An Improved Safety Solver for Computer Go

Xiaozhen Niu and Martin Müller

Department of Computing Science,
University of Alberta, Edmonton, Canada
{xiaozhen, mmueller}@cs.ualberta.ca

Abstract. Most Go-playing programs use a combination of search and heuristics based on an influence function to determine whether territories are safe. However, to assure the correct evaluation of Go positions, the safety of stones and territories must be proved by an exact method.

The first exact algorithm, due to Benson [1], determines the unconditional safety of stones and completely surrounded territories. Müller [3] develops static rules for detecting safety by alternating play, and introduces search-based methods.

This paper describes new, stronger search-based techniques including region-merging and a new method for efficiently solving weakly dependent regions. In a typical final position, more than half the points on the board can be proved safe by our current solver. This almost doubles the number of proven points compared to the 26.4% reported in [3].

1 Introduction

This paper describes recent progress in building a search-based solver for proving the safety of stones and territories. The main application is late in the game, when much of the board has been partitioned into relatively small areas that are completely surrounded by stones of one player. In previous work [3], the analysis of such positions was done by a strict divide and conquer approach, analyzing one region at a time. The current solver implements several techniques that relax this strict separation approach. One technique merges several *strongly related* regions into a single one for the purpose of search. Another technique deals with how to search separately a set of *weakly dependent* regions in order to prove the safety of the union of all these regions.

This paper is organized as follows: the remainder of the introduction describes the terminology and previous work. Section 2 provides a formal framework for the safety prover, and Section 3 describes the solver: it gives an overview of the steps in the proving algorithm, describes the new technique of region merging and introduces the concept of weakly dependent regions. Section 4 deals with search enhancements, and Section 5 describes the experimental setup and results, followed by conclusions and future work in Section 6.

1.1 Preliminaries: Terminology and Go Rules

Our terminology is similar to [1, 3], with some additional definitions. Differences are indicated below. A *block* is a connected set of stones on the Go board. Each block has a number of adjacent empty points called *liberties*. A block that loses its last liberty is *captured*, i.e., removed from the board. A block that has only one liberty is said to be *in atari*. Figure 1 shows two black blocks and one white block. The small black block ▲ contains two stones, and has five liberties (two marked A and three marked B).

Given a color c, let $A_{\neg c}$ be the set of all points on the Go board which are *not* of color c. Then a *basic region* of color c (called a region in [1, 3]) is a maximal connected subset of $A_{\neg c}$. Each basic region is surrounded by blocks of color c. In this paper, we also use the concept of a *merged region*, which is the union of two or more basic regions of the same color. We will use the term region to refer to either a basic or a merged region. In Figure 1 A and B are basic regions and $A \cup B$ is a merged region.

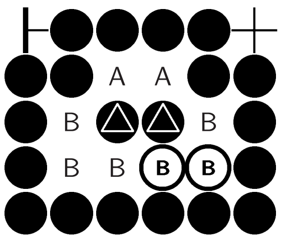


Fig. 1. Blocks, basic regions and merged regions

We call a block b *adjacent* to a region r if at least one point of b is adjacent to one point in r. A block b is called *interior block* of a region r if it is adjacent to r but no other region. Otherwise, if b is adjacent to r and at least one more region it is called a *boundary block* of r. We denote the set of all boundary blocks of a region r by $Bd(r)$. In Figure 1, the black block ▲ is a boundary block of the basic region A but an interior block of the merged region $A \cup B$. The *defender* is the player playing the color of boundary blocks of a region. The other player is called the *attacker*.

Our results are mostly independent of the specific rule set used. As in previous work [1, 3], suicide is forbidden. Our algorithm is incomplete in the sense that it can only find stones that are safe by two sure liberties [3]. This excludes cases such as conditional safety that depends on winning a ko, and also less frequent cases of safety due to double ko or snapback. The solver does not yet handle coexistence in *seki*.

1.2 Previous Work on Safety of Blocks and Territories

Benson's algorithm for *unconditionally alive blocks* [1] identifies sets of blocks and basic regions that are safe, even if the attacker can play an unlimited number of moves in a row, and the defender never plays. Müller [3] defines static rules for detecting safety by *alternating play*, where the defender is allowed to reply to each attacker move. [3] also introduces local search methods for identifying regions that provide one or two *sure liberties* for an adjacent block. Experimental results for a preliminary implementation in the program Explorer were presented for Benson's algorithm, static rules, and a 6-ply search.

Van der Werf implemented an extended version of Müller's static rules to provide input for his program that learns to score Go positions [6]. Vilà and Cazenave developed static classification rules for many classes of regions up to a size of 7 points [7]. These methods have not been implemented in our solver yet.

1.3 Contributions

The main contributions of this paper are as follows.

- A new method to merge regions and a search technique to prove that a merged region provides two sure liberties.
- A new divide-and-conquer analysis of weakly dependent regions.
- A greatly improved $\alpha\beta$ search routine for finding 2-vital regions (defined in Subsection 2.1), with improvements in move ordering, evaluation function, and pruning by recognizing forced moves.
- An improvement that takes external eyes of boundary blocks into account during the search.

2 Establishing the Safety of Blocks and Territories

Below we describe three definitions (in 2.1) and discuss the recognition of safe regions (in 2.2).

2.1 Definitions

The following definitions, adapted from [3], are the basis for our work. They are used to characterize blocks and territories that can be made safe under alternating play, by creating two sure liberties for blocks, and at the same time preventing the opponent from living inside the territories. During play, the liberty count of blocks may decrease to 1 (they can be in atari), but they are never captured and ultimately achieve two sure liberties.

Regions can be used to provide either one or two liberties for a boundary block. We call this number the *Liberty Target* $LT(b, r)$ of a block b in a region r. A search is used to decide whether all blocks can reach their liberty target in a region, under the condition of alternating play, with the attacker moving first and winning all ko fights.

Definition: Let r be a region, and let $Bd(r) = \{b_1, \ldots, b_n\}$ be the set of boundary blocks of r. Let $k_i = LT(b_i, r)$, $k_i \in \{1, 2\}$, be the liberty target of b_i in r. A defender strategy S is said to *achieve all liberty targets* in r if each b_i has at least k_i liberties in r initially, as well as after each defender move.

Each attacker move in r can reduce the liberties of a boundary block by at most one. The definition implies that the defender can always regain k_i liberties for each b_i with his next move in r. The following definition of life under alternating play is analogous to Benson's.

Definition: A set of blocks B is *alive under alternating play* in a set of regions R if there exist liberty targets $LT(b,r)$ and a strategy S that achieves all these liberty targets in each $r \in R$ and

$$\forall b \in B \quad \sum_{r \in R} LT(b,r) \geq 2$$

Note that this construction ensures that blocks will never be captured. Initially each block has two or more liberties. Each attacker move in a region r reduces only liberties of blocks adjacent to r, and by at most 1 liberty. By the invariant, the defender has a move in r that restores the previous liberty count. Each block in B has at least one liberty overall after any attacker move and two liberties after the defender's local reply.

It is easy to adapt this definition to the case where blocks have sure external liberties outside of R. The sum of liberty targets for such blocks can be reduced to 1, if the block has one sure external liberty, or 0, if the block is already safe.

Definition: We call a region r *one-vital* for a block b if b can achieve a liberty target of one in r, and *two-vital* if b can achieve a target of two.

2.2 Recognition of Safe Regions

The attacker *cannot live inside* a region surrounded by safe blocks if there are no two non-adjacent potential attacker eye points, or if the attacker eye area forms a *nakade* shape. Our current solver uses a simple static test for this condition as described in [3].

3 Methods for Processing Regions

We describe the structure of the safety solver (3.1), region merging (3.2), weakly dependent regions (3.3), and other improvements to the solver (3.4).

3.1 The Structure of the Safety Solver

Our safety solver includes five sub-solvers.

Benson solver – implements Benson's classic algorithm to analyze unconditional life.

Static solver – uses static rules to recognize safe blocks and regions under alternating play, as described in [3]. No search is used.

1-Vital solver – uses search to recognize regions that are 1-vital for one or more boundary blocks. As in [3] there is also a combined search for 1-vitality and connections in the same region, that is used to build chains of safely connected blocks.

Generalized 2-Vital solver – uses searches to prove that each boundary block can reach a predefined liberty target. For *safe blocks*, the target is 0, since their safety has already been established using other regions. Blocks that have one external eye outside of this region are defined as *external eye blocks*. For these blocks the target is 1. For all other non-safe blocks the target is 2 liberties in this region. All the search enhancements described in the next section were developed for this solver.
The 2-Vital solver in [3] could not handle external eye blocks, it would try to prove 2-vitality for all non-safe boundary blocks.

Expand-vital solver – uses searches to prove the safety of partially surrounded areas, as in [3]. This sub-solver can also be used to prove that non-safe stones can connect to safe stones in a region.

The basic algorithm of the safety solver is as follows.

1. The static solver is called first. It is very fast and resolves the simple cases.
2. The 2-Vital solver is called for each region. As a simple heuristic to avoid computations that most likely will not succeed, searches are performed only for regions up to size 30.
3. The Expand-vital solver is called for regions that have some safe boundary blocks. The safety of those blocks has been established by using other regions. Our previous solver in [3] only used the steps so far.
4. (New) Region merging. After the previous steps, all the easy-to-prove safe basic regions have been found. In this step the remaining unproven related regions are merged. For each small-enough merged region (up to size 14 in the current implementation) the generalized 2-Vital solver is called. The mechanism is described in detail in Subsection 3.2.
5. (New) Weakly dependent regions. A new algorithm deals with weakly dependent regions. In this step both the 1-Vital solver and the 2-Vital solver are used. A detailed description is given in Subsection 3.3.
6. (New) As in step 3, the Expand-vital solver is called for those regions for which one or more new safe boundary blocks have been found.

3.2 Region Merging

One of the major drawbacks of our previous solver is that it processes basic regions one by one and ignores the possible relationship between them. Figure 2 shows an example of two related regions. The previous solver treats regions A and B separately, and neither region can be solved. However the merged region $A \cup B$ can be solved easily.

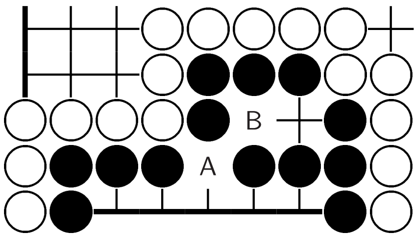


Fig. 2. Two related regions

The first algorithm step scans all regions and merges all related regions.

Two regions are defined as *related* if they have a common boundary block. After the merging step, the 2-Vital solver is used to recognize safe merged regions.

This method can solve simple cases such as the one in Figure 2. However, since merging all related regions usually creates a very large merged region, the search space often becomes too large.

To improve the locality of search, we distinguish between *strongly dependent* regions, which share more than one common boundary block, and *weakly dependent* regions with exactly one common boundary block.

Our current solver uses a two-step merging process. In the first step, strongly dependent basic regions are merged. In the second step *groups of weakly dependent regions* are formed. A *group* can contain both basic regions and merged regions computed in the first step. Figure 3 shows an example.

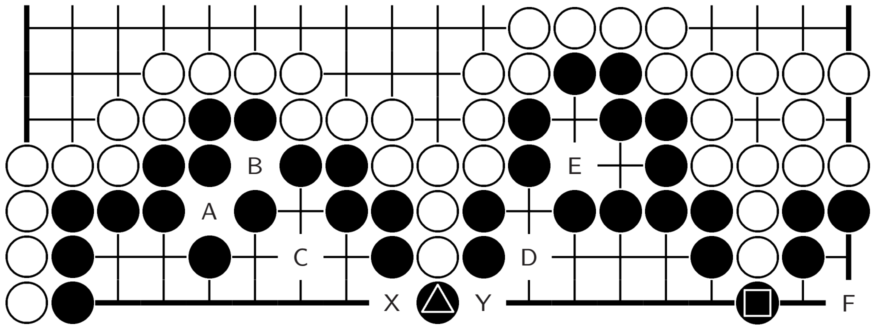


Fig. 3. Strongly and weakly dependent regions

In this figure, there are total of 6 related black regions A, B, C, D, E, and F. Since the huge outside region contains surrounding white stones that are already safe, we do not need to consider the huge outside region.

A complete merge of all six regions yields a combined new region with size 32, which is too large to be fully searched. Two-step merging creates the following result: The first step identifies connected components of strongly dependent regions and merges them. A, B and C are strongly dependent and are merged into a new region $R_1 = A \cup B \cup C$. Next D and E are merged into $R_2 = D \cup E$. Region F is not strongly dependent on any other region and is not merged. The second step identifies weak dependencies between R_1, R_2 and F and builds the group. R_1 and R_2 are weakly dependent through block ▲, and R_2 and F are weakly dependent through block ■. The result is a group of weakly dependent regions $\{R_1, R_2, F\}$ with region sizes of 15, 14 and 3 respectively. The regions within a group are not merged but searched separately, as explained in the next section.

The common boundary block between two weakly dependent regions has both *internal* and *external* liberties relative to each region. For example, for block ▲ and $R_2 = D \cup E$, the liberty Y is internal and the liberty X is external.

3.3 Weakly Dependent Regions

We distinguish between two types of weak dependencies. In type 1, the common boundary block has more than one liberty in each region. For example, in Figure 4 the shared boundary block of regions A and B has more than 1 liberty in each region. In type-1 dependencies, our search in one region does not consider the external liberties of the common block.

In type-2 weak dependencies, the common boundary block has only one liberty in at least one of the regions. In Figure 3 black block ▲ has only 1 liberty in both regions R_1 and R_2. We need to consider the external liberties for the common block because moves in one region might affect the result of the other. However, we do not want to merge these two regions because of the resulting increase in problem size.

The pseudo code in Figure 5 describes the method for processing groups of weakly dependent regions.

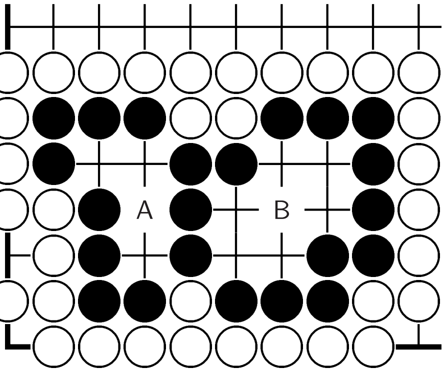


Fig. 4. First type of weakly dependent regions

```
for each weakly dependent group $G$
    if ( total size of all regions in $G < 14$) // 14 is a constant determined empirically
        $r_G$ = merge all regions in $G$;
        call 2-vital solver for $r_G$
    else
        for each region $r \in G$
            for each shared boundary block $b$ between $r$ and another region $r_2 \in G$
                do a 1-vital search for $b$ in $r_2$;
            reduce liberty target for all successfully tested boundary blocks to 1
            take unproven (1-vital search not successful) blocks as special blocks;
            generate external moves for special blocks (for both attacker/defender);
            call 2-vital solver for $r$.
```

Fig. 5. Search for weakly dependent groups

3.4 Other Improvements to the Solver

The following further enhancements were made to the solver since the version described in [3].

Improved static solver – the static solver contains a more complete implementation of the concepts in [3] than the preliminary version used in the 1997 experiments. Its performance is about 3 to 4 % better on the same test set. See Subsection 5.1 for detailed results.

External eyes of blocks – if a boundary block of a region r has one sure liberty elsewhere, this information is stored and used in the search for r by lowering the liberty target for that block.

Time limit instead of depth limit – the previous experiments used a fixed-depth 6-ply search. The current solver uses a time limit instead, which allows it to search much deeper.

4 Search Enhancements

Below we discuss move generation and move ordening (in 4.1) and evaluation functions (in 4.2).

4.1 Move Generation and Move Ordering

In this work, we focus on proving that a region and its boundary blocks are safe. Therefore we have concentrated our efforts on generating and ordering the defender's moves. For the attacker, all legal moves in the region plus a pass move will be generated. When processing weakly dependent regions as described in Subsection 3.3, extra moves outside of the region might be generated for either attacker or defender. For details of this procedure please see [5]. The attacker is allowed to recapture immediately a ko. Therefore, the attacker will always win a ko-fight inside a region.

The solver utilizes Explorer's generic $\alpha\beta$ search engine which implements standard move-ordering techniques such as (1) trying the best move from a previous iteration first and (2) killer moves. There is currently no game-specific move ordering or pruning for the attacker. For the defender, the following safe forward-pruning technique is used: when a boundary block of a region is in atari, only moves that can possibly avert the capturing threat, such as extending the block's liberties or capturing the attacker's adjacent stones are generated. If no forced moves are found, all legal moves for the defender are generated.

For ordering the defender's moves, both a high-priority move-motivation detector and a normal scoring system are used. The motivation detector analyzes the purpose of the attacker's previous move, and classifies the situation as one of three priorities.

1. The attacker's move is close to one of the empty cutting points.
2. The attacker's move extends one or more cutting blocks.
3. Other attacker moves.

For priority-1 and priority-2 positions, a set of high-priority moves according to the attacker's motivation is generated first. For priority 1, most likely the attacker is trying to cut, so the cutting points close to this move, as well as the cutting points' 8 neighbor points, have high priority. For priority 2, most likely the attacker is trying to expand its own cutting block. Capturing this block is an urgent goal for the defender. Therefore, all liberties of this block are given high priority. The number of adjacent empty points is used to order liberties.

All moves in priority-3 positions and all remaining moves in priority-1 and priority-2 positions are sorted according to a score that is computed as a weighted sum

$$Move\ score = f_1 * LIB + f_2 * NDB + f_3 * NAB + f_4 * CB + f_5 * AP.$$

The formula uses the following five features.

1. Liberties of this defender's block (LIB).
2. Number of neighboring attacker's blocks (NAB).
3. Number of neighboring defender's blocks (NDB).
4. Capture bonus (CB): 1 if an opponent block is captured, 0 otherwise.
5. Self-atari penalty (AP): -1 if move is self-atari, 0 otherwise.

The following set of weights worked well in our experiments: $f_1 = 10, f_2 = 30, f_3 = 20, f_4 = 50, f_5 = 100$.

4.2 Evaluation Functions

Heuristic Evaluation Function – The evaluation function in [3] used only three values: *proven-safe*, *proven-unsafe* and *unknown*. Since most of the nodes during the search evaluate to *unknown*, we can improve the search by using a heuristic evaluation to differentiate nodes in this category. The heuristics are based on two observations.

1. An area that is divided into more subregions is usually easier to evaluate as *proven-safe* for our static evaluation function.
2. If the attacker has *active blocks* with more than 1 liberty, it usually means that the attack still has more chances to succeed.

Let NSR be the number of subregions and NAB be the number of the attacker's active blocks. Then the heuristic evaluation of a position is calculated by the formula

$$eval = f_1 * NSR + f_2 * NAB, \quad f_1 = 100, f_2 = -50$$

Exact Evaluation Function – The exact evaluation function recognizes positions that are *proven-safe* or *proven-unsafe*. A powerful function is crucial to achieve good performance. However, there is a tradeoff between evaluation speed and power. In our evaluation function there are two types of exact static evaluations, *HasSureLiberties()* and *StaticSafe()*. *HasSureLiberties()* is a quick static test to check whether all boundary blocks of a region have two sure liberties and the opponent cannot live inside the region. *StaticSafe()*, is a simplified static safety solver which takes the subregions created by the search into account. Because it has to compute regions, *StaticSafe()* is much slower than *HasSureLiberties()*. The relative speed of the two methods varies widely, but 5 to 10 times slower is typical. We use the following compromise rule: If the previous move changes the size of a region by more than 2 points, then *StaticSafe()* is used. Otherwise, the quicker *HasSureLiberties()* is used. In contrast, [3] used only a weaker form of *HasSureLiberties()*.

5 Experimental Setup and Results

The safety solver described here has been developed as part of the Go program Explorer [2]. To compare the performance of our current solver with the previous solver [3], our test set 1 is the same, the problem set *IGS_31_counted* from the Computer Go Test Collection [2]. The set contains 31 problems. Each of them is the final position of a 19 × 19 game played by human amateur players.

We also created an independent test set 2. It contains 27 final positions of games by the Chinese professional 9 dan player ZuDe Chen. Both sets are available at `http://www.cs.ualberta.ca/~mmueller/cgo/general.html`.

All experiments were performed on a Pentium 4 with 1.6 GHz and a 64MB transposition table. The following abbreviations for the solvers and enhancements are used in the tables.

Benson – Benson's algorithm, as in [3].
Static-1997 – static solver from [3].
Search-1997 – search-based solver, 6-ply depth limit, from [3].
Static-2004 – current version of static solver.
M1 – a basic 2-liberties search, similar to the one in [3].
M2 – M1 + consider external eyes of blocks as in Subsection 3.4.
M3 – M2 + region merging method as in Subsection 3.2.
M4 – M3 + move ordering and pruning as in Subsection 4.1.
M5 – M4 + improved heuristic and exact evaluation functions as in Subsection 4.2.
M6 – full solver, M5 + weakly dependent regions as in Subsection 3.3.

5.1 Experiment 1: Overall Comparison of Solvers

Table 1 shows the results for all methods listed above for test set 1. The set contains 31 full-board positions with a total of 31 × (19 × 19) = 11,191 points, 1,123 blocks and 802 regions. For methods M1–M6, a long time limit of 200 seconds per region was used. For results with shorter time limits, see Experiment 2.

Table 1. Search improvements in test set 1

Version	Safe points	Safe blocks	Safe regions
Benson	1,886 (16.9%)	103 (9.2%)	204 (25.4%)
Static-1997	2,481 (22.2%)	168 (15.0%)	N/A
Search-1997	2,954 (26.4%)	198 (17.6%)	N/A
Static-2004	2,898 (25.9%)	212 (18.9%)	321 (40.0%)
M1	4,017 (35.9%)	326 (29.0%)	404 (50.4%)
M2	4,073 (36.4%)	330 (29.4%)	406 (50.6%)
M3	5,029 (44.9%)	444 (39.5%)	495 (61.7%)
M4	5,070 (45.3%)	451 (40.2%)	498 (62.1%)
M5	5,396 (48.2%)	484 (43.1%)	519 (64.7%)
M6 (Full)	5,740 (51.3%)	523 (46.6%)	548 (68.3%)
Perfect	11,191 (100%)	1,123 (100%)	802 (100%)

Table 2 shows the results for all methods listed above for test set 2. This test set contains a total of 27 × (19 × 19) = 9,747 points, 1,052 blocks and 742 regions.

Table 2. Search improvements in test set 2

Version	Safe points	Safe blocks	Safe regions
Benson	1,329 (13.6%)	106 (10.1%)	160 (21.6%)
Static-2004	2,287 (23.5%)	188 (17.9%)	251 (33.8%)
M1	3,244 (33.3%)	273 (25.9%)	320 (43.1%)
M2	3,305 (33.9%)	278 (26.0%)	325 (43.8%)
M3	4,079 (41.9%)	380 (36.1%)	409 (55.1%)
M4	4,220 (43.3%)	394 (37.5%)	420 (56.7%)
M5	4,594 (47.1%)	440 (42.0%)	455 (61.4%)
M6 (Full)	4,822 (49.5%)	483 (45.9%)	481 (64.9%)
Perfect	9,747 (100%)	1,052 (100%)	742 (100%)

In results of test set 1, the current static solver performs similarly to the best 1997 solver. Adding search and adding region merging yield the biggest single improvements in performance, about 10% each. The heuristic evaluation function and weakly dependent regions add about 3% each. Other methods provide smaller gains with these long time limits, but they are essential for more realistic shorter times, as in the next experiment.

Results for test set 2 are a little bit worse than for test set 1, but that is true even for the baseline Benson algorithm. Our conclusion is that test set 2 is just a little bit harder, and the performance of the solver is comparable to its performance on test set 1.

5.2 Experiment 2: Detailed Comparison of Solvers

This experiment compares the six search-based methods M1–M6 in more detail on test set 1. The static solver can prove 321 out of 802 regions safe. Our best solver M6 can prove 548 regions with a time limit of 200 seconds per region. The remaining 254 regions have not been solved by any method.

A total of (548–321) = 227 regions can be proven safe by search. To further analyze the search improvements, we divide these regions into four groups of increasing difficulty, as estimated by the CPU time used.

Group 1, *very easy* (regions 322–346) – this group contains 25 regions. Most regions in this group have small size, less than 10. All methods M1–M6 solve all 25 regions quickly within a time limit of 0.1s (0.2s for M1).

Group 2, *easy* (regions 347–408) – this group contains 62 regions. Figure 6 shows two examples. Table 3 shows the number of regions solved by each method with different time limits. The number in braces is the difference between two methods. The performance of M1 and M2 is not convincing. By using region merging, M3 solves all 62 regions within 0.5s. The more optimized methods M4–M6 solve all within 0.1s. Region merging drastically improves the performance of solving these easy regions.

Group 3, *moderate* (regions 409–495) – this group contains 87 regions. Figure 7 shows two examples. Table 4 contains the test results. In this group, the search enhancements drastically improve the solver. M1 and M2 solve few problems. M3

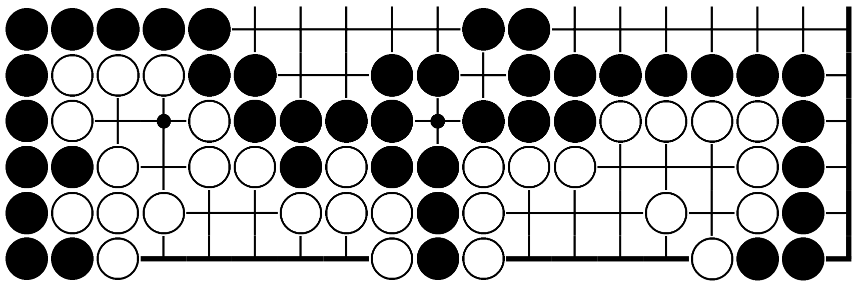


Fig. 6. Two examples of easy problems in group 2. Left: merged white region, size 10. Right: basic white region, size 11.

Table 3. Search results for Group 2, easy (62 regions)

Version	M1	M2	M3	M4	M5	M6
T=0.1s	0	23	38	**62**	**62**	**62**
T=0.5s	29 (+29)	31 (+8)	**62** (+24)			
T=1.0s	39 (+10)	40 (+9)				
T=5.0s	43 (+4)	42 (+2)				
T=10s	43 (+0)	44 (+2)				
T=50s	43 (+0)	49 (+5)				
T=200 seconds	43 (+0)	49 (+0)				
Solved	43	49	**62**	**62**	**62**	**62**

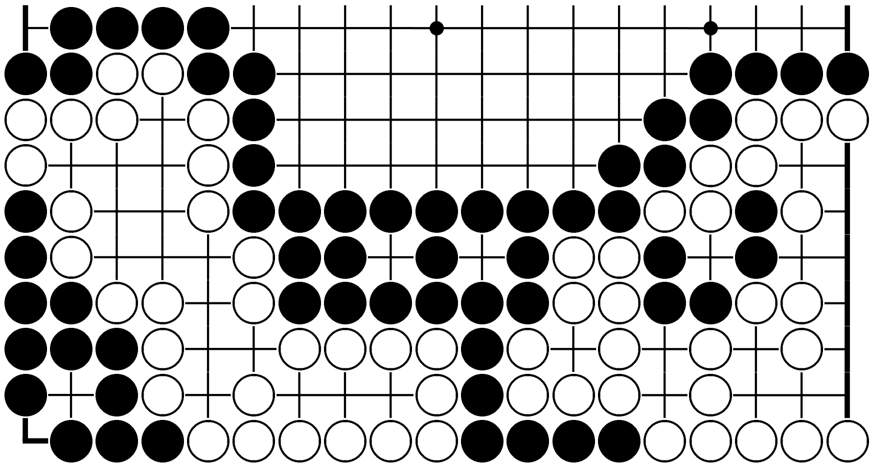


Fig. 7. Two examples of moderate problems in group 3. Left: merged white region, size 16. Right: basic white region, size 19. One white block has an external eye.

can solve 79 regions, but more than half of them need more than 10 seconds. The evaluation function drastically speeds up the solver. M5 solves all regions within 10 seconds. M6, using weakly dependent regions, solves 23 regions within 0.1s, as opposed to 0 for M5. All 87 regions are solved within 5s. In this category M6 outperforms all other methods.

Group 4, *hard* (regions 496–548) – this group contains the 53 regions that are solved in 5 to 200 seconds by M6. Figure 8 shows three examples. Table 5 contains the test results. This group includes 20 weakly dependent regions that cannot be solved by M1 to M5. Many of these problems take more than a minute even with M6. They represent the limits of our current solver.

Table 4. Search results for Group 3, moderate (87 regions)

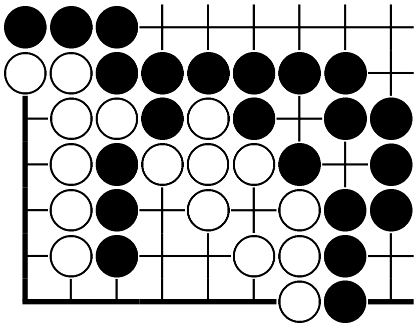
Version	M1	M2	M3	M4	M5	M6
T=0.1s	0	0	0	0	0	23
T=0.5s	0	0	14 (+14)	14 (+14)	10 (+10)	37 (+14)
T=1.0s	0	6 (+6)	33 (+19)	33 (+19)	38 (+28)	59 (+22)
T=5.0s	0	6 (+0)	38 (+5)	38 (+5)	68 (+30)	**87** (+28)
T=10s	0	8 (+2)	38 (+0)	40 (+2)	**87** (+19)	
T=50s	0	10 (+2)	73 (+35)	79 (+39)		
T=200 seconds	13 (+13)	17 (+7)	79 (+6)	82 (+3)		
Solved	13	17	79	82	**87**	**87**

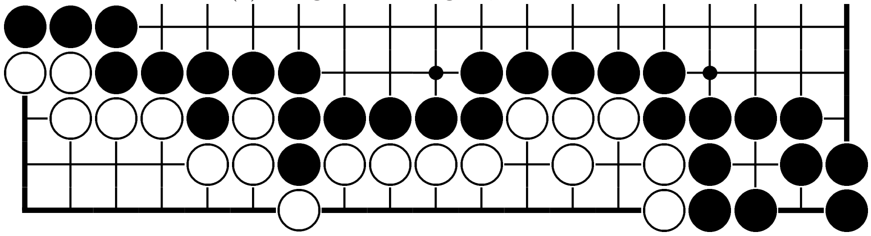
Table 5. Search results for Group 4, hard (53 regions)

Version	M1	M2	M3	M4	M5	M6
T=0.1s	0	0	0	0	0	0
T=0.5s	0	0	0	0	0	0
T=1.0s	0	0	0	0	0	0
T=5.0s	0	0	0	0	0	0
T=10s	0	0	0	0	11 (+11)	11 (+11)
T=100s	0	0	15 (+15)	17 (+17)	21 (+10)	28 (+17)
T=200 seconds	5 (+5)	5 (+5)	17 (+2)	20 (+3)	33 (+12)	**53** (+25)
Solved	5	5	17	20	33	**53**

6 Conclusions and Future Work

The results of our work on proving territories safe are very encouraging. Using a combination of new region-processing methods and search enhancements, our current safety solver is significantly faster and more powerful than the solver in [3]. However, most large areas with more than 18 empty points remain unsolvable due to the size of the search space. Figure 9 shows an example. Although this region has only 18 empty points, our current solver cannot solve it within 200 seconds and a 14-ply search. In order to handle larger areas, it can be improved in the following areas.


(a) Merged white region, size 17.


(b) Two weakly dependent white regions, size 11 and 9.

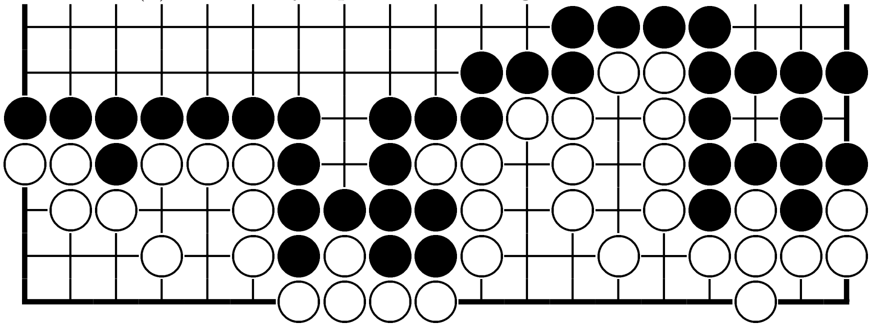

(c) Three weakly dependent white regions, size 13, 14 and 2.

Fig. 8. Three examples of hard problems in group 4

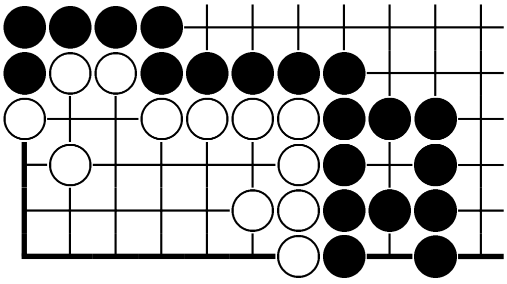


Fig. 9. Unsolved region, size 18

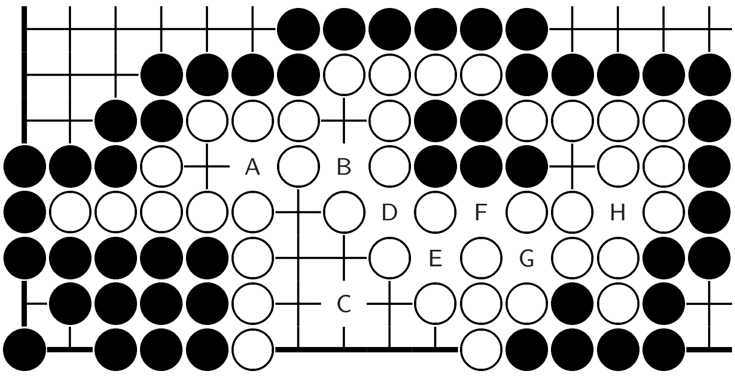


Fig. 10. An example of multiple related regions

Move generation – more Go knowledge could be used for safe forward pruning. Instead of generating all legal moves, in many cases the program could analyze the attacker's motivations and generate refutation moves. Move ordering and pruning for the attacker should also be investigated.

Evaluation function – our current exact evaluation function is all-or-nothing, and tries to decide the safety of the whole input area. If the area becomes partially safe during the search, this information is ignored. However, it would be very useful in order to simplify the further search. Also, more research on fine-tuning the evaluation function is needed.

Region processing – we can reduce the search space by treating a large region as several weakly dependent small sub-regions. Most sub-regions will be affected only by moves in the sub-region and possibly moves close to the boundary in other sub-regions. In addition, many strongly related regions could be treated as weakly related regions in practice. Figure 10, slightly simplified from position 16 of test set 1, shows an example. By our definition, regions $A \dots H$ are strongly related, and are merged into a single region of size 25. However, if the partition were $A \cup C$ and $B \cup D \cup E \cup F \cup G \cup H$ then each merged region would be small and could be solved. In practice, this happens very often, for example in 7 out of the 31 test positions in test set 1. Better methods are needed to analyze the relationships between regions and to process regions more selectively.

Search method – in place of $\alpha\beta$, a modern search algorithm such as DF-PN [4, 8] would probably work well in this domain.

More future work ideas include the following six topics.

- Handle special cases such as seki, snapback, double ko.
- Use the solver in EXPLORER to prove regions unsafe and find successful invasions, or defend against them.

- Compare the performance against heuristic Go programs in borderline cases where it is hard to judge statically whether a defensive move is necessary. Such a test would indicate how much the method can improve the playing strength of Go programs.
- Develop a heuristic version that can find possible weaknesses in large areas.
- Compare the performance with Life and Death solvers such as GoTools [9] in positions where the safety of territory problem is equivalent to a life-and-death problem.
- Build a solver for small board Go that utilizes this engine.

References

1. D.B. Benson. Life in the game of Go. *Information Sciences*, 10:17–29, 1976. Reprinted in Computer Games, Levy, D.N.L. (Editor), Vol. II, pp. 203-213, Springer Verlag, 1988.
2. M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006.
3. M. Müller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In H. Matsubara, editor, *Game Programming Workshop in Japan '97*, pages 80–86, Computer Shogi Association, Tokyo, Japan, 1997.
4. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, University of Tokyo, 2002.
5. X. Niu. Recognizing safe territories and stones in computer Go. Master's thesis, University of Alberta, 2004. in preparation.
6. E. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. Learning to score final positions in the game of Go. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10*, pages 143 – 158. Kluwer, 2004.
7. R. Vilà and T. Cazenave. When one eye is sufficient: a static classification. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10*, pages 109 – 124. Kluwer, 2004.
8. M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. An effective two-level proof-number search algorithm. *Theoretical Computer Science*, 313(3):511–525, 2004.
9. T. Wolf. The program GoTools and its computer-generated tsume go database. In H. Matsubara, editor, *Game Programming Workshop in Japan '94*, pages 84–96, Computer Shogi Association, Tokyo, Japan, 1994.