Exploring Positional Linear Go

Noah Weninger and Ryan Hayward(✉)

Department of Computing Science, University of Alberta, Edmonton, Canada
hayward@ualberta.ca

Abstract. Linear Go is the game of Go played on the $1 \times n$ board. Positional Linear Go is Linear Go with a rule set that uses positional superko. We explore game-theoretic properties of Positional Linear Go, and incorporate them into a solver based on MTD(f) search, solving states on boards up to 1×9.

1 Introduction

Now that computers have surpassed humans in playing the game of Go [1,2]—also known as Baduk or Weiqi—perhaps it is time to revisit the problem of *solving* the game of Go. Solving—i.e., finding exact minimax values—even for 9×9 Go is currently intractable, but progress has been made for smaller boards. In 2002 the program Migos by Erik van der Werf et al. solved all opening 5×5 Go positions [3–5]. Later, Migos solved positions for various rectangular boards with at most 30 cells [6,7].

Linear Go is Go on the $1 \times n$ board. *Positional Linear Go* (PLGo, or n-PLGo when we specify the board size) is Linear Go with a rule set that uses positional superko, e.g., Tromp-Taylor rules [8] with no suicide and no komi.[1] So a non-pass move cannot recreate any position that occurred earlier, and the game ends after two consecutive passes. A *state* is defined by the player to move, current position, the set of earlier positions, and whether the previous move was pass.

As far as we are aware, PLGo has not been solved before. Van der Werf solved Go with a rule set in which—except for immediate ko, which is forbidden—under certain conditions the return to an earlier position results in no result, or draw. In this version of Go, consider a state S with graph G of all continuations from S. In G, consider a move sequence $(m_1, \ldots, m_{t-1}, m_t)$ leading from a state with position X to a subsequent state with same position. When traversing G to find a strategy that bounds the minimax score for player P, one must prune move m_t—which results in a draw—but also move m_{t-1} whenever this move is made by P, for otherwise P's opponent can then draw. By contrast, in PLGo only move m_t is pruned. So the search space for the above version of Go is smaller than for PLGo.

All empty-board PLGo scores found so far—up to $n = 9$—agree with those of the version of Linear Go solved by Migos. See Fig. 1.

[1] We wanted a rule set that is concise, precise, and—like Asian Go—has no suicide.

M. H. M. Winands et al. (Eds.): ACG 2017, LNCS 10664, pp. 101–114, 2017.
https://doi.org/10.1007/978-3-319-71649-7_9

Version of Linear Go solved by MIGOS: minimax score													
n	1	2	3	4	5	6	7	8	9	10	11	12
score	0	0	3	4	0	1	2	3	0	1	2	2

Fig. 1. Empty board 1st-player minimax score as solved by MIGOS [7].

2 Observations on Solving Positional Linear Go

In this paper, we describe board positions like this `(-x-o--)`, where x and o are black and white stones, respectively. We describe move sequences like this (1.b2 2.w4 ...) or this (b2 w4 ...) or this (**2** 4 ...). Also, when no captures have occurred, we describe move sequences like this `(-1-2--)` or as in Fig. 2.

To illustrate PLGo, consider the left state of Fig. 2. Black moved first at cell 2; White replied at cell 4. Now Black has 5 possible moves: pass—always legal—or at any of the 4 empty cells. Assume Black plays at 5. Now White has 2 possible moves: pass or at 6, since playing at 1 or 3 would be suicide. Assume White plays at 6, capturing the Black stone at 5. Now Black has 3 possible moves: pass, 1 or 3—playing at 5 would capture the stone at 6 and recreate an earlier position, violating positional superko.

We leave it as an exercise for the reader to solve the states in Fig. 2.

Fig. 2. For each state, find the principal variation and minimax score. Solutions in Figs. 6 and 14.

In principle, solving Go—or any other 2-player complete information game—is straightforward: perform minimax search on the game tree. If the game allows transpositions, i.e., different play sequences that yield the same state, it can be faster to search on the game graph rather than the game tree.

In practice, solving Go in this way becomes intractable even for small boards, due to the size of the search tree. Figure 3 shows the 2-PLGo Go tree. For n-PLGo, the game tree is only slightly larger than the game graph, which—discounting positions unreachable from the root position—is significantly larger than the number of legal positions—about $.85 \times .97^{1+n} \times 2.98^n$ [9]—since the same position can have different histories and so appear in several nodes. See Figs. 3 and 4.

To prove a minimax value, one needs both a lower bound and an upper bound. Trees that establish such bounds are called proof trees. Figure 4 shows a proof tree for the lower bound of the 5-PLGo first-player empty board minimax value.

For $n = t \times t$, proof trees for n-PLGo tend to be larger than proof trees for $t \times t$ Go: boards with at least two rows and two columns often have unconditionally safe groups, and detecting such groups reduces the search space [10]. But in PLGo, determining whether a group is safe usually requires knowing the history as well as the position.

Solving a PLGo state is similar in flavor to the NP-hard problem of finding a longest path in the associated transition graph, so pruning is likely to significantly reduce solving time. In this section we consider PLGo properties that allow pruning.

The following gives a lower bound on the empty-board score.

n	legal n-PLGo positions
1	1
2	5
3	15
4	41
5	113
6	313
7	867
8	2401
9	6649
10	18413
11	50991
12	141209
13	391049

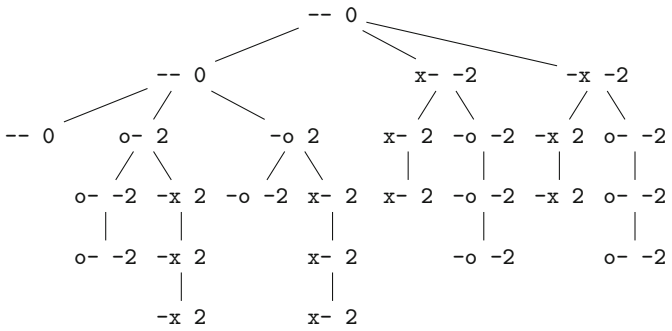

Fig. 3. Left: the number of legal $1 \times n$ Go positions [9]. A position can occur in multiple states, so the number of legal $1 \times n$ Go states is more than the number of legal $1 \times n$ Go positions. Right: the 2-PLGo state transition graph for all states reachable from the empty board. Each node score is 1st-player minimax.

Theorem 1. *For $m \times n$ Go with positional superko, the first-player empty-board minimax score is non-negative.*

Proof. Argue by contradiction: assume the first-player minimax score t is negative.

First find the minimax value when the first move is pass. Case 1: the opponent passes, the game ends, score 0. Case 2: the opponent does not pass. Now use strategy stealing: after exchanging the names of the players, the game transition graph is identical to the original game transition graph. The opponent's minimax score is $t < 0$, so the opponent prefers case 1, pass, minimax score 0.

Thus, from the original position, the first player has a move (pass) with minimax score 0, which is greater than the minimax score t, which is a contradiction.

The *history* of a PLGo state is the set $\mathcal{P} = \{P_0, \ldots, P_t\}$ of board positions P_j, where P_0 and P_t are the original and current position respectively. Define a

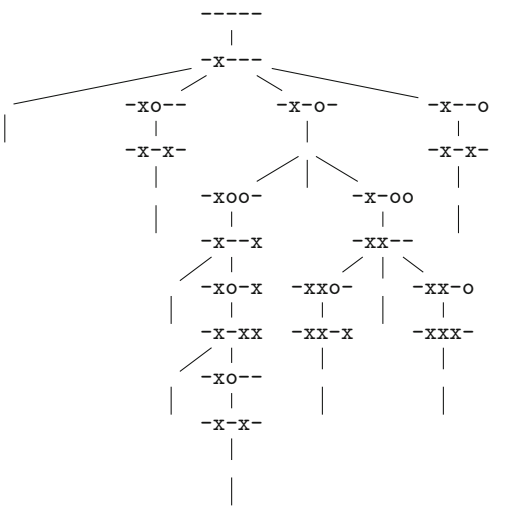

Fig. 4. Proof tree for lower bound for 5-PLGo: 1st player scores at least 0. Each empty node is pass.

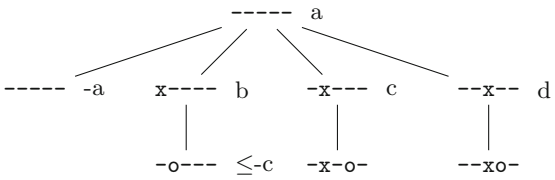

Fig. 5. Two PLGo empty-board minimax (in)equalities. Node labels are minimax value. Empty-board value is a, so value of (1.pass) is $-a$. Value of (1.b2) is c, so value of (1.b1 2.w2) is at most $-c$.

position as a sequence of values—black, white, empty—indexed by the cells. The *position* of a state is its current position. Define a PLGo *state* $S = (\mathcal{P}, x, p)$ by its history $\mathcal{P}$, the player to move x, and a Boolean p that is true whenever the previous move was pass. Label cells of the board from 1 to n so that consecutively labeled cells are neighbors, e.g., as in Fig. 2. For a state $S = (\mathcal{P}, z, p)$ with position P, for player x (either z or the opponent of z), define $\mu_x(P)$ as the score of P and $\mu_x(S)$ as the minimax value of S.

Conjecture 1. For n-PLGo with $n \leq 7$, let $S = (\mathcal{P}, x, p)$ be a state such that neither player has played at cell 1 nor 2. For each j in $\{1, 2\}$, let S_j be the state resulting from x playing at cell j. Then $\mu_x(S_1) \leq \mu_x(S_2)$.

We have verified Conjecture 1 for all empty board positions with n at most 7.

For a state S and a continuation of legal moves T, $S+T$ is the state obtained by playing T from S. In such a sequence, ϕ denotes pass.

Theorem 2. *For n-PLGo with $n \geq 2$, let* S *be a state in which neither player has played at cells 1 nor 2. Then for each player x with opponent y, $\mu_x(S' = S + x1\, y2) \leq \mu_x(S'' = +x\phi\ y2)$.*

Proof. S' and S'' have the same position and histories, except that S' contains the position P of $S' + x1$ whereas S'' does not. Consider an optimal x-strategy Π for S'. Notice that x can also use Π for S'', since any move that is legal for x in a continuation of S' is legal in the corresponding continuation of S''. The move options for y will be the same in both continuations: every non-pass move by y leaves at least one y-stone on the board, so y can never make a move that would create $S + x1$.

We can use Theorem 2 to prune some cases that would be covered by Conjecture 1. At state S, consider the three moves in this relative order: 2 before pass before 1. Upon arriving at the move to 1, compare the current lower bound α on $\mu_x(S)$ with the current upper bound β on $\mu_x(S + x\phi\, y2)$: if $\alpha > \beta$ then move 1 can be pruned.

Define E as the state with empty board and empty history. Let b be Black, the first player. For each cell j, define E_j as $E + bj$, i.e. the state obtained after b plays at cell j, so in each case with player-to-move White.

Theorem 3 and Corollary 1 are depicted in Fig. 5.

Theorem 3. $\mu_b(E + \phi) = -\mu_b(E)$.

Proof. The opponent can steal b's strategy.

Corollary 1. *For n-PLGo with $n \geq 2$, $\mu_b(E_1) \leq -\mu_b(E_2)$.*

Proof. From E_1, White can play at cell 2 and steal Black's strategy from E_2.

PLGo has some positions whose associated states are always safe under alternating play. Call a state *stable* if its minimax value equals its position score. For instance, consider the left state in Fig. 2: the union of territory and stones is {1,2} for Black and {4,5,6} for White, so the current Black score is −1. But the Black minimax value is +6, so this state is not stable. Here is another example: any state with position `(-x-x-)` is stable: `o`'s only move is pass, and the position score of +5 for x is maximum possible.

For n-PLGo with $n \geq 3$, for a player z and a state S, call S *z-total* if every cell in S is either empty or z-occupied, and z has never played at any of the empty cells, and one of these three conditions holds: 1 and n are empty, 2 and $n-1$ are occupied, and every gap between consecutive z-blocks has size at most 2; or 1, n, $n-1$ are empty, 2 and $n-2$ are occupied, and every gap between consecutive z-blocks has size exactly 1; or the symmetric case to the previous obtained by relabeling 1,...,n as n,...,1 respectively. For instance, `-x--xxx-x-xx-` and `-xx-xxx-x-x--` are x-total, whereas neither `-x--xx--` nor `-x--xx` is x-total. Call a state *total* if, for one of the players z, it is z-total.

Theorem 4. *For n-PLGo, every z-total state is stable, with z minimax value $+n$.*

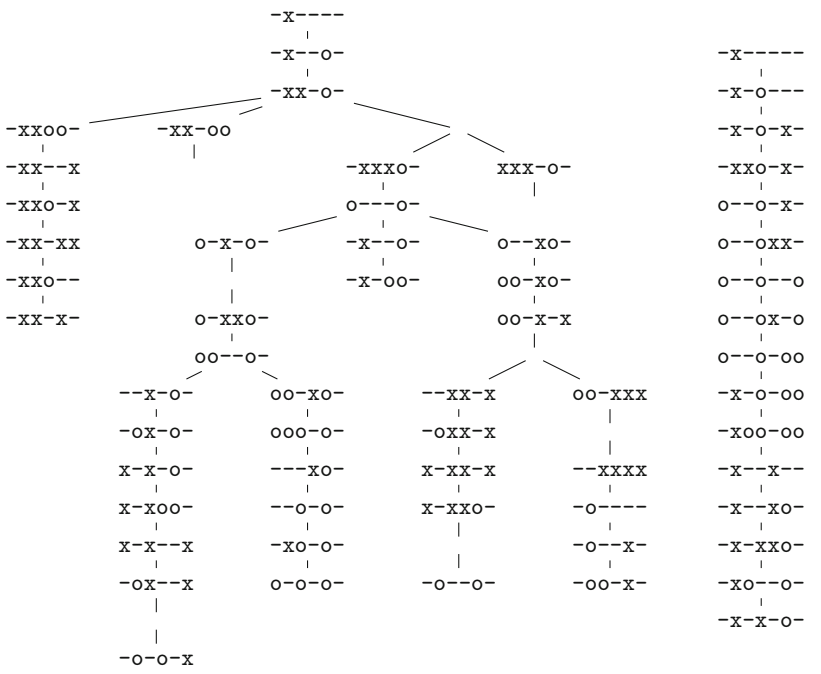

Fig. 6. Left: main lines showing state `(-13-2-)` is stable: neither player benefits by moving. Right: after showing `(-x-x-o-)` states are stable, principal variation of proof that `(-1-3-2-)` is stable.

Proof. The opponent cannot play into any gap of size 1: that would be suicide. For the first case, if the opponent plays into a gap of size 2, z can reply in the same gap and capture the opponent, leaving a gap of size 1. For the last two cases, if the opponent plays at the end gap of size two, z can reply in the same gap and capture. After that, the opponent has no legal moves, either by ko or suicide.

Consider $1 \times n$ Go. An *end* cell is cell 1 or cell n. For a position and a fixed player, (i) a *chain* is a maximal set of connected stones; (ii) two chains are *adjacent* if separated by exactly one cell, which is empty; (iii) a *group* is a maximal set of chains, none containing an end, each adjacent to at least one other in the set. For instance, in `(xx-xx-x-x-o--xx-)` x has 2 groups (cells 4 to 9, and 13,14) and o has 1 group (at 11).

For n-PLGo with $n \geq 3$, a state is *loosely packed* if neither player has ever played at any cell that is now empty and its own territory (although the opponent might have played there and been captured), cells 1 and n are empty, cells 2 and $n-1$ are occupied, and the gap between two consecutive chains is exactly 1. For instance, states with position `-x-o-xxx-oo-` are loosely packed, states with position `-x-o-x` are not, and states with position `-x-x-o-` are loosely packed as long as `x` has never played at cell 3.

Theorem 5. *For n-PLGo with $n \leq 7$, every loosely packed state is stable.*

Proof (sketch). By case analysis of each position, which we have verified by computer. For positions such as `-x-x-o-` in which each chain has size 1, the proofs are relatively straightforward. When one chain is larger the proofs can be longer, especially when n is even. See Figs. 6 and 7.

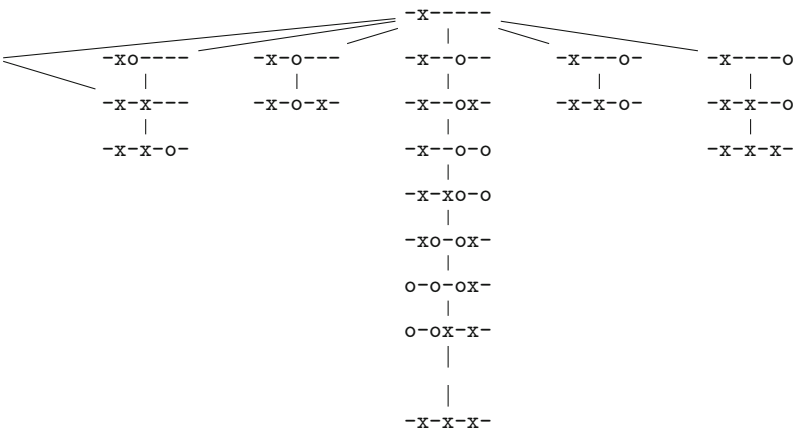

Fig. 7. Main lines of proof graph for 7-PLGo after using Theorem 5. 1st player minimax value is +2. Empty node is pass. Moves in a proof graph need not be strongest: e.g., strongest reply to `(-xo----)` is `(-xo--x-)`, not `(-x-x---)` as shown.

Theorem 5 cannot be extended. The 8-PLGo state with move sequence (**2** 7 **3** 5) to position `(-xx-o-o-)` is loosely packed but not stable: from this position the main line is (**4** 6 **8** 6 **pass** 7) leaving `(-xxx-oo-)` with $\mu_x = +1$. Also, the 8-PLGo state (**2** 4 **7** **6**) to position `(-x-o-xx-)` is loosely packed but not stable: `o` cannot attack but `x` can, with main line (**5** 8 **5** p **3** p **7**) leaving `(-xx-x-x-)` with $\mu_x = +8$.

For a n-PLGo position P with $n \geq 3$, a *telomere* is a subsequence of P that includes exactly one of the board ends, i.e., for some $t \geq 2$, induced by cell set either $\{t, t+1, \ldots n\}$ or $\{1, 2, \ldots, n+1-t\}$. The *complement* of a telomere T with cell set S is the telomere whose cell set is the complement of T with respect to $\{1, 2, \ldots, n\}$. For instance, for $n = 4$, the complement of the telomere induced by $\{2, 3, 4\}$ is the telomere induced by $\{1\}$. The *end* of a telomere is the end of the sequence corresponding to the end position—either 1 or n—of the board; the *front* of a telomere is the other end. When writing telomeres, we put a parenthesis at the end and a square bracket at the front. For instance, for position `(-x-oox)`, `(-x]` and `[-oox)` are complementary telomeres.

For a player z and a position P, a *z-telomere* is a telomere whose front cell is z, and within the telomere the chain containing the front cell has two liberties,

and for the state S whose history consists only of position P, the opponent of z has no legal moves into the cells of the telomere. For instance, `(-x]` is an `x`-telomere, since for the state with position `(-x-oox)`, `o` has no legal moves to cell 1. Similarly, `[xo-x-)` is not an `x`-telomere of position `(o-xo-x-)`, since the front cell does not have a liberty.

For the most recent position P of a state S, a telomere is *fresh* if that telomere does not appear at those locations in any earlier position in the history of S.

Theorem 6. *Let* S *be a state with a fresh* `x`*-telomere with complement* `(-o-]` *and let* P *be the most recent position of* S*. Then* $\mu_x(S) \geq \mu_x(P)$*.*

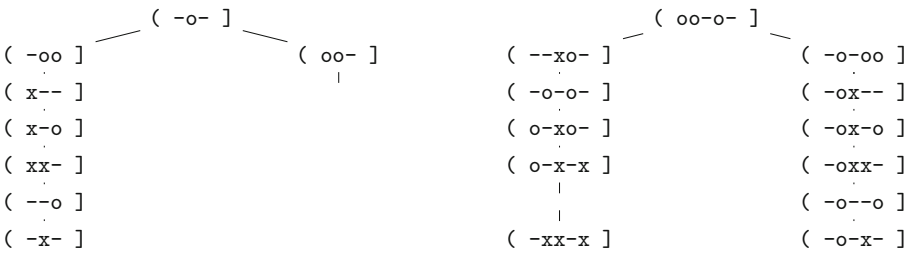

Fig. 8. Proving telomere properties of Theorems 6 and 8.

Proof. If it is `x` to play, `x` can pass. So assume it is `o` to play. If `o` passes either the game ends (if `x` played first) or `x` can pass, and the game ends with final score $\mu_x(P)$ and we are done. There are two other options for `o`. Figure 8 shows the lines of play for these options. If at any point `o` passes then `x` can pass, or continue and increase the score even further.

Theorem 7. *Let* S *be a n-PLGo state with* `o` *to play and a fresh* `x`*-telomere with complement as shown in Fig. 9. Then* $\mu_x(S)$ *is at least as shown, and pruning* `o` *moves as shown does not change the minimax value.*

Proof (sketch). Case `(-o-]` follows from Theorem 6. Consider case `(-o--]`. `o` moves to 1 and 4 can be pruned: `x` replies at 3 and scores at least n.

Consider case `(----]`. If `o` plays at 1 or 4 then `x` replies at 3 and eventually scores n, so these moves can be pruned. If `o` plays at 2 then `x` replies at 4, reducing to case `(-o-]`, so `x` scores at least $n - 5$. If `o` plays at 3 then `x` replies at 2 and `o` loses all unless capturing at `1`, then `x` passes and `o` can pass and score $n - 7$ or play at 4 and lose all or play at 2 and leave `(ooo-]`, in which case `x` captures at 4 and scores at least $n - 5$ by case `(---]`.

The arguments for other cases are similar. We omit the details.

Theorem 8. *Let* S *be a state with a fresh* `x`*-telomere with complement* `(-o-o-]` *and let* P *be the most recent position of* S*. Then every move for* `o`*, with possible exception of playing at cell 3, yields to a state* S' *with* $\mu_x(S') \geq \mu_x(P)$*.*

complement	prune	lower bound on μ_x
(--]	(**]	n
(o-]	(o*]	n
(---]	(*-*]	$n-5$
(o--]	(o**]	n
(-o-]	(*o*]	$n-5$
(--o]	(oo*]	n
(----]	(*--*]	$n-7$
(o---]	(o***]	n
(-o--]	(*o-*]	$n-7$
(--o-]	(*-o*]	$n-7$
(---o]	(oo-*]	$n-5$
(o-o-]	(o-o*]	$n-5$
(-oo-]	(-oo-]	?
(ooo-]	(ooo*]	$n-5$
(-----]	(*----]	?
(o----]	(o****]	n

Fig. 9. Pruning and bounds for Theorem 7; * cells pruned; ? no bound.

Proof. Similar to the proof of Theorem 6. It suffices to consider the cases of Fig. 8. One case arrives at (-o-x-] and leads to (-o-]: use Theorem 6 to finish the proof.

3 A PLGo Solver

Following van der Werf et al. [3–5] we implemented a solver based on alpha-beta search. We considered two variants, Aspiration Window Search by Shams et al. [11,12] and MTD(f) Search by Plaat et al. [13]. Initial tests showed the latter to be more effective, so we chose it. Our implementation is enhanced with (1) iterative deepening, (2) transposition tables, (3) enhanced move ordering seeded by game knowledge, and (4) knowledge based pruning, described in the following subsections.

It is non-trivial to extract a principal variation from MTD(f) search results, so once the root state is solved we find a principal variation by searching again with an aspiration window around the known minimax value.

Some PLGo scores and variations are shown in Figs. 10, 11 and 12.

3.1 Iterative Deepening

Iterative deepening, a commonly used search enhancement, iteratively re-searches with a gradually increasing search depth cutoff until the value of the state is determined. Transposition tables (see below) prevent much work from smaller depth cutoffs from being repeated. If it is required that the value of all nodes in the search tree are known exactly, then iterative deepening offers no advantage. However, all that is required for alpha-beta search to terminate is for

n	minimax value by 1st-move location
2	−2 –
3	−3 3 –
4	−4 4 – –
5	−5 0 0 – –
6	−6 1 −1 – – –
7	−7 2 −2 2 – – –
8	−3 3 −1 1 – – – –
9	−4 0 −1 0 0 – – – –

Fig. 10. n-PLGo empty-board minimax values. Missing entries follow by left-right symmetry.

the upper and lower bounds to meet: this is known as a beta-cutoff. In PLGo it is often the case that some child nodes are easy to solve while others are not. Iterative deepening allows us to solve easier children first, causing a beta-cutoff to occur earlier than if more costly children had been searched first. If the search fails to reach the beta-cutoff using only information from children solved with the current depth cutoff, it returns an estimated value that is used as a guess in MTD(f). However, a beta-cutoff does not occur with all nodes, so in some cases all children must be fully searched. Our enhanced move ordering, described below, also exploits iterative deepening.

n	n-PLGo short principal variations by 1st-move
2	**1** 2 –
3	**1** 2 **2** –
4	**1** 3 2 **2** 3 **4** – –
5	**1** 4 **2** 3 **2** 1 **2** 4 **3** 4 **2** 1 **2** – –
6	**1** 5 **2** 3 **2** 1 **2** 5 **3** **3** 2 **1** 5 **2** 4 **2** – – –
7	**1** 6 **4** 2 **5** 3 **4** 5 **2** 6 **4** **3** 2 **1** 6 **2** 4 **2** **4** 6 **2** – – –
8	**1** 7 **3** 6 **2** 4 **2** **2** 7 **5** **3** 2 **7** 4 **6** **4** 7 **2** 6 – – – –
9	**1** 3 **8** 6 **4** 5 **2** 8 4 6 **3** 2 **6** 8 4 7 **9** 5 4 8 **5** 4 2 **8** 6 **5** 2 **8** 4 **6** 7 **6** – – – –

Fig. 11. PLGo short empty-board principal variations: after the first move, each player plays a strong move that leads to a relatively short game. Black moves in **bold**.

3.2 Transposition Tables

Using transposition tables in Go is made more difficult by superko rules, which require the history in addition to the current position, causing game states to be of variable size and often quite large. In game playing software, it is generally sufficient to simply use a Zobrist hash of the game state as the transposition table key since the decrease in performance from testing the entire state for equality

solving 8-PLGo 1.b1					
move 2	score	seconds	nodes	search depth	short pv
w2	−3	8186	11879007487	31	2 **7** 5 **3** 2
w3	−3	1879	2555094103	31	3 **7** 5 2
w4	−1	149	205699259	35	4 **7** 3 **6** 2
w5	−3	8202	11759010780	31	5 **3** 4 **7** 2
w6	1	180	258613192	31	6 **7** 2 **5** 3 **6**
w7	−3	33160	50349939752	43	7 **3** 6 **2** 4 **2**
w8	1	8051	11489810583	57	8 **6** 3 **5**

Fig. 12. The hardest 8-PLGo opening. In each short pv, each player avoids prolonging the game. E.g. with **1** 4 **7** 5 **2** 3 **6** 8 **2** 7 **6** 7 **8** 5 **3** 7 White plays optimally but prolongs the game, and with **1** 4 **7** 5 **2** 3 **2** 1 **6** 8 **6** 7 **2** **6** **7** **8** 5 **4** 7 **5** 3 **5** 1 **4** 6 **4** 5 **2** 4 **8** 4 **7** 5 6 **7** 8 **3** 1 **3** 2 **7** **3** 2 **1** 4 **5** 2 **4** 6 **4** 8 **5** 3 **5** **7** 4 **1** 4 **6** 8 **6** 5 **3** 4 **5** 2 **7** 4 8 **6** 3 **7** both players play optimally but prolong the game.

outweighs that of occasional hash collisions. However, when solving games, hash collisions can lead to incorrect results if the entire state is not also checked for equality. To tackle this problem, we use the transposition table sparingly, only saving results to the table when they represent a sufficiently large subtree of the search and are known to be exact. Our replacement scheme prefers nodes which represent a larger amount of completed work; additionally, PV nodes are preferred over CUT nodes or ALL nodes. An improvement might be to use graph history interaction methods that allow states with the same position to be considered as a unit for lookup purposes [14,15].

3.3 Enhanced Move Ordering

Search algorithms such as alpha-beta perform best when the best move is searched first. Our knowledge of PLGo enabled us to construct a heuristic move order that takes advantage of this property. Following Tromp, we try the pass move first [16]:

1. pass,
2. cells 2 and $n-1$ if cells 1 and n respectively are empty,
3. even numbered cells, counting inwards from each board end,
4. capturing moves,
5. moves that do not create self-atari,
6. all other cells, except 1 and n,
7. cells 1 and n.

This ordering is used when a board is searched for the first time. If the same board is searched again, moves are first ordered based on whether they are exact (such nodes will likely be in the transposition table), then by the estimated score from the previous iteration of iterative deepening, and lastly by the computational effort (estimated subtree size) required to search them. We use a hash table

that ignores collisions to store this ordering information, frequent hash collisions can yield poor move ordering. In practice we find collisions occur rarely, so the benefits of a fast hash table outweigh the disadvantage of an occasionally poor move ordering.

3.4 Knowledge Based Pruning

We use the theorems presented in this paper to reduce search space. In particular, Theorems 6 and 8 are used as follows. At each node in the search tree, we check whether the opponent's most recent move was to a cell where she had never played: in such cases we check whether either theorem applies, and adjust alpha-beta bounds and/or prune the search accordingly.

3.5 Knockout Tests

To show the relative impact of our solver features, we ran a knockout test (also called ablation test, i.e., when features are removed) on the hardest 7-PLGo and easiest 8-PLGo openings, see Fig. 13. As expected, iterative deepening move ordering is beneficial. Other results are surprising; telomere recognition is a slight detriment, perhaps because most cases covered are easily solved; and the transposition table is detrimental on easily solved positions, perhaps because hashing by full history yields few successful lookups but requires much writing. The 1–2 conjecture is not helpful on the 8-PLGo opening, perhaps because pruning results in some cutoffs not being found.

Figure 14 gives the solution to the first problem posed in Fig. 2.

solver feature knockout test			
instance	feature removed	time (sec)	time / all-features time
7-PLGo 1.b3	—	.44	1.0
	1-2 conjecture	.52	1.18
	total state	.93	2.11
	iter. deepening move ordering (IDMO)	2.1	4.77
	IDMO and knowledge move ordering	27.8	63.18
	loosely packed stable	.73	1.66
	telomere	.42	.95
	transposition table (TT)	.30	.68
8-PLGo 1.b4	—	25.0	1.0
	1-2 conjecture	14.3	.57
	total state	83.4	3.34
	IDMO	810.9	32.4
	telomere	24.0	.96
	TT	14.7	.588

Fig. 13. Solver feature knockout test.

```
-x-ox-   -x-o-o   -xxo-o   o--o-o   o-xo-o   oo-o-o
      oo-ox-   p   --x-x-   -ox-x-   x-x-x-
```

Fig. 14. Principal variation for Fig. 2: `x` (black) captures all cells, minimax score +6.

4 Conclusion and Future Work

We have explored properties of Positional Linear Go and implemented a solver. Our approach is motivated more by game theory than by algorithmic design. It would be of interest to show further properties and to build a stronger solver.

Acknowledgments. We are grateful to Martin Müller, Erik van der Werf, Victor Allis, and the referees for helpful comments, and to the NSERC Discovery Grants Program for research funding.

References

1. Moyer, C.: How Google's AlphaGo beat a Go world champion. Atlantic (2016)
2. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
3. van der Werf, E.: 5×5 Go is solved (2002). http://erikvanderwerf.tengen.nl/5x5/5x5solved.html. Accessed 01 Jan 2017
4. van der Werf, E.C., van den Herik, H.J., Uiterwijk, J.W.: Solving Go on small boards. ICGA J. **26**, 92–107 (2003)
5. van der Werf, E.: AI techniques for the game of Go. Ph.D. thesis, Maastricht University (2004)
6. van der Werf, E.: First player scores for $M \times N$ Go (2009). http://erikvanderwerf.tengen.nl/mxngo.html. Accessed 01 Jan 2017
7. van der Werf, E.C., Winands, M.H.M.: Solving Go for rectangular boards. ICGA J. **32**, 77–88 (2009)
8. Tromp, J.: The game of Go aka Weiqi in Chinese, Baduk in Korean. http://tromp.github.io/go.html. Accessed 01 Jan 2017
9. Tromp, J.: Number of legal Go positions (2016). https://tromp.github.io/go/legal.html. Accessed 01 Jan 2017
10. Müller, M.: Playing it safe: recognizing secure territories in computer Go by using static rules and search. In: Proceedings of Game Programming Workshop, Computer Shogi Association (1997)
11. Chess Programming Wiki: Aspiration windows (2017). https://chessprogramming.wikispaces.com/Aspiration+Windows. Accessed 01 Jan 2017
12. Shams, R., Kaindl, H., Horacek, H.: Using aspiration windows for minimax algorithms. In: Proceedings of IJCAI 1991, pp. 192–197. Morgan Kaufmann Publishers (1991)
13. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-first fixed-depth minimax algorithms. Artif. Intell. **87**, 255–293 (1996)

14. Kishimoto, A., Müller, M.: A general solution to the graph history interaction problem. In: 19th National Conference on Articial Intelligence, AAAI, pp. 644–649 (2004)
15. Kishimoto, A.: Correct and efficient search algorithms in the presence of repetitions. Ph.D. thesis, University of Alberta (2005)
16. Tromp, J.: Solving 2 × 2 Go (2016). https://tromp.github.io/java/go/twoxtwo.html. Accessed 01 Jan 2017