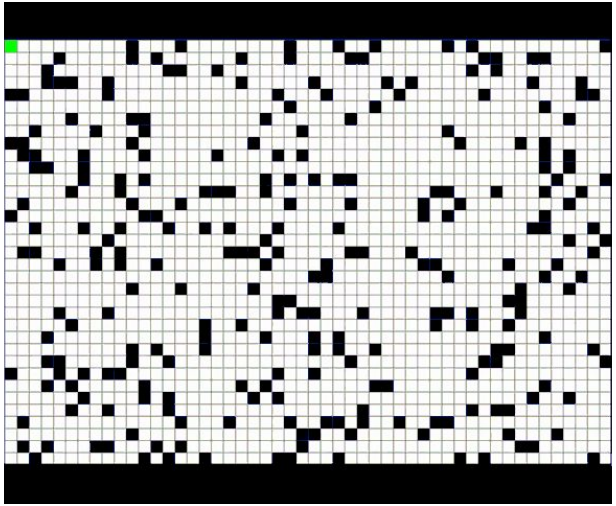
Efficiently Solving Games with Expected Work Search

Owen Randall, Martin Müller, Ting-Han Wei, and Ryan Hayward

Introduction

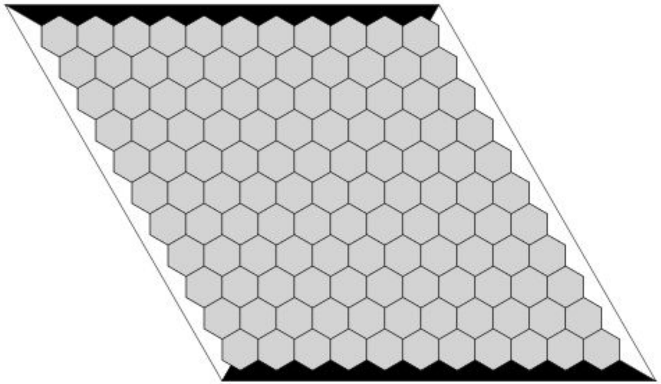
- Expected Work Search (**EWS**) is a new combinatorial search algorithm
    - Not a search engine like Google or Bing
    - Searches for solutions to problems with many possible positions
- Use **heuristics** to efficiently guide search
    - Can exponentially improve performance
- EWS can outperform existing algorithms
    - **3.8x** to **291x** faster when solving Go
    - **2.0x** to **3032x** faster when solving Hex

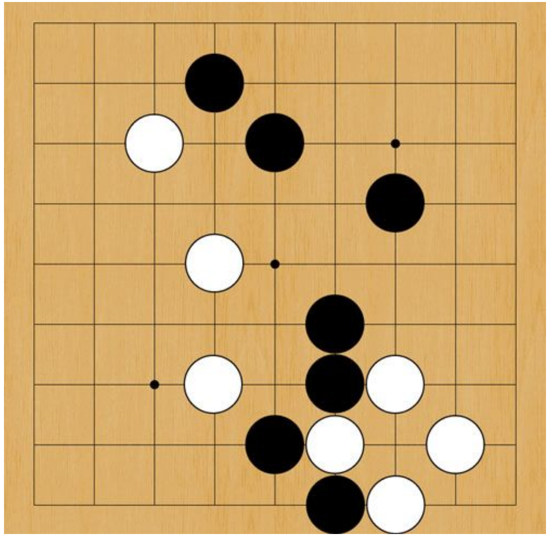


https://en.wikipedia.org/wiki/A*_search_algorithm#/media/File:Astarpathfinding.gif

Solving

- Evaluate EWS by **solving** two player perfect information games
  - Specifically Go and Hex
  - Well defined rules
  - Scalable difficulty
- Solving games prove who wins under **optimal play**
  - Any winning move = winning position
  - All losing moves = losing position

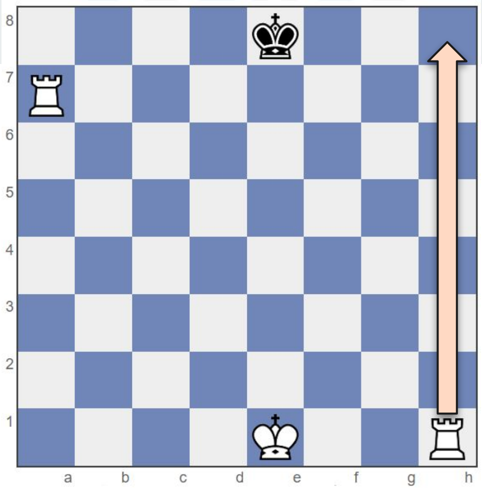


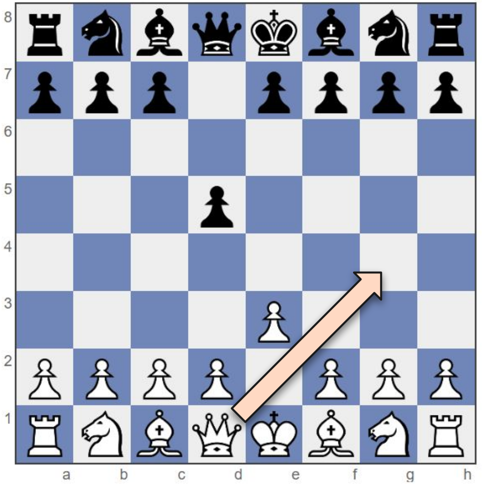


https://github.com/sotetsuk/pgx?tab=readme-ov-file https://www.irish-go.org/how-to-play-go/

Heuristics

- Win rate estimation
  - Monte Carlo Tree Search
  - Prioritizes strong moves
- Proof size estimation
  - Proof number search
  - Prioritizes quickly ending the game
- Both have strengths and weaknesses
- EWS combines both using Expected Work

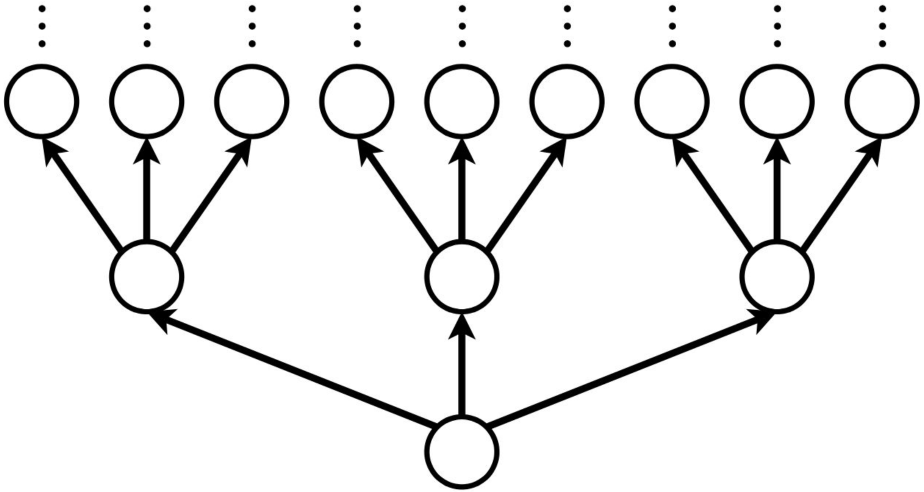




https://www.365chess.com/board_editor.php

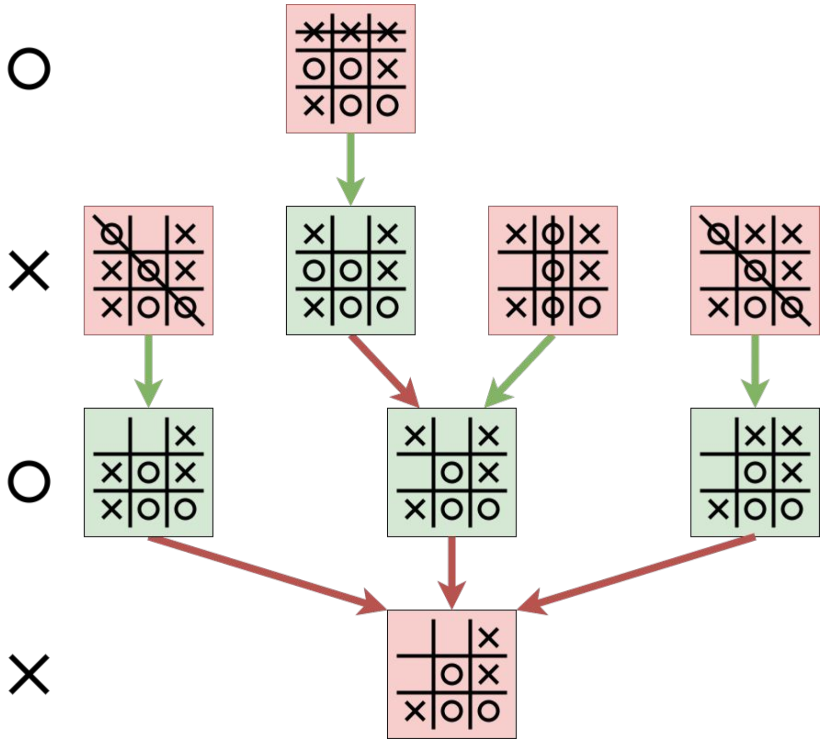
Expected Work Search

- Build a node tree in memory
- Nodes represent positions
- Each node has a win rate, and Expected Work estimate
- Each iteration of EWS builds the tree, refines node estimates, and backs up proofs



Expected Work Calculation

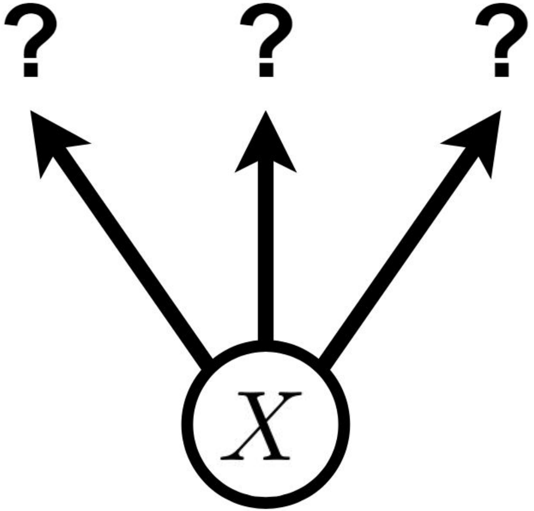
- How can we estimate the work it will take to solve a node?
- Inductively assume children have expected work values
- Separate cases for winning & losing
- Use MCTS style win rates to estimate winning probability
- Base case expected work values come from random simulations



Expected Work Calculation

- A position is either winning or losing

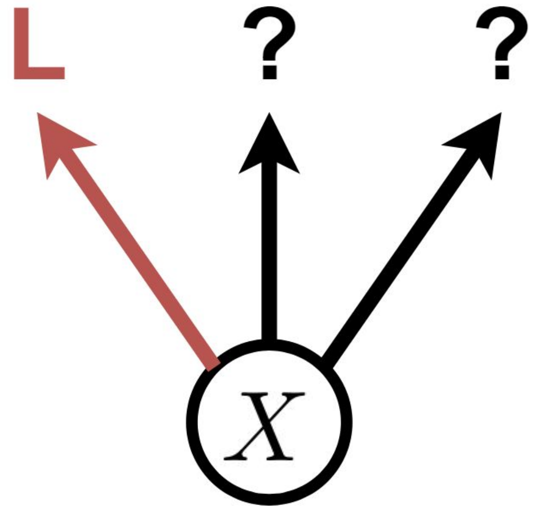
- Losing case:



Expected Work Calculation

- A position is either winning or losing

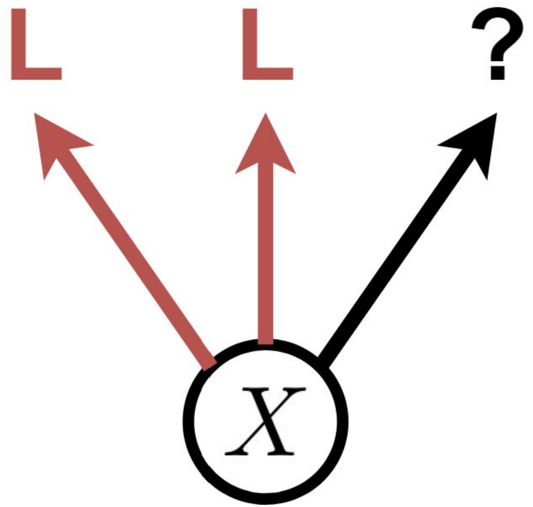
- Losing case:



Expected Work Calculation

- A position is either winning or losing

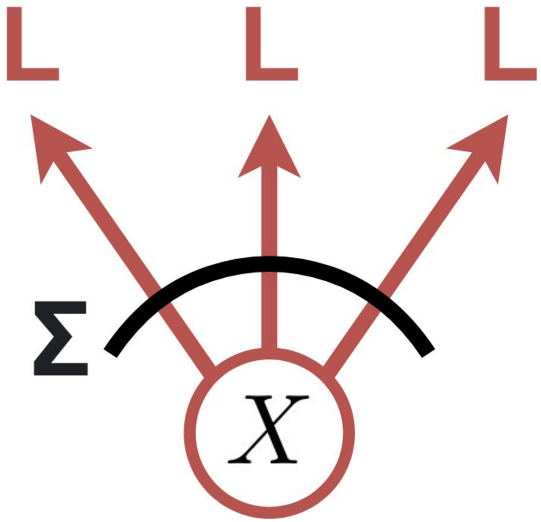
- Losing case:



Expected Work Calculation

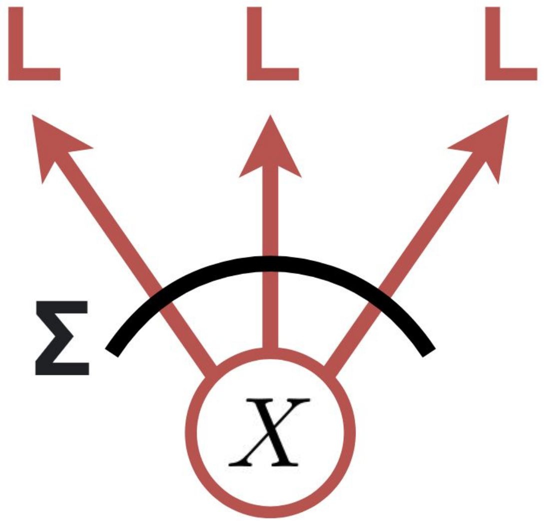
- A position is either winning or losing

- Losing case:



Expected Work Calculation

- A position is either winning or losing
- Losing case:
- Sum of winning children's EW

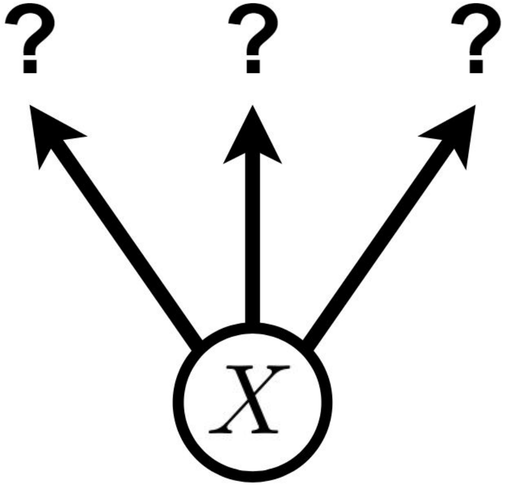


$$EW_{loss}(X) := \sum_{i=1}^{n} EW_{win}(C_i)$$

Expected Work Calculation

- A position is either winning or losing

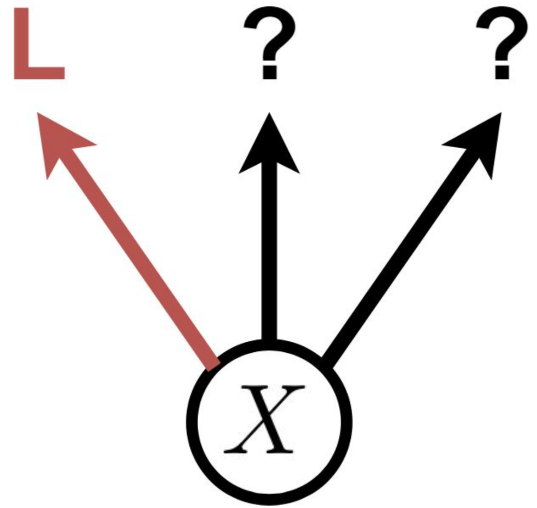
- Winning case:



Expected Work Calculation

- A position is either winning or losing

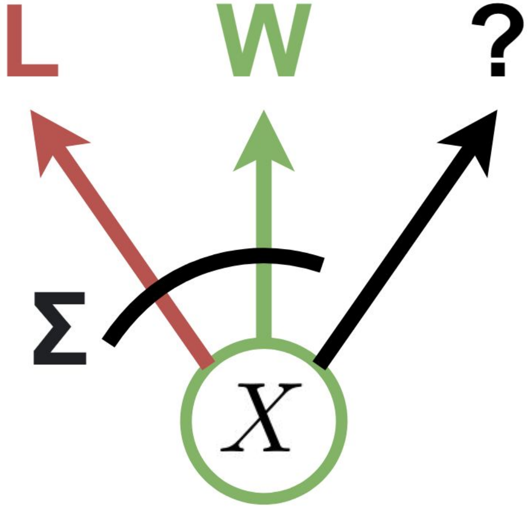
- Winning case:



Expected Work Calculation

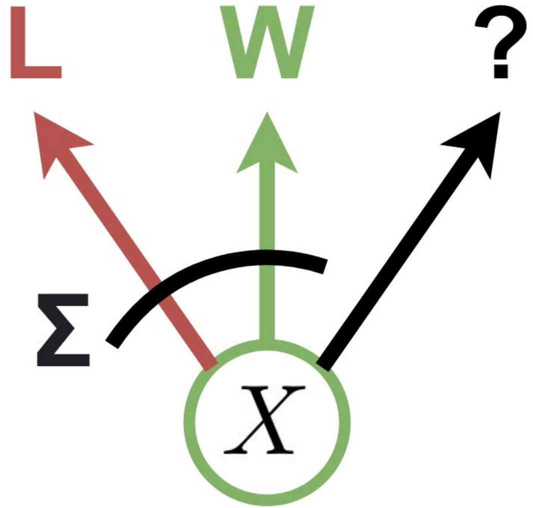
- A position is either winning or losing

- Winning case:



Expected Work Calculation

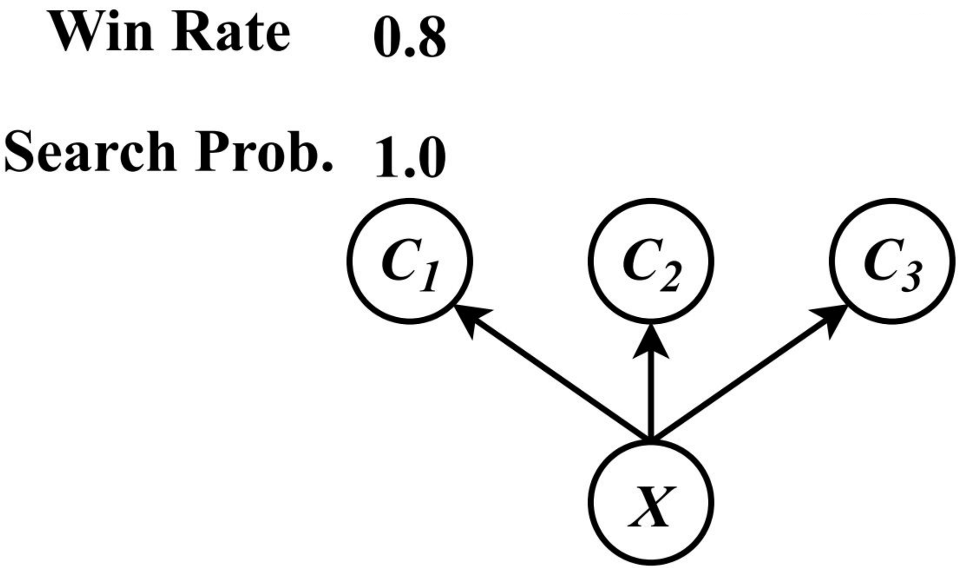
- A position is either winning or losing
- Winning case:
- **Weighted** sum of losing children's EW



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

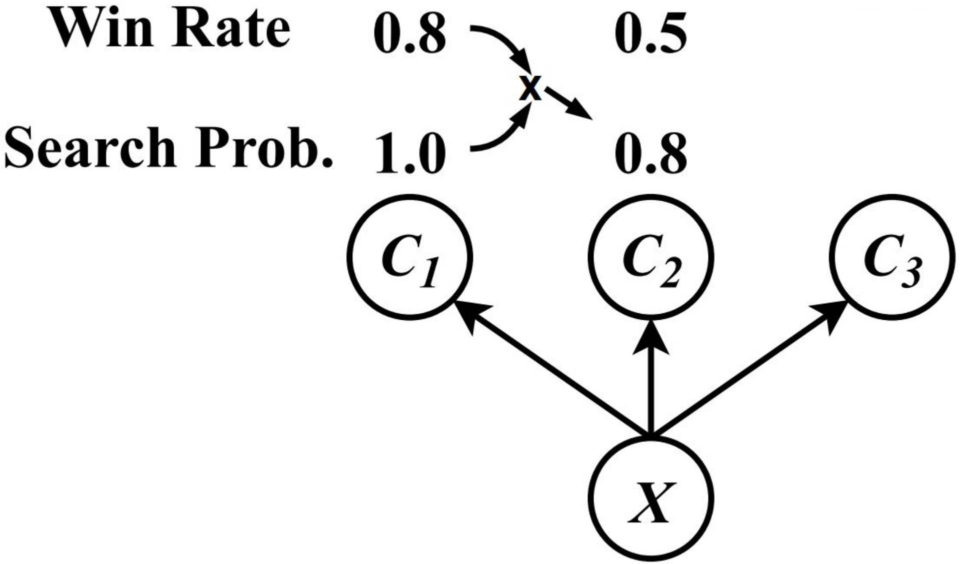
$$EW_{win}(X) := \sum_{i=0}^{n} \left(EW_{loss}(C_i) \cdot \prod_{j=0}^{i-1} WR(C_j) \right) \qquad (2)$$

- Children are searched in order
- Search stops once a losing child is found
- The probability of searching a child = the probability none of the previous children are losing



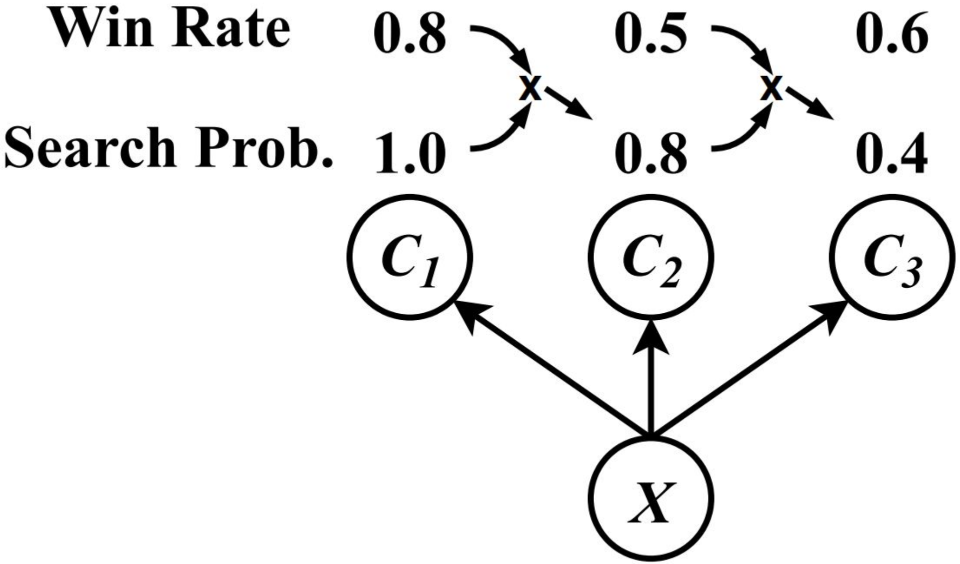
$$EW_{win}(X) := \sum_{i=0}^{n} (EW_{loss}(C_i) \cdot \prod_{j=0}^{i-1} WR(C_j)) \quad (2)$$

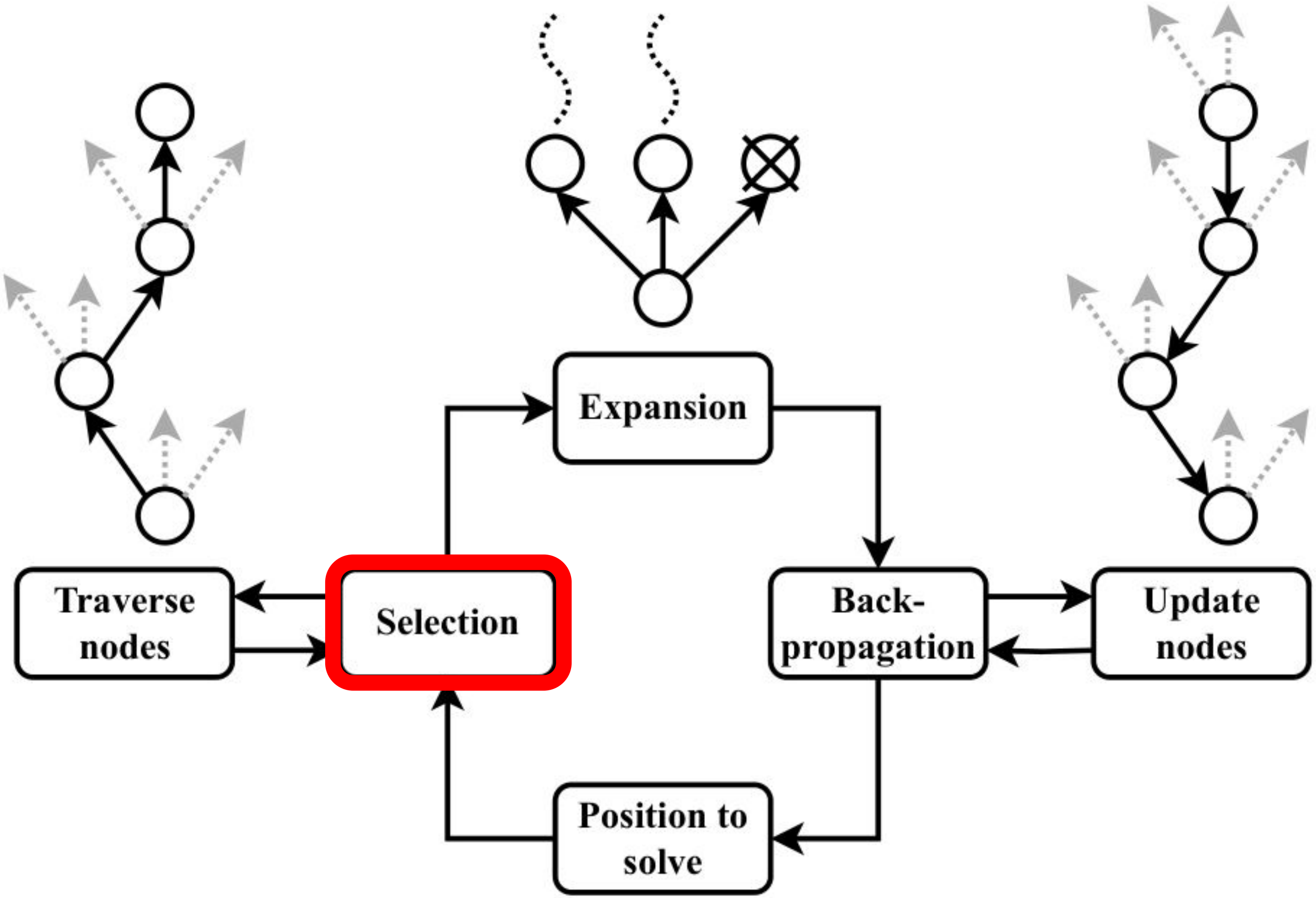
- Children are searched in order
- Search stops once a losing child is found
- The probability of searching a child = the probability none of the previous children are losing



$$EW_{win}(X) := \sum_{i=0}^{n} \left(EW_{loss}(C_i) \cdot \prod_{j=0}^{i-1} WR(C_j)\right) \quad (2)$$

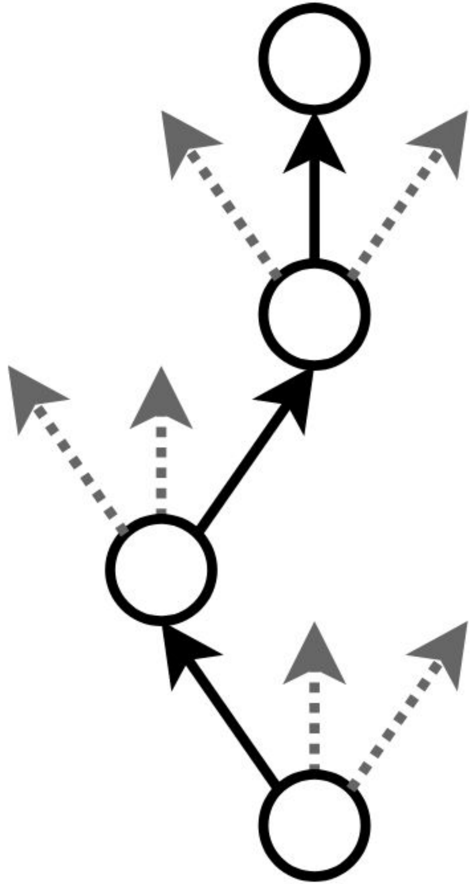
- Children are searched in order
- Search stops once a losing child is found
- The probability of searching a child = the probability none of the previous children are losing




Expansion
Traverse nodes
Selection
Back-propagation
Update nodes
Position to solve

Selection

- Traverse the search tree
- Select the best child according to the move ordering
- If the selected child has been expanded, continue selection
- Otherwise, expand the leaf node



Child Ordering

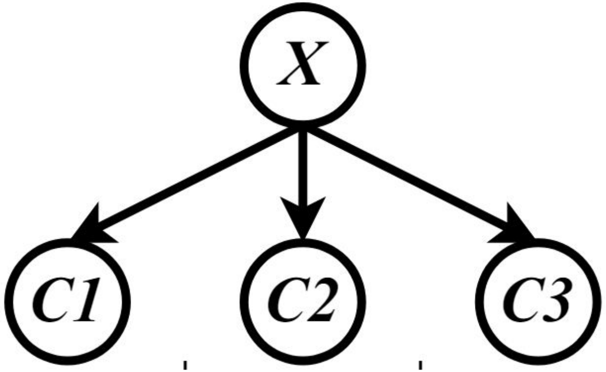
- Sort children with:

$$EW_{loss}(A)/(1-WR(A)) < EW_{loss}(B)/(1-WR(B))$$
$$\iff$$
$$A < B$$

- Best first move ordering
- Move ordering of children determines EW_{win}
    - **This ordering minimizes EW_{win}**

Child Ordering

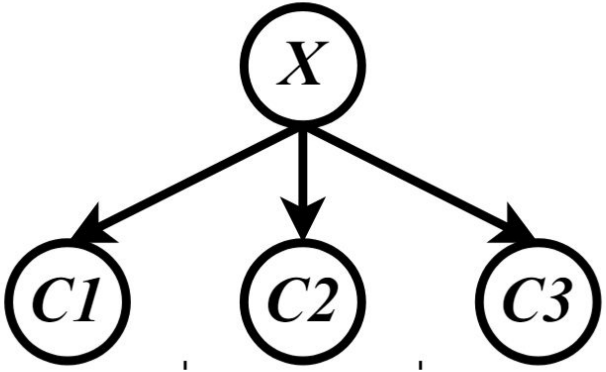
$$EW_{loss}(A)/(1-WR(A)) < EW_{loss}(B)/(1-WR(B))$$



	C1	C2	C3
WR	0.6	0.2	0.15
EW_{loss}	900	1000	1500

Child Ordering

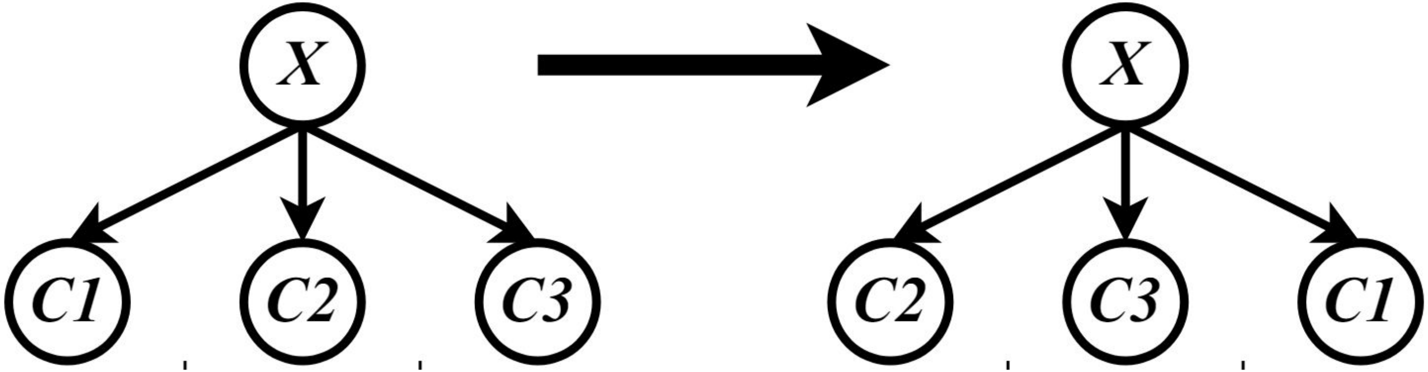
$$EW_{loss}(A)/(1-WR(A)) < EW_{loss}(B)/(1-WR(B))$$



WR	0.6	0.2	0.15
EW_{loss}	900	1000	1500
EW/(1-WR)	**2250**	**1250**	**1765**

Child Ordering

$$EW_{loss}(A)/(1-WR(A)) < EW_{loss}(B)/(1-WR(B))$$



WR	0.6	0.2	0.15
EW_{loss}	900	1000	1500
EW/(1-WR)	**2250**	**1250**	**1765**

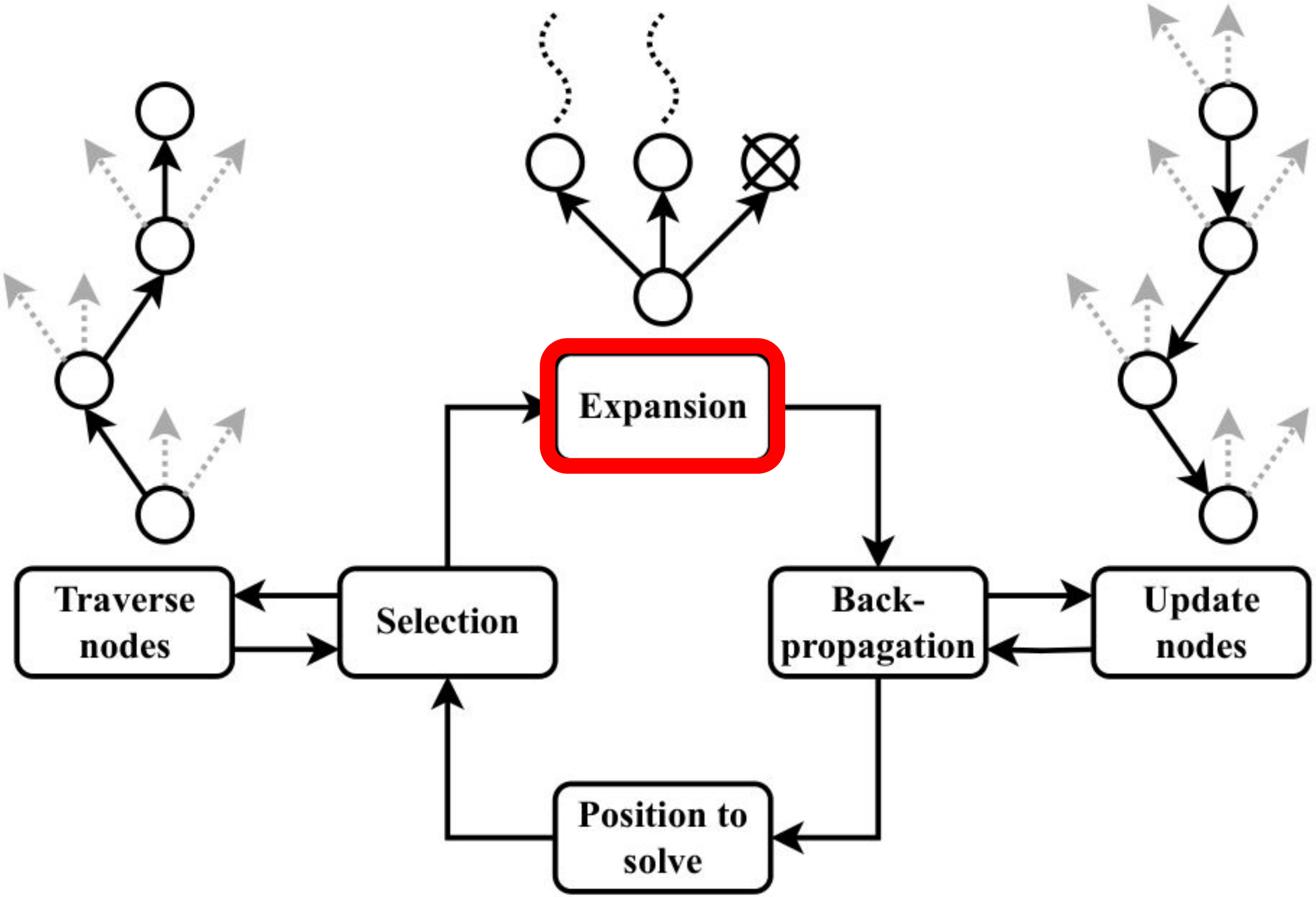
0.2	0.15	0.6
1000	1500	900
1250	**1765**	**2250**

Selection

- Traverse the search tree
- Select the best child according to the move ordering
- If the selected child has been expanded, continue selection
- Otherwise, expand the leaf node

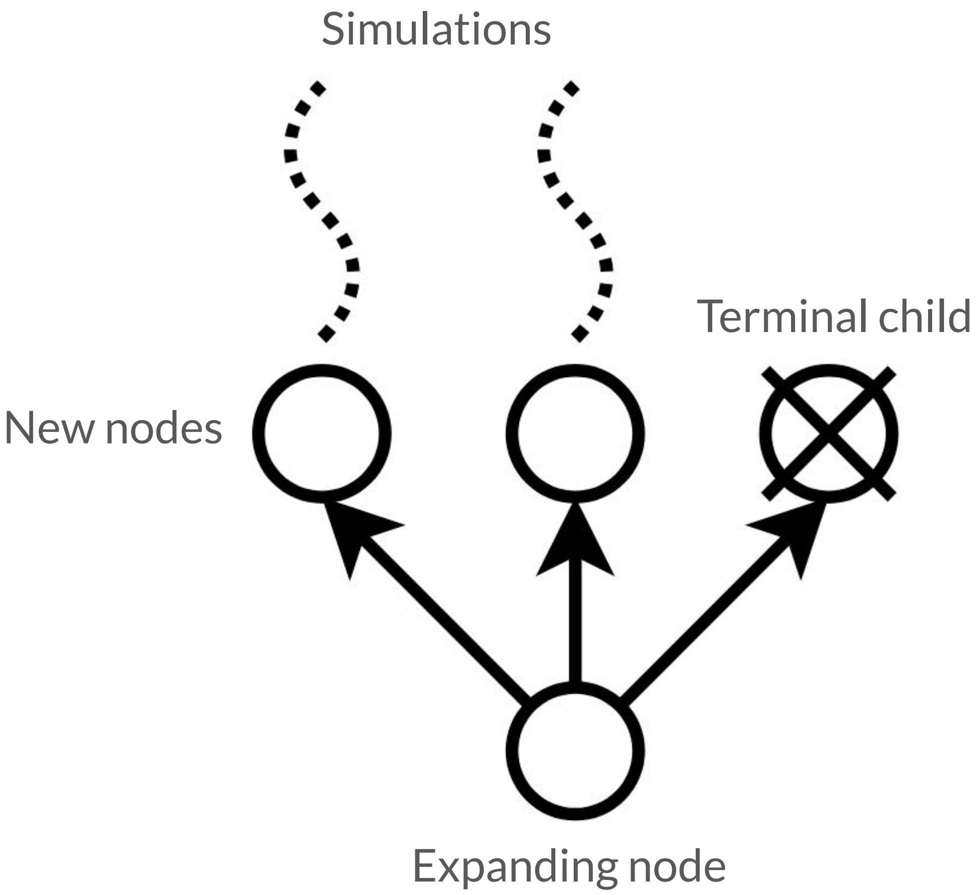
Algorithm 1 SelectBackpropagate X: returns whether X is solved and if so whether X is winning

```
C := X.children[0]                                        ▷ Selection
Make move C.move
if C.expanded then
    isSolved, isWinning = SelectBackpropagate(C)
else
    isSolved, isWinning = Expand(C)
Undo move C.move                                          ▷ Backpropagation
if isSolved and isWinning then
    Remove C from X.children
    if X.children is empty then
        return true, false                                ▷ Solved loss
else if isSolved and not isWinning then
    return true, true                                     ▷ Solved win
Sort X.children with (3)
Update X.winRate
Update X.expectedWorkLoss with (1)
Update X.expectedWorkWin with (2)
return false, false                                       ▷ Unsolved
```


Expansion
Traverse nodes
Selection
Back-propagation
Update nodes
Position to solve

Expansion

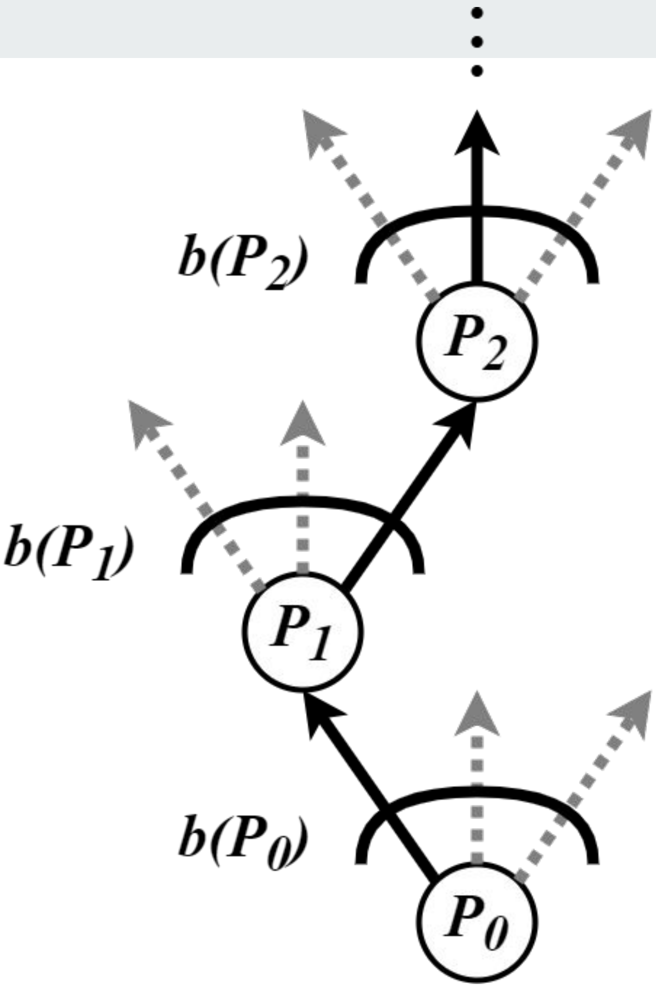
- Create new nodes
- Check for terminal children
- Initialize WR and EW with random simulation results
- Return if the expanded node is proven



Initial Evaluation

- Random simulations
- Win rates = wins / visits
- Initialize EW as the sum of the branching factors of positions in random simulations

$$EW_0(X) := \sum_{i=0}^{m} b(P_i)$$

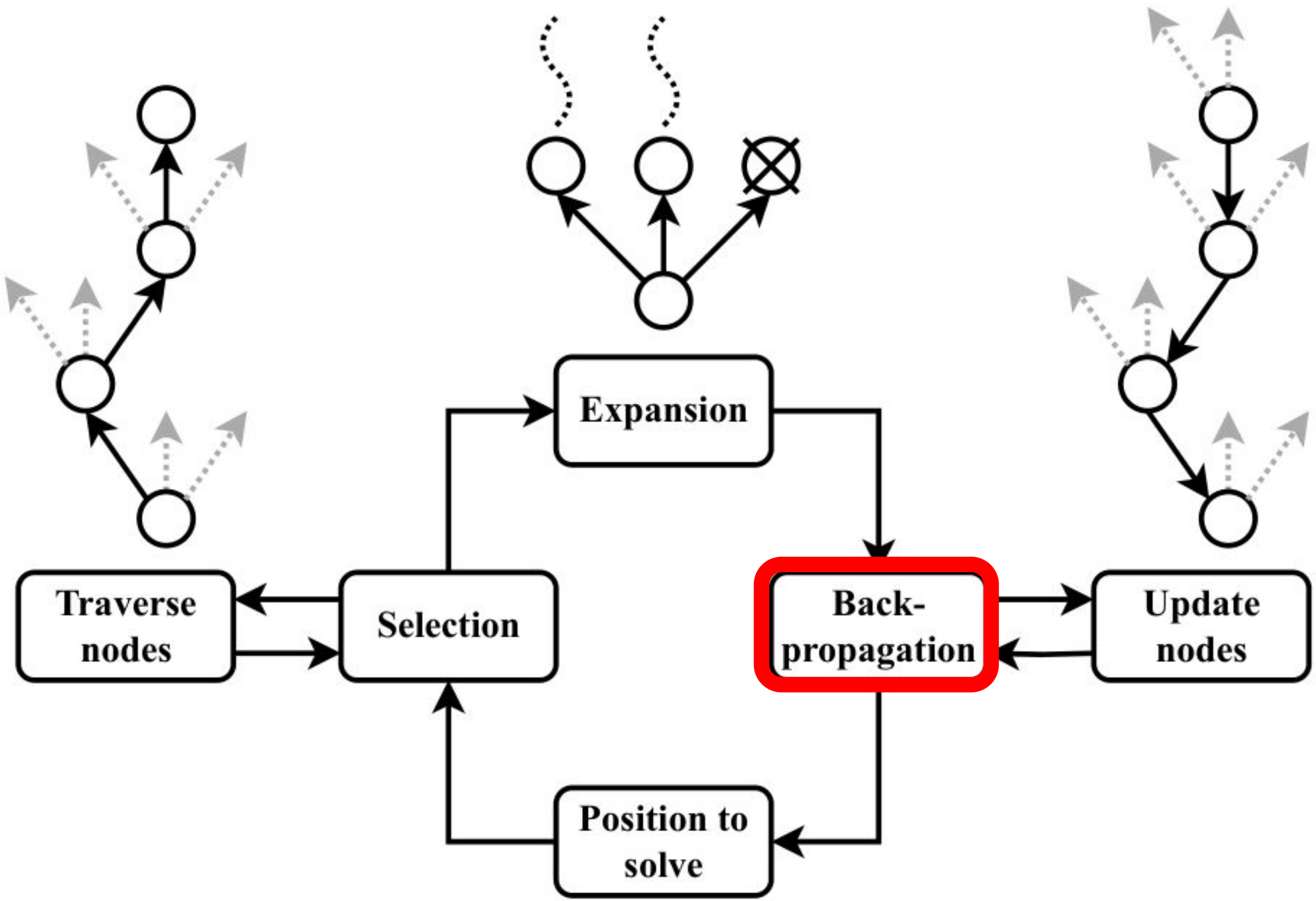


Expansion

- Create new nodes
- Check for terminal children
- Initialize WR and EW with random simulation results
- Return if the expanded node is proven

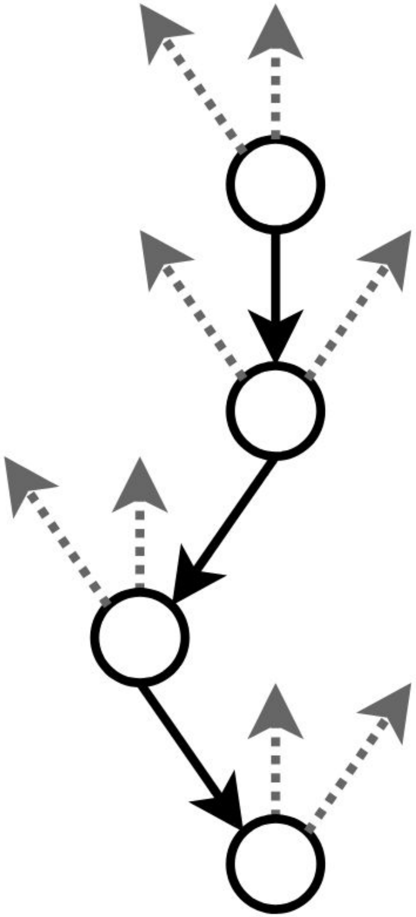
Algorithm 2 Expand X: returns whether X is solved and if so whether X is winning

```
X.expanded := true
for all legal moves m do
    if m is a terminal winning move for X then
        return true, true                          ▷ Solved win
    else if m is not a terminal losing move for X then
        Create node C
        C.expanded = false
        C.move := m
        Evaluate C.winRate
        Evaluate C.expectedWorkLoss
        Evaluate C.expectedWorkWin
        Add C to X.children
if X.children is empty then
    return true, false                             ▷ Solved loss
else
    return false, false                            ▷ Unsolved
```


Expansion
Traverse nodes
Selection
Back-propagation
Update nodes
Position to solve

Backpropagation

- Back up new information along the path chosen by selection:
- Win rate
- Expected Work
- Proofs
- Proven nodes are pruned



Backpropagation

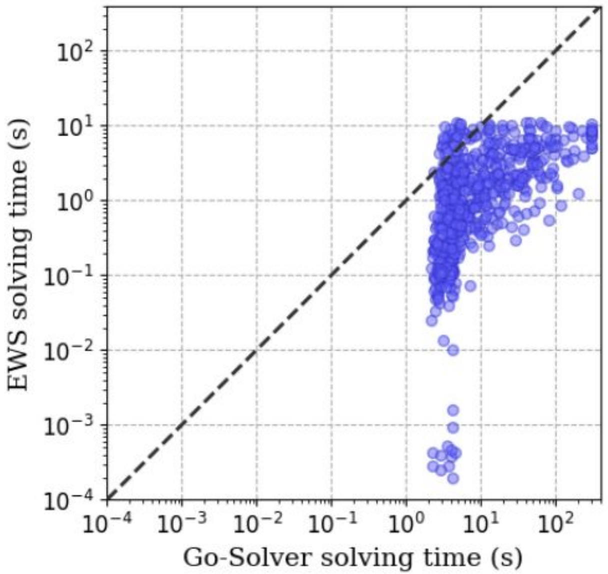
- Back up new information along the path chosen by selection:
- Win rate
- Expected Work
- Proofs
- Proven nodes are pruned

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

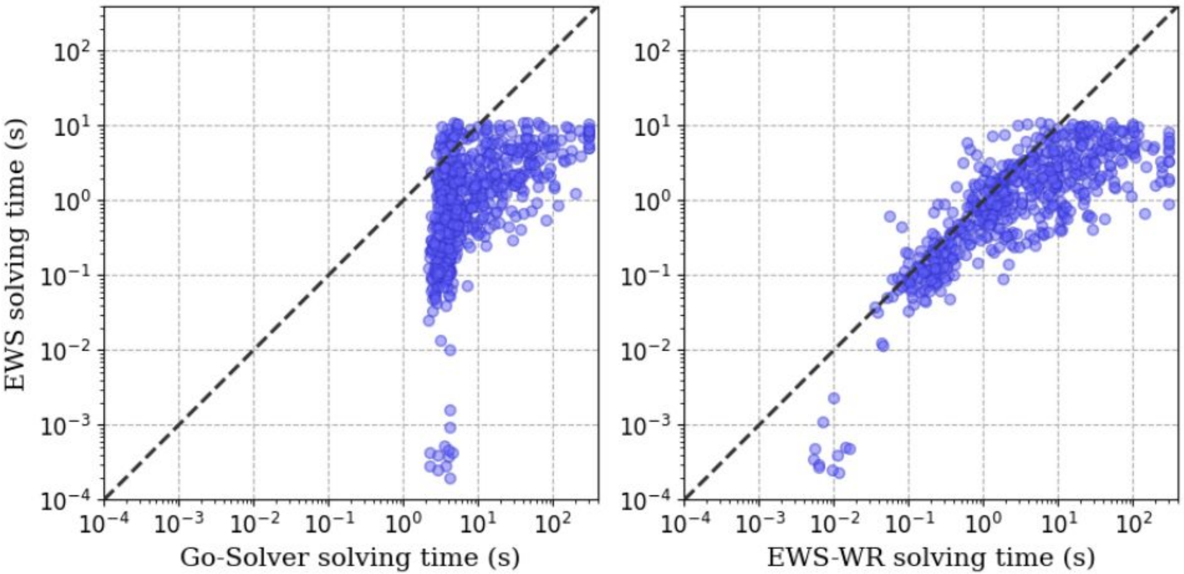
```
C := X.children[0]                                   ▷ Selection
Make move C.move
if C.expanded then
    isSolved, isWinning = SelectBackpropagate(C)
else
    isSolved, isWinning = Expand(C)
Undo move C.move                                     ▷ Backpropagation
if isSolved and isWinning then
    Remove C from X.children
    if X.children is empty then
        return true, false                           ▷ Solved loss
else if isSolved and not isWinning then
    return true, true                                ▷ Solved win
Sort X.children with (3)
Update X.winRate
Update X.expectedWorkLoss with (1)
Update X.expectedWorkWin with (2)
return false, false                                  ▷ Unsolved
```

Results

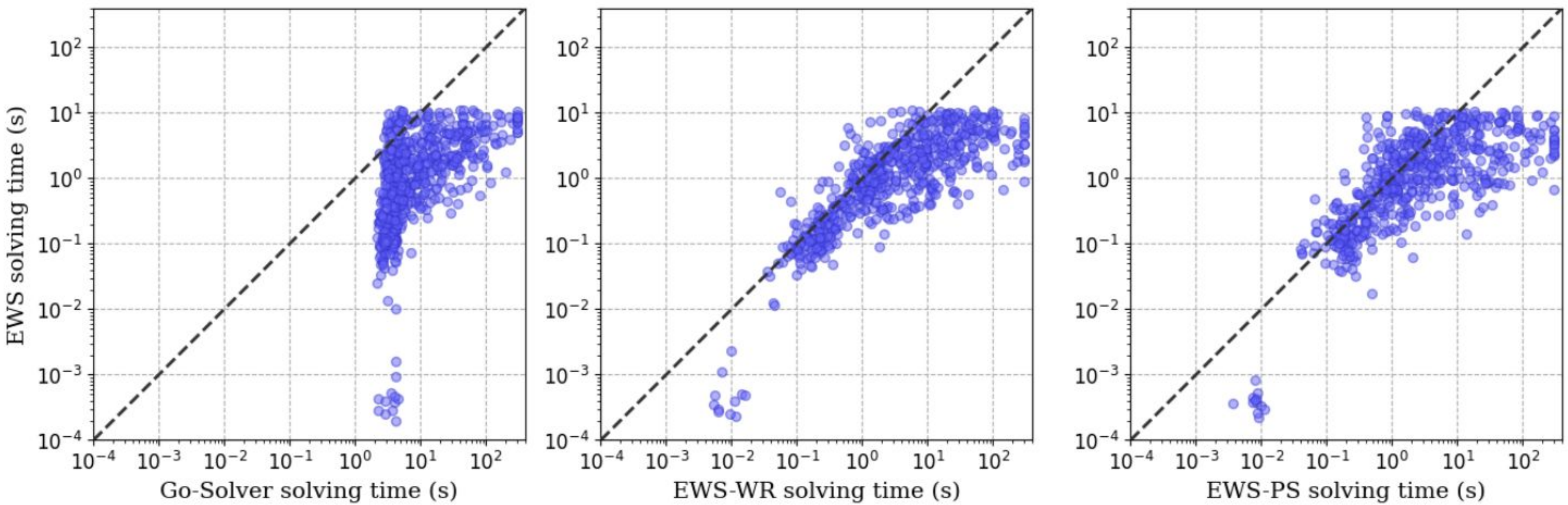
- Compared against different independent solving implementations
    - GoSolver, MIGOS, Enhanced AB, Morat PNS, Morat MCTS
- Ablation study
    - Removed proof-size / win-rate information
    - Performance drops
- Evaluated a variety of Go positions
    - Strong general performance
    - New results on empty boards
- Evaluated empty nxn Hex
    - Strong results with and without knowledge



Program	Av. solve time (s)	# Faster than EWS	# Timeouts
EWS	2.32	-	-
Go-Solver	22.63	31 (5.2%)	11 (1.8%)



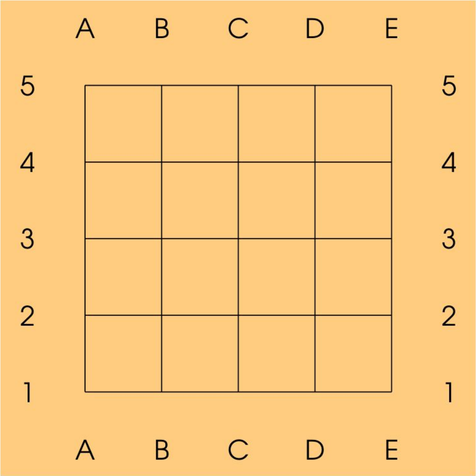
Program	Av. solve time (s)	# Faster than EWS	# Timeouts
EWS	2.32	-	-
Go-Solver	22.63	31 (5.2%)	11 (1.8%)
EWS-WR	19.70	96 (16.0%)	11 (1.8%)



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%)

Solving Square Go Boards

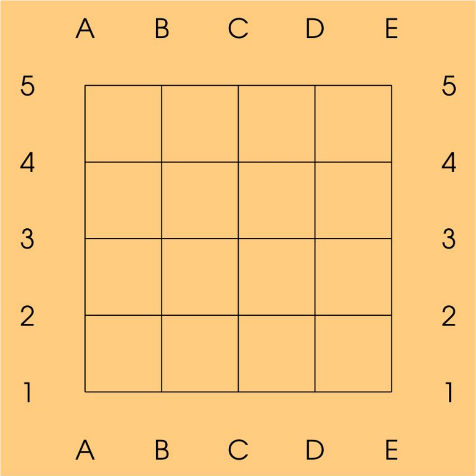
- Compare against:
    - EWS-WR (without win rate estimation)
    - EWS-PS (without proof size estimation)



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	-

Solving Square Go Boards

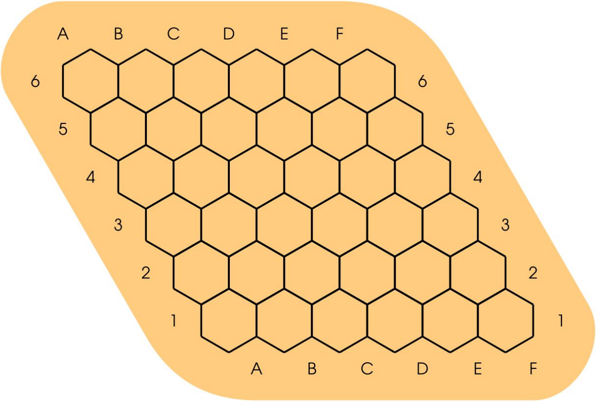
- Compare against:
  - EWS-WR (without win rate estimation)
  - EWS-PS (without proof size estimation)
  - Go-Solver (iterative deepening alpha beta)
  - MIGOS (state of the art using Japanese rules)



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	-	-

Solving Square Hex Boards

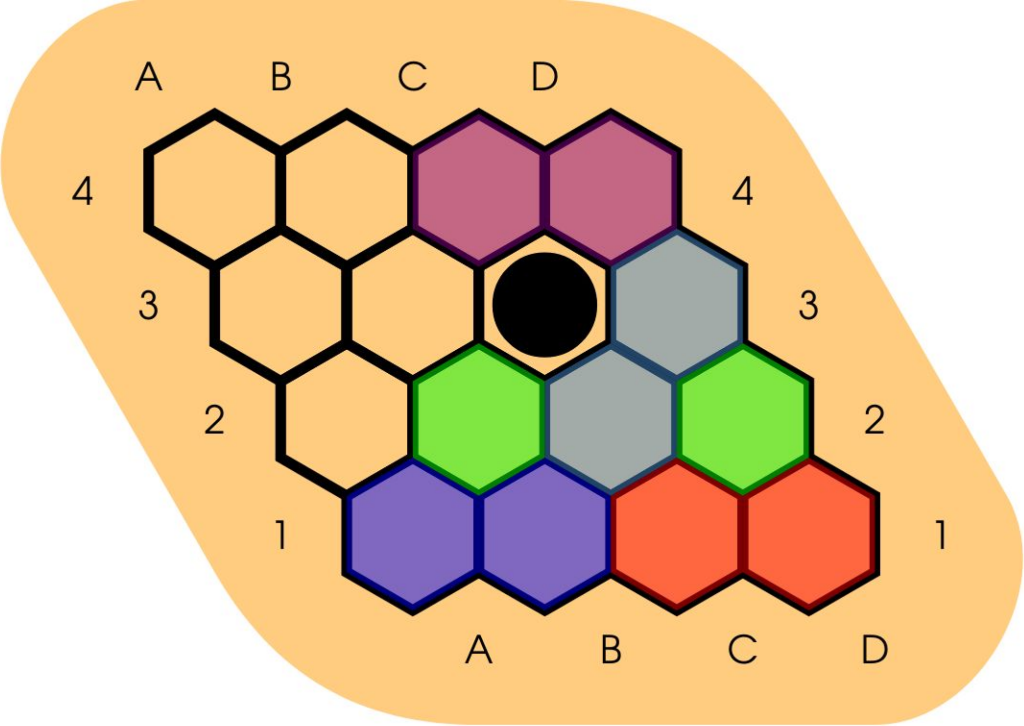
- Comparing against:
  - An enhanced alpha beta solver we developed
  - Morat Monte Carlo Tree Search
  - Morat Proof Number Search



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	-

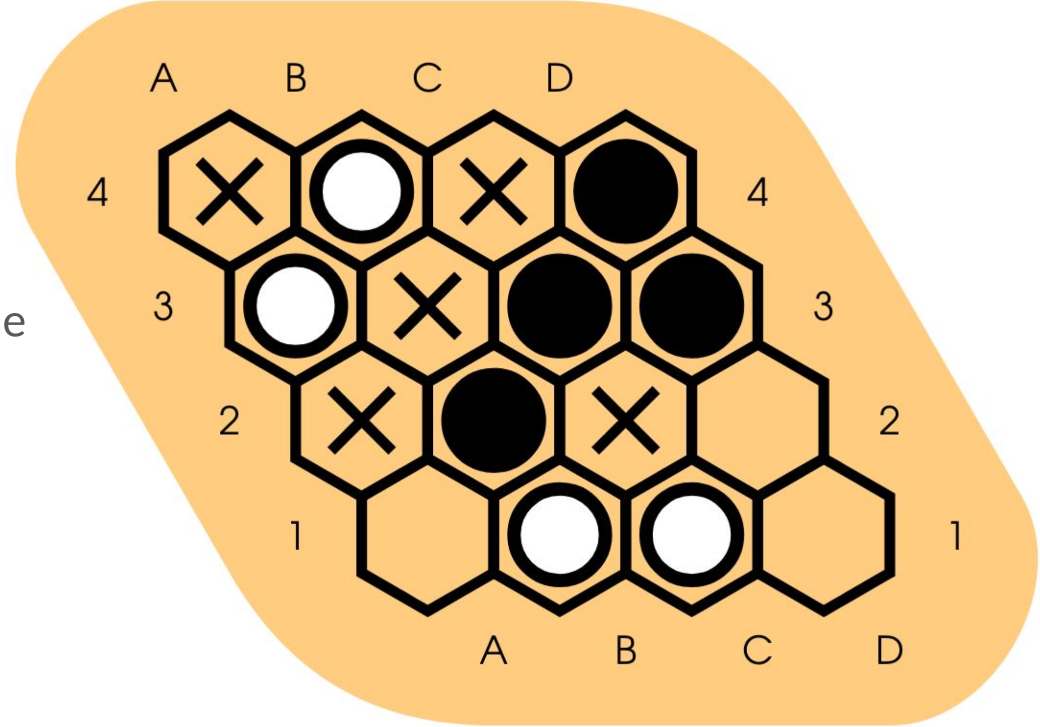
Solving with Hex Knowledge

- Virtual Connections
- Strategies to guarantee connections
- Full board connections determine the winner
- Intersections of first player VCs form a must-play zone



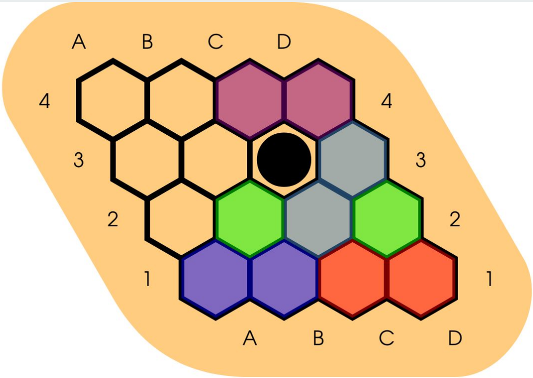
Solving with Hex Knowledge

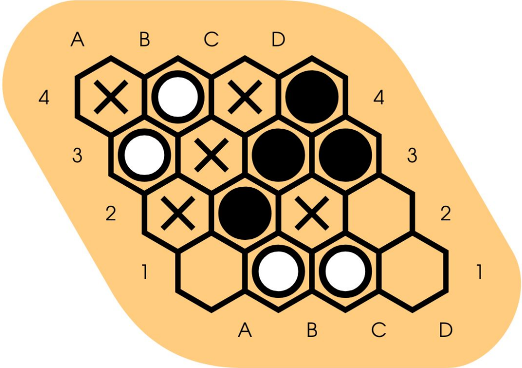
- Dead cells / Inferior cell analysis
- Some moves are irrelevant to the game's theoretical outcome
- Dead cells can be filled in, inferior moves can be ignored
- Found with pattern matching



Solving with Hex Knowledge

- Significantly reduces search space
- For 6x6 Hex: 93,963,192 nodes -> 26 nodes
- 8x8 can be solved
- Search is much slower, but more node efficient





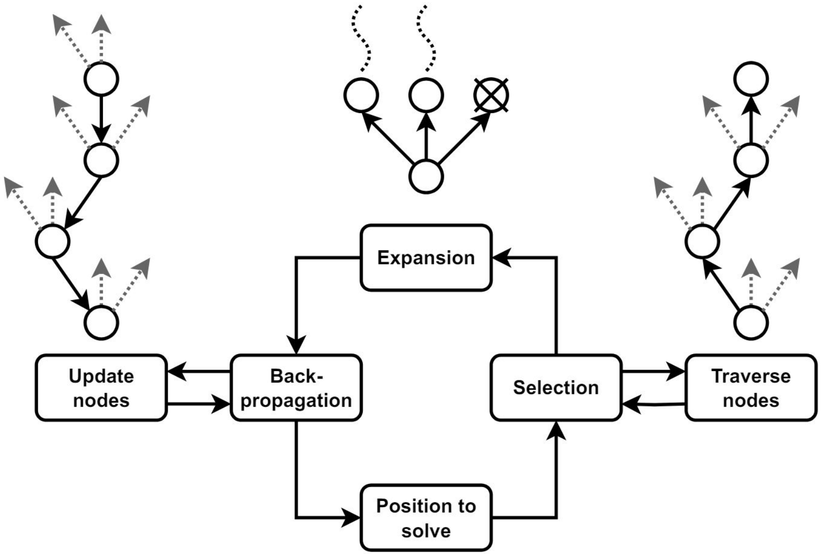
Program	6×6 Time (s)	6×6 Nodes	7×7 Time (s)	7×7 Nodes	8×8 Time (s)	8×8 Nodes
EWS with knowledge	0.005	26	0.588	2,318	234.449	555,158

Conclusion

- Expected Work Search combines proof size and win rate heuristics
  - Balances these heuristics
  - Covers previous weaknesses

- Strong results on Go and Hex
  - New results solving 5x5 Go with pos. superko
  - More Hex knowledge is needed for S.O.T.A results

- Plenty of future work for EWS



Thank you for your time!

Questions?

References

[1] Allis, L., Herik, H., and Huntjens, M. Go-Moku and Threat-Space Search. *Computational Intelligence 12* (1994), pp. 7–23.

[2] Allis, L., van der Meulen, M., and van den Herik, H. Proof-Number Search. *Artificial Intelligence 66*, 1 (1994), pp. 91–124.

[3] Anshelevich, V. V. The Game of Hex: An Automatic Theorem Proving Approach to Game Programming. In *AAAI/IAAI 17* (2000), pp. 189–294.

[4] Anshelevich, V. V. A Hierarchical Approach to Computer Hex. *Artificial Intelligence 134*, 1 (2002), pp. 101–120.

[5] Arneson, B., Hayward, R., and Henderson, P. Solving Hex: Beyond Humans. In *Computers and Games* (2010), pp. 1–10.

[6] Benson, D. Life in the Game of Go. *Information Sciences 10* (1976), pp. 17–29.

[7] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games 4*, 1 (2012), pp. 1–43.

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

[9] Doe, E., Winands, M. H. M., Kowalski, J., Soemers, D. J. N. J., Górski, D., and Browne, C. Proof Number Based Monte-Carlo Tree Search. In *IEEE Conference on Games* (2022), pp. 206–212.

[10] Du, H., Wei, T. H., and Müller, M. Solving NoGo on Small Rectangular Boards. In *Advances in Computer Games* (2024), Springer Nature Switzerland, pp. 39–49.

[11] Enzenberger, M., Müller, M., Arneson, B., and Segal, R. 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* (2011), pp. 259–270.

[12] Ewalds, T. V. Playing and Solving Havannah, 2012. *University of Alberta Master's Thesis* (2012), https://github.com/tewalds/morat.

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

[14] Henderson, P., Arneson, B., and Hayward, R. Solving 8x8 Hex. *IJCAI International Joint Conference on Artificial Intelligence* (2009), pp. 505–510.

[15] Kishimoto, A., and Müller, M. A General Solution to the Graph History Interaction Problem. In *Proceedings of the 19th National Conference on Artifical Intelligence* (2004), pp. 644–649.

[16] Kishimoto, A., Winands, M., Müller, M., and Saito, J. T. Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal 35* (2012), pp. 131–156.

[17] Knuth, D. E., and Moore, R. W. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence 6*, 4 (1975), pp. 293–326.

[18] Kocsis, L., and Szepesvári, C. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML* (2006), pp. 282–293.

[19] Müller, M. Playing it Safe: Recognizing Secure Territories in Computer Go by Using Static Rules and Search. In *Game Programming Workshop* (1997), pp. 80–86.

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

[21] Neto, J. P., and Taylor, W. Game Mutators for Restricting Play. *Game & Puzzle Design* (2015), pp. 64–65.

[22] Niu, X., Kishimoto, A., and Müller, M. Recognizing Seki in Computer Go. In *Advances in Computer Games. 11th International Conference* (2006), vol. 4250 of *Lecture Notes in Computer Science*, Springer, pp. 88–103.

[23] Niu, X., and Müller, M. An Improved Safety Solver for Computer Go. In *Computers and Games: 4th International Conference* (2006), vol. 3846 of *Lecture Notes in Computer Science*, Springer, pp. 97–112.

[24] Niu, X., and Müller, M. An Open Boundary Safety-of-Territory Solver for the game of Go. In *Computers and Games. 5th International Conference* (2007), vol. 4630 of *Lecture Notes in Computer Science*, Springer, pp. 37–49.

[25] Niu, X., and Müller, M. An Improved Safety Solver in Go using Partial Regions. In *Computers and Games. 6th International Conference* (2008), vol. 5131 of *Lecture Notes in Computer Science*, Springer, pp. 102–112.

[26] Pawlewicz, J., and Hayward, R. B. Scalable Parallel DFPN Search. In *Computers and Games* (2013), pp. 138–150.

[27] Pawlewicz, J., and Lew, L. Improving Depth-First PN-Search: 1 + Epsilon Trick. In *Proceedings of the 5th International Conference on Computers and Games* (2006), pp. 160–171.

[28] Randall, O., Wei, T.-H., Hayward, R., and Müller, M. Improving Search in Go Using Bounded Static Safety. In *Computers and Games* (2022), pp. 14–23.

[29] Rosin, C. Multi-armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence 61* (2010), pp. 203–230.

[30] Saffidine, A., and Cazenave, T. Developments on Product Propagation. In *Computers and Games* (2013), pp. 100–109.

[31] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. Checkers is Solved. *Science 317*, 5844 (2007), pp. 1518–1522.

[32] Schaeffer, J., and Plaat, A. New Advances in Alpha-Beta Searching. In *Proceedings of the 1996 ACM 24th annual Conference on Computer Science* (1996), pp. 124–130.

[33] Shih, C.-C., Wu, T.-R., Wei, T. H., and Wu, I.-C. A Novel Approach to Solving Goal-Achieving Problems for Board Games. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence* (2021), pp. 10362–10369.

[34] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., 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., and Hassabis, D. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature 529* (2016), pp. 484–489.

[35] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science 362*, 6419 (2018), pp. 1140–1144.

[36] Slate, D., and Atkin, L. Chess Skill in Man and Machine. In *Springer-Verlag* (1977), pp. 82–118.

[37] Song, J., and Müller, M. An Enhanced Solver for the Game of Amazons. *IEEE Transactions on Computational Intelligence and AI in Games 7* (2015), pp. 16–27.

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

[39] van der Werf, E., Herik, H., and Uiterwijk, J. Solving Go on Small Boards. *ICGA Journal 26* (2003), pp. 92–107.

[40] van der Werf, E., and Winands, M. Solving Go for Rectangular Boards. *ICGA Journal 32*, 2 (2009), pp. 77–88.

[41] Wagner, J., and Virág, I. Solving Renju. *Joint International Computer Games Association 24* (2001), pp. 30–35.

[42] Winands, M. H. M., Björnsson, Y., and Saito, J.-T. Monte-Carlo Tree Search Solver. In *Computers and Games* (2008), pp. 25–36.

[43] Wu, T.-R., Shih, C.-C., Wei, T.-H., Tsai, M.-Y., Hsu, W.-Y., and Wu, I.-C. AlphaZero-based Proof Cost Network to Aid Game Solving. In *International Conference on Learning Representations* (2022).

[44] Zobrist, A. A New Hashing Method with Application for Game Playing. *ICGA Journal 13* (1990), pp. 69–73.