Improving Search in Go Using Bounded Static Safety

Owen Randall[1,2(✉)], Ting-Han Wei[1,2], Ryan Hayward[1], and Martin Müller[1,2]

[1] Computing Science, University of Alberta, Edmonton, AB, Canada
{davidowe,tinghan,hayward,mmueller}@ualberta.ca
[2] Amii, University of Alberta, Edmonton, AB, Canada

Abstract. Finding the winner of a game of Go is difficult even on small boards due to the game's complexity. Static safety detection algorithms can find a winner long before the end of a game, thereby reducing the number of board positions that must be searched. These static algorithms find safe points for each player and so help prove the outcome for positions that would otherwise require a deep search. Our board evaluation algorithm Bounded Static Safety (BSS) introduces two new methods for finding valid lower bounds on the number of safe points: *extending liberties* and *intersecting play*. These methods define statically evaluated greedy playing strategies to raise the lower bound of a player's guaranteed score. BSS can solve positions from a test set of 6×6 games at an average of 27.29 plies, a significant improvement over the previous best static safety method of locally alternating play (31.56 plies), and far surpassing Benson's unconditional safety (42.67 plies).

Keywords: Go game · Safety under alternating play · Game solving · Game tree search · Static evaluation

1 Introduction

Go is a classic test bed for computing science research. Much previous work on board evaluation has focused on heuristic estimates. For example, AlphaGo [13] and its successors [12,14,15] train a value network, a deep neural network that predicts the winning probability in an arbitrary Go game state. While these heuristics led to superhuman strength Go programs, they do not find the game-theoretic value of a Go state: exact board evaluation methods are required for this task [11,16,17]. Exact game solvers can also be used to find mistakes in the play of strong game playing programs [4], and exact board evaluation in Go has been used to provide the ground truth for training neural networks which predict point ownership: this auxiliary task improved the accuracy of the open source program KataGo [18,20]. Therefore there is practical value in pursuing and improving exact board evaluation methods. We focus on static analysis of safe points: we give a strategy that guarantees a minimum number of points for

C. Browne et al. (Eds.): CG 2022, LNCS 13865, pp. 14–23, 2023.
https://doi.org/10.1007/978-3-031-34017-8_2

a player. If this number is above a threshold which depends on board size and *komi*, then the player is guaranteed a win.

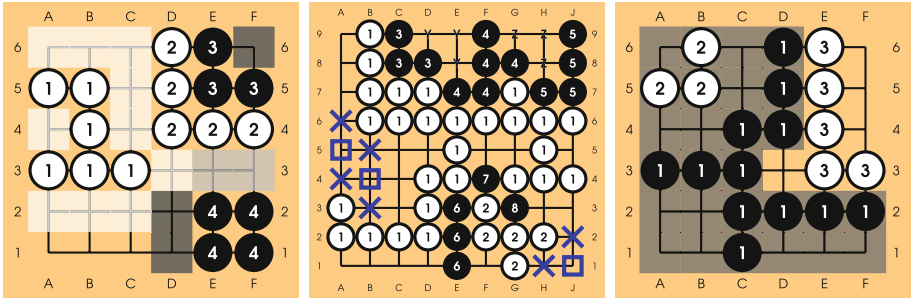
Bounded Static Safety (BSS) improves upon previous algorithms for static safety [2,6]: after first using such algorithms to find a core set of safe points, it uses new greedy strategies to find higher guaranteed scores for each player. Our experiments show that BSS typically solves game positions much earlier than previously possible. This improves the efficiency of search-based solvers for Go, enabling more difficult positions to be solved in the same amount of time.

2 Related Work

Benson's static algorithm [2] finds *blocks* of stones that are *unconditionally safe*: they cannot be captured, even when the defender always passes. *Safety by local alternating play* (LAP) finds a larger set of of safe stones and surrounded regions by allowing simple local defender responses, based on concepts such as *miai strategies* and *chains* [6]. We use the Benson and LAP implementations in the open-source program Fuego [3] as baselines for evaluating BSS. Work by Niu et al. applies static safety analysis based on LAP within a specialized local search framework [8], with extensions to finding safety in open regions [9,10], and resolving seki [7]. Those algorithms are designed for large-scale searches on 19×19 boards, while our BSS algorithm focuses on fast static analysis which can be used within a small board solver.

3 Background

3.1 Rules of Go



(a) (b) (c)

Fig. 1. Example Go positions. (a) With block numbers and shaded liberties (b) Sure liberty regions (c) Safe regions.

This section provides a very brief summary of the rules of Go; see [1] for full details. In Go, two players take turns placing a stone of their color on a point on the board. A *block* is a group of adjacently connected stones of the same color. A *liberty* is an empty point adjacent to a block. Figure 1a shows an example position with labeled blocks and their liberties. A play on the last liberty of a block captures it, and it is removed from the board. A no-suicide rule forbids players from killing their own blocks.

3.2 Sure Liberty Regions

Let $A_{\neg c}$ be all of the points which are not of color c, then a basic *region* of color c is a maximally connected subset of $A_{\neg c}$; see [8] for full details. Regions are enclosed by blocks of their color, e.g. in Fig. 1b, the points marked Y make up a region enclosed by the black blocks marked 3 and 4. If a defender block B is an enclosing block for region R, and there exists a strategy such that B always has a liberty in R when it is the attacker's turn, then R is a *sure liberty region* for B. This concept is closely related to, but more narrowly defined than the popular notion of an *eye*. For example, in Fig. 1a, A4 is a trivial sure liberty region for block 1. A block with at least two sure liberties is safe under local alternating play (LAP).

Miai strategies help to identify sure liberty regions. An *interior point* of a region is a point which is not adjacent to an enclosing block. A region is a sure liberty region if every empty interior point is adjacent to at least two unique adjacent defender liberties. A miai strategy on such liberty pairs ensures that there is always a liberty available to the enclosing block, even if it is the attacker's turn, as the attacker must leave at least one empty point in the region due to the no-suicide rule. For example in Fig. 1b, the region containing point A9 enclosed by block 1 is a sure liberty region for block 1 using the miai strategy. The points in the region marked by squares are interior points and each require a pair of adjacent liberties which are marked by crosses. Separate blocks which enclose the same sure liberty regions can be merged into a chain, and if a set of chains has at least two sure liberty regions each then it is safe under alternating play [6]. In Fig. 1b the blocks marked 3, 4, and 5 can be merged into a single safe chain using the regions marked Y and Z, which also provide two sure liberties.

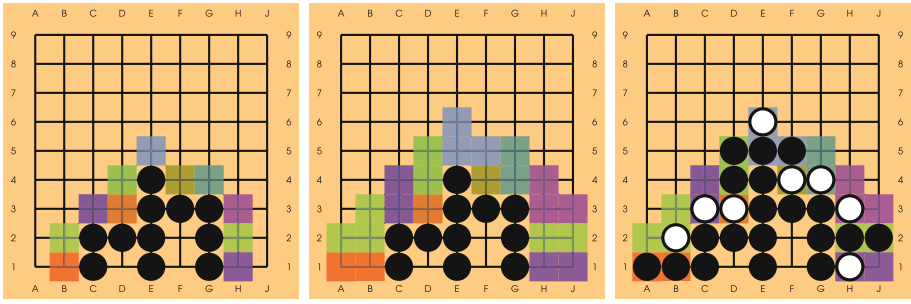
3.3 Proving the Safety of Regions

The points of a region are proven to be safe if the region is enclosed by safe defender blocks, and there exists a defender strategy which prevents the attacker from creating any safe attacker blocks inside. In Fig. 1c, safe blocks and regions for Black are shaded black. The region containing the point A6 is safe for Black, as its enclosing block is proven safe, and White cannot create any safe blocks in the region. Assuming it is White to play, White can play at point F5 to make the block marked 3 safe, making this an unsafe region for Black. A region which contains two non-adjacent points which could become attacker eye points is considered unsafe.

4 Improving Lower Bounds for the Number of Safe Points

We propose two novel static techniques, *extending liberties* and *intersecting play*. In LAP safety, lower bounds on player scores are calculated by simply counting the number of safe points on the board. We develop defender strategies which greedily occupy nearly empty points, in order to increase the defender lower bound. To compute the bound, the attacker plays an optimal response to each greedy strategy. Evaluating the score resulting from these strategies *does not require search*. Defender scores are initialized by using LAP safety analysis, before additional safe points are counted using the new techniques. The resulting lower bound is at least as good as the underlying baseline method.

4.1 Technique 1: Extending Liberties



(a) EL points (b) Extension trees (c) Possible result of play

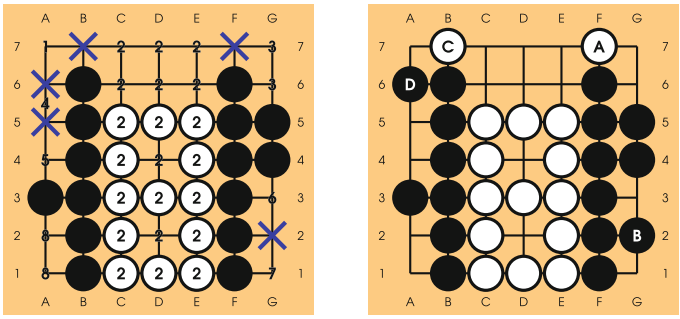
Fig. 2. Example position with extension trees colored. Empty points of the same color belong to the same extension tree. (Color figure online)

The extending liberties strategy has the defender greedily play on *Extending Liberties (EL) points* which are empty points in unsafe regions that are adjacent to safe defender blocks. Any defender stones placed adjacent to a safe defender block will also become safe, improving the lower bound of their score. Figure 2a shows every EL point for Black as a different shaded color. Playing on an EL point could create new EL points, e.g. in Fig. 2a if Black plays C3, points B3 and C4 become EL points. We define an *extension tree* as a set of an EL point and any empty adjacent points. Extension trees must be non-overlapping to avoid over-counting. Figure 2b shows a valid extension tree layout where the points in each extension tree are shaded the same color. There can be multiple valid extension tree layouts. In practice a valid layout is chosen arbitrarily.

The value of an EL point is equivalent to the size of the extension tree it belongs to. In Fig. 2b EL point B2 is more valuable than EL point B1, because its

extension tree has more points. If the defender plays on an EL point, any points in the same extension tree become EL points themselves, however their respective extension trees do not grow. If the attacker plays on an EL point, any points in the same extension tree are removed from play. The defender plays greedily with respect to the value of EL points. Thus the optimal attacker strategy is to also play greedily with respect to the value of EL points. Figure 2c shows a possible line of play if Black is following the extending liberties strategy and White is responding optimally. The lower bound on the defender's score is increased by the number of safe points they would gain by playing this strategy. For the example position in Fig. 2 the lower bound for Black's score would increase from 14 to 22 using the extending liberties technique.

4.2 Technique 2: Intersecting Play



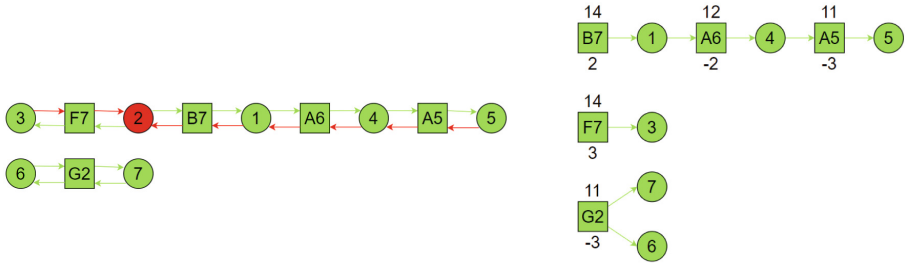
(a) Position (b) Result of play

Fig. 3. An example 7×7 position.

Intersection points are empty points such that placing a stone on that point subdivides the region it was placed in. In an *intersecting play strategy*, the defender only plays on their intersection points which create new safe points. The optimal attacker responses are therefore also on the same points to prevent defender safety. Similar to the extending liberties strategy, we define a value for each intersection point, and the defender plays greedily with respect to these values.

We introduce the following terminology to describe this technique:

- *Intersection node*: holds information corresponding to an intersection point; connected to region nodes.
- *Region node*: holds information corresponding to a region that is enclosed by defender blocks and intersection points; connected to intersection nodes.



(a) IRGs of Fig. 3a (b) Corresponding value trees

Fig. 4. Example position for intersecting play. (Color figure online)

- *Intersection-region graph* (IRG): a bipartite graph containing intersection and region nodes. Holds relevant information on the positioning of intersection points and regions.
- *Value tree* (VT): a tree derived from an IRG, which only contains nodes with the potential to be safe.

Figure 3a shows Black's intersection points marked as crosses and the regions associated with region nodes marked as numbers. Note that region nodes are distinct from basic regions as they can be enclosed by intersection points. Region nodes of size zero can exist between adjacent intersection points, e.g. region node 4. Figure 4a shows the IRGs corresponding to the example position. Green nodes are potentially safe, red nodes cannot be safe. E.g. region 2 already contains safe attacker points, so the defender cannot be safe there. Connections are unsafe if they can belong to a trail which includes an unsafe node, in Fig. 4a unsafe connections are colored red.

VTs are constructed from the safe trees which exist in each IRG. The root of each VT is the intersection node adjacent to an unsafe region in its corresponding IRG. If an IRG has no unsafe nodes an arbitrary intersection node is chosen for the root of a VT. Figure 4b shows the value trees derived from the IRGs in Fig. 4a. The numbers above each intersection node are their absolute values, and the numbers below are their relative values. Absolute values represent the number of safe points the defender will gain if they play at this point. In Fig. 3a if Black plays A6, the adjacent block becomes safe, earning Black 12 safe points, therefore the intersection point's absolute value is 12.

A defender move which creates a safe block will affect the value of every intersection point adjacent to that block. In Fig. 3a if the defender plays G2, the adjacent block will become safe which will change the absolute value of intersection point F7 from 14 to 3. The absolute values of intersection points are dependent if they are adjacent to the same unsafe defender block, or are in the same VT. The relative value of an intersection point is the increase in the number of safe points the defender will gain by playing on this point compared to playing on any other dependent intersection point.

Formally, let $A(x)$ be the absolute value of intersection point x, let $B(x)$ be the block adjacent to x, let $VT(x)$ be the VT of x, let $U(b)$ be whether block b is unsafe, and let I be the set of all intersection points. Then the relative value of intersection point x is:

$$R(x) := A(x) - \mathbf{max}\{A(y) \forall y \in I \setminus x | (B(y) = B(x) \wedge U(B(X))) \vee (VT(x) = VT(y))\}$$

In Fig. 4b the relative value of the intersection point on B7 is 2, because its absolute value is 14 and the maximum absolute value of its dependent intersections points is 12.

The defender plays greedily with respect to the relative values of intersection points. The attacker can respond optimally by threatening the safety of defender blocks if possible, or by playing greedily with respect to relative values otherwise. If an unsafe defender block only has two adjacent intersection points, the attacker can threaten the safety of the block by playing on one of these points. In Fig. 3a intersection point B7 initially has a larger relative value than point G2. However, when White plays F7 the safety of the block is threatened, forcing Black to play G2. This allows White to play B7 on their next turn. Figure 3b shows the result of play when following the intersecting play strategy.

5 Results

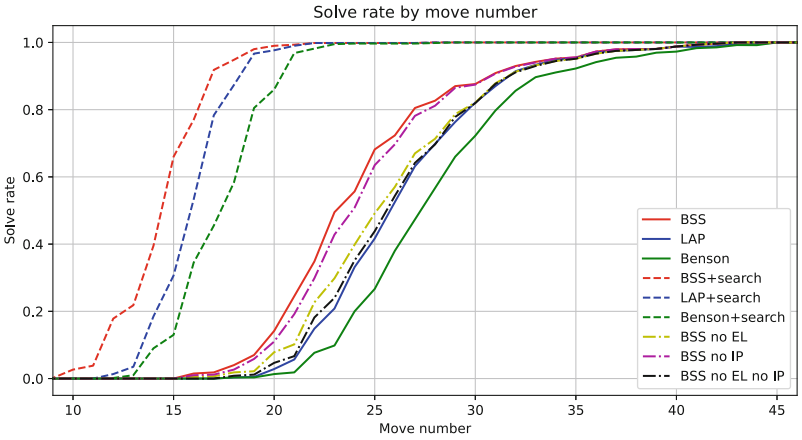


Fig. 5. Solve rate of the positions in 600 6×6 games by the move number of each position for each safety evaluation type. (Color figure online)

We evaluate BSS on a test set of 600 6×6 games. We chose 6×6 as it is the smallest unsolved square board size. The games in the test set last between 29 and 124 moves, with a median of 40, for a total of 26,885 positions. We generate

Table 1. The average number of moves each safety evaluation type required to be played before they were able to solve the position.

Type	BSS	LAP	Benson	BSS +search	LAP +search	Benson +search	BSS no EL	BSS no IP	BSS no EL no IP
Av. first solve	**27.29**	31.56	42.67	**14.89**	16.35	17.79	30.83	27.73	31.28

these games by sampling lines of play which are expected to be similar to states that must be covered by a proof tree. One player is a strong heuristics-guided agent, while the opponent plays randomly. The heuristic player represents strong move choices in the OR nodes of a proof tree (moves by the winning side), while the random player creates a random sample from all the legal moves that must be analyzed in an AND node (the losing side). The heuristic player uses one of six Proof Cost Networks (PCN) [19], which are trained similarly to AlphaZero [14], but with the altered training target of minimizing proof size instead of maximizing the win rate. We use 6 PCNs with different training parameters and generate 100 games each to create a dataset with diverse lines of play.

We compare BSS against the static safety implementations for LAP safety [6] and Benson safety [2] in the open-source program Fuego [3]. A position is statically solved when the safety evaluation finds enough safe points for a win. For a 6×6 board with a komi of 3.5, Black must have at least 20 points to win with a score of 20:16 or better, and White must have at least 17 points to win by 19:17. This disregards rare cases of seki where some points remain neutral at the end. A position is solved by search when a proof tree is constructed showing that one player can always win. We evaluate the strength of a safety evaluation function by the earliest point in a game when it is first statically solved - earlier generally indicates a stronger safety evaluation.

Figure 5 shows the *solve rate* - the percentage of positions solved - for each move number, with solid lines for BSS (red), LAP (blue) and Benson (green). The average number of moves required to first solve the games in the test set are shown in Table 1. BSS succeeds more than 4 moves earlier than LAP on average, and more than 15 moves earlier than Benson.

The next experiment uses static safety evaluation within a 30 s (wall-clock time) iterative-deepening alpha-beta search with the distance-to-eye heuristic move ordering [5] and hand-tuned positional evaluation function [17]. Static safety is called in two places: 1) to prune moves inside opponent safe regions, and 2) to determine whether the position is statically solved.

The results are shown using the dashed lines of Fig. 5. BSS improves search efficiency over LAP and Benson. The gap between methods is considerably smaller when using search since many early board states are still beyond the reach of all these static methods.

The remaining dashed-dotted lines in Fig. 5 summarize an ablation study of the two techniques extending liberties and intersecting play. Without those techniques, our static safety implementation has performance similar to LAP, as expected. Intersecting play and extending liberties each contribute to the

performance of BSS, with extending liberties being more important. We conjecture that the number of intersection points present is often low, and intersection points are often strong moves, so the strong player may often play these points early.

6 Concluding Remarks

BSS is the new state of the art algorithm for exact static safety evaluation. By using the lower bounds found by the extending liberties and intersecting play techniques, we can statically solve Go positions early than previously possible in many cases. Our experiments show that games can be solved by BSS earlier than previous works, that BSS can improve the efficiency of searches, and that our extending liberty technique is the most important for solving early positions. We believe that this work allows for improvement in various Go problems, such as finding game theoretical values, creating ground truths for auxiliary neural network training tasks, and for evaluating Go playing programs.

Acknowledgements. The authors acknowledge financial support from NSERC, the Natural Sciences and Engineering Research Council of Canada, Alberta Innovates, DeepMind, and the Canada CIFAR AI Chair program.

References

1. Sensei's Library. https://senseis.xmp.net/
2. Benson, D.: Life in the Game of Go. Information Sciences 10, 17–29 (1976). Reprinted in Computer Games, Levy, D.N.L. (ed.), vol. II, pp. 203–213, Springer, New York (1988)
3. Enzenberger, M., Müller, M., Arneson, B., Segal, R.: Fuego-an open-source framework for board games and go engine based on Monte Carlo tree search. IEEE Trans. Comput. Intell. AI Games **2**(4), 259–270 (2010)
4. Haque, R., Wei, T.H., Müller, M.: On the road to perfection? Evaluating Leela chess zero against endgame tablebases. In: ACG 2021. LNCS, vol. 13262, pp. 142–152. Springer, Cham (2021). https://doi.org/10.1007/978-3-031-11488-5_13
5. Kishimoto, A.: Correct and efficient search algorithms in the presence of repetitions. Ph.D. thesis, University of Alberta (2005)
6. Müller, M.: Playing it safe: recognizing secure territories in computer go by using static rules and search. In: Game Programming Workshop in Japan '97, pp. 80–86. Computer Shogi Association, Tokyo, Japan (1997)
7. Niu, X., Kishimoto, A., Müller, M.: Recognizing Seki in computer go. In: van den Herik, H.J., Hsu, S.-C., Hsu, T., Donkers, H.H.L.M.J. (eds.) ACG 2005. LNCS, vol. 4250, pp. 88–103. Springer, Heidelberg (2006). https://doi.org/10.1007/11922155_7
8. Niu, X., Müller, M.: An improved safety solver for computer go. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 97–112. Springer, Heidelberg (2006). https://doi.org/10.1007/11674399_7
9. Niu, X., Müller, M.: An open boundary safety-of-territory solver for the game of go. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 37–49. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75538-8_4

10. Niu, X., Müller, M.: An improved safety solver in go using partial regions. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 102–112. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87608-3_10
11. Schaeffer, J., et al.: Checkers is Solved. Science **317**(5844), 1518–1522 (2007)
12. Schrittwieser, J., et al.: Mastering atari, go, chess and shogi by planning with a learned model. CoRR abs/1911.08265 (2019). http://arxiv.org/abs/1911.08265
13. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (2016)
14. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. CoRR abs/1712.01815 (2017). http://arxiv.org/abs/1712.01815
15. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**, 354–359 (2017). https://doi.org/10.1038/nature24270
16. van der Werf, E., Winands, M.: Solving go for rectangular boards. ICGA J. **32**(2), 77–88 (2009). https://doi.org/10.3233/ICG-2009-32203
17. van der Werf, E., Herik, H., Uiterwijk, J.: Solving go on small boards. ICGA J. **26** (2003). https://doi.org/10.3233/ICG-2003-26205
18. Wu, D.J.: Accelerating self-play learning in go. CoRR abs/1902.10565 (2019). http://arxiv.org/abs/1902.10565
19. Wu, T.R., Shih, C.C., Wei, T.H., Tsai, M.Y., Hsu, W.Y., Wu, I.C.: AlphaZero-based proof cost network to aid game solving. In: International Conference on Learning Representations (2022). https://openreview.net/forum?id=nKWjE4QF1hB
20. Wu, T.R., et al.: Multi-labelled value networks for computer go. CoRR abs/1705.10701 (2017). http://arxiv.org/abs/1705.10701