zobrist. A New Hashing Method with Application for Game Playing. *ICGA Journal*, 13:69–73, 1990.