

An Open Boundary Safety-of-Territory Solver for the Game of Go

Xiaozhen Niu and Martin Müller

Department of Computing Science, University of Alberta, Edmonton, Canada, T6G 2E8
{xiaozhen, mmueller}@cs.ualberta.ca

Abstract. This paper presents *SAFETY SOLVER 2.0*, a safety-of-territory solver for the game of Go that can solve problems in areas with open boundaries. Previous work on assessing safety of territory has concentrated on regions that are completely surrounded by stones of one player. *SAFETY SOLVER 2.0* can identify open boundary problems under real game conditions, and generate moves for invading or defending such areas. Several search enhancements improve the solver’s performance. The experimental results demonstrate that the solver can find good moves in small to medium-size open boundary areas.

1 Introduction

Since the final score of the game of Go is determined by who can create more territory, estimating the safety of territories is a major component of a Go program. In previous work [8, 7], the search-based safety-of-territory solver *SAFETY SOLVER 1.0* was used to determine the safety status of a given region. However, the program operated under several restrictions. First, the region had to be completely enclosed. Second, the search was strictly local and did not utilize any external liberties of the boundary blocks.

In real games, most territories do not become fully enclosed until the late endgame. During most of the game, they have open boundaries. The surrounding conditions of a region, such as the number of external liberties of boundary blocks, are also very important. Ignoring them can lead to overly pessimistic safety estimates.

SAFETY SOLVER 2.0 is an improved safety-of-territory solver for open boundary areas. This program can generate moves for either invading or defending such areas. Additional searches provide answers to a series of related safety questions such as whether who plays first matters, whether the number of external liberties changes the local safety status, and whether changing the winner of ko fights affects safety.

The effectiveness of *SAFETY SOLVER 2.0* depends crucially on providing it with useful input. A heuristic board partitioning technique based on the concept of *zones* as implemented in the Go program Explorer [6] is used to identify open boundary areas in a full-board position.

1.1 Safety of an Open Boundary Area

In a game of Go, as more and more stones are played, the board is gradually partitioned into relatively small areas. These areas can be completely surrounded by stones of one

player, but more often they are not. Current Go programs use heuristic rules or pattern matching to generate moves to invade or defend open boundary areas. This approach is hit-and-miss, and chances to play better moves often exist. Figure 1 shows a typical example from our test set.

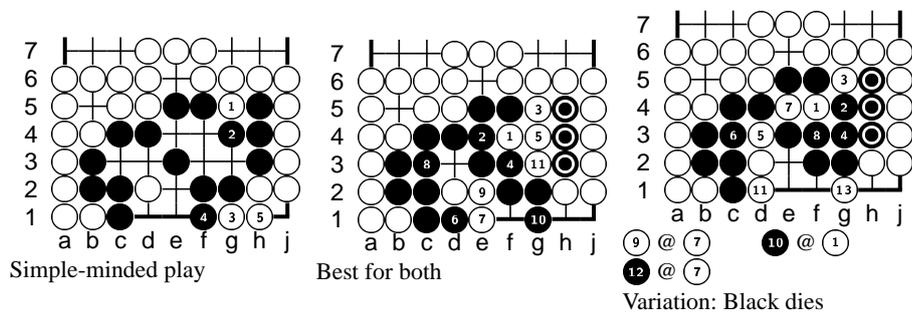


Fig. 1. An open boundary example

Assume that White, the attacker, plays first. In this position most Go programs will play the simple endgame moves shown on the left of Figure 1. These moves lead to a 10 point Black territory. Eventually, Black must add one more stone inside to prevent double-atari at **f4**. However, **f4** is a much better first move for White. The diagram in the middle of Figure 1 shows a best move sequence. After this sequence, in order to make the whole group alive Black has to give up the block marked by \odot . The rightmost diagram in Figure 1 demonstrates that if Black resists White **f4** with **g4**, the whole group dies. In this example although the size of Black open area is small (only 10), White's best attacking gains large benefit compared to the simple-minded play.

The safety status of an open boundary area is strongly related to its external context such as the number of external liberties of boundary blocks, possible outside connections, and the number and size of ko threats for each player. In the above example, if Black's boundary blocks had more external liberties, then the whole area would be safe territory. In many cases, the number of external ko threats also affects the local safety status. Therefore, an exact open boundary safety-of-territory solver needs to take all relevant context information into account.

The major difference between a life-and-death solver and a safety-of-territory solver is that a life-and-death solver does not consider territory. As long as a group is alive, the three alternatives of living with a large area, living with two small eyes, and forming a seki are all equally correct. In contrast, the goal of an open boundary safety-of-territory solver is to maximize (or minimize the opponent's) safe territory. *SAFETY SOLVER 2.0* achieves that in a flexible way. It can switch between different search goals, including a life-and-death solver. Section 2.2 describes these options in detail.

1.2 Related Work

Many approaches to territory evaluation in Go have been proposed in the literature. Static methods include Benson's algorithm for *unconditionally alive blocks* [1], Van der Werf's extension of Müller's static rules for safety under alternating play, and Vilà and Cazenave's static classification rules for regions up to a size of 7 points [11]. Search-based methods include Müller's local search for identifying the safety of regions by alternating play, and *SAFETY SOLVER 1.0* as described in [8, 7]. All these methods can handle only fully enclosed problems.

Several approaches for dealing with open boundary areas have been discussed or developed for life-and-death solvers. Solvers such as Kishimoto's tsume-Go solver [3] can deal with cases where the defender's boundary is open. However, the whole search region must be fully enclosed by safe attacker stones. Wolf [12] discusses practical issues about extending his program GoTools to open problems. Heuristic static eye-space analysis introduced by [2] can also handle open problems.

1.3 Contributions

The main contributions of this paper are:

- *SAFETY SOLVER 2.0*, a new improved solver for open boundary safety of territory problems.
- Re-search techniques for different search goals in the same position.
- Forward pruning techniques to safely reduce the search space and improve the performance of the solver.
- A heuristic method, using the Go program Explorer, to identify open boundary problems in real game positions.

The structure of this paper is as follows: Section 2 describes the structure and processing steps of the safety solver. Section 3 describes heuristic board partitioning for identifying open boundary problems. Section 4 introduces multiple searches for solving different goals. Section 5 introduces two forward pruning techniques. Section 6 provides experimental results. Conclusions and discussions about future work are given in the final section.

2 Open Boundary Safety-of-Territory Solver

2.1 Safety Solver

The safety solver described in this paper extends previous ones in [5, 8, 7]. It has been integrated into the Go program *Explorer* [5]. The solver uses the df-pn(r) search algorithm [4, 3] with domain-specific heuristic functions for initializing proof and disproof numbers. *SAFETY SOLVER 2.0* contains the following new features:

- Search goals customized by different parameters.
- Re-searches to provide solutions for different search goals.
- Preliminary integration into the Go program Explorer.

2.2 Input Parameters for the Search

Defining search goals for open boundary areas is more complex than for enclosed areas. In *SAFETY SOLVER 2.0*, the following input parameters specify a search goal for an open area. Figure 2 provides an example for two of these parameters.

- A set of points, called the area.
- The color of the defender and attacker.
- The first player. The search result is from this player’s point of view.
- Two different search goals: prove *boundary safe* or *territory safe*? For the boundary safe goal, all boundary blocks surrounding an area must be proven safe. This goal is equivalent to a life-and-death search. For the territory safe goal, all interior points of the area need to be proven safe as well. The attacker may neither live inside the area, nor reduce the area from the outside.

For example in Figure 2, White is the attacker and plays first at **a2**. If the goal is boundary safe, Black can play **c1** to make eyes and does not need to consider whether the territory is safe, as shown in the left of Figure 2. However if the goal is territory safe, then Black has to disconnect White by playing at **a3**. The move sequence on the right of Figure 2 indicates that eventually White has the choice to form a seki or kill all black blocks by ko. The Black zone is boundary safe but not territory safe.

- *Seki recognition*. *Seki* is a position where neither player can capture the opponent’s stones, so coexistence is the best result. When seki recognition is switched on, the solver distinguishes between the three results - win, loss and seki - by using the re-search techniques of [7]. Otherwise, the only possible outcomes are a win or a loss for the first player.
- Using external liberties. An *external liberty* of a boundary block lies outside the search area. Such liberties can affect the safety status of an area. For example, in Figure 2 the black block  has no external liberties and block  has one at **f1**. If either of them had one more external liberty, the area would be safe territory. When external liberties are not considered by the solver, it performs a quicker search to decide whether the area is locally safe. If external liberties are used, then the solver generates such external moves for the attacker and the defender as well, leading to a larger search space. Section 5.1 contains details on extended move generation for external liberties.
- The *ko winner* can also affect the safety status. For example, after the move sequence shown in the right of Figure 2, given enough ko threats White can capture all Black blocks. If Black is the ko winner, then Black can make the territory safe by winning a “thousand-year ko”.

SAFETY SOLVER 2.0 concentrates on proving whether an area is safe or not *locally*. It does not consider whether the defender’s boundary blocks can connect to other outside safe blocks from a full-board view. By default, the search goal of the solver is set to prove territory safe. To ensure that the local result is correct, the solver must handle seki and count external liberties. However no ko winner needs to be set initially, which relaxes the assumption in *SAFETY SOLVER 1.0* that the attacker wins all ko.

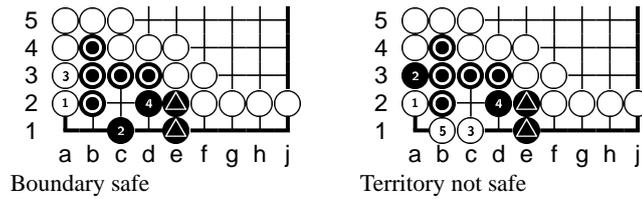


Fig. 2. Safety status changes under different goals

2.3 Integration with Explorer: First Steps

SAFETY SOLVER 1.0 as described in [8, 7] was implemented within the framework of the Go program Explorer but not called during game play. In *SAFETY SOLVER 2.0*, attacking and defending moves generated by the safety solver are passed to Explorer's global move generator. The solver uses the standard goal setting as described in Section 2.2. The heuristic value of these moves is set as follows:

- If the current search goal is boundary safe, then the best attacking or defending move is given the defender's minimum block value.
- If the search goal is territory safe, then the best attacking or defending move receives a value proportional to the number of interior points of the area.

3 Board Partitioning for Identifying Open Boundary Problems

The Go program Explorer uses heuristic territory evaluation to partition the board into *zones* [6]. On a board, empty points in gaps between stones of the same color can function as *dividers* or *potential dividers*. Dividers are small gaps that are sufficiently narrow so that the opponent can always be stopped from connecting through. Potential dividers are larger gaps that can be converted into dividers by a single move. Zones are computed by using dividers, potential dividers and heuristic knowledge. They are surrounded by stones, dividers and potential dividers. Figure 3 shows two Black zones. The boundary of the left zone consists of three stones and four dividers marked by A. The right zone's boundary contains six stones and two dividers marked by B. Both zones have 15 interior points.

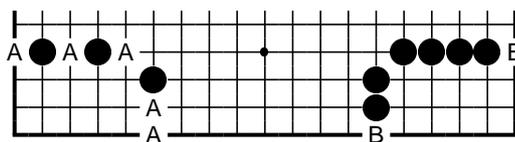


Fig. 3. Two examples of open boundary zones

Search areas for *SAFETY SOLVER 2.0* are computed by processing all zones. The first step selects candidate zones for search. Zones that are too large, with over 15 empty interior points are ignored since such large areas can rarely be solved in reasonable time by the current system. Zones that are too open are also not suitable. In the current implementation, if more than one third of a zone boundary consists of dividers, it is not searched. The left zone in Figure 3 is such an example. The solver will only search the right zone.

As in *SAFETY SOLVER 1.0* [8], *related zones* need to be merged. Two zones are related if they share one or more common boundary blocks. The merging algorithm in *SAFETY SOLVER 2.0* extends the one for fully enclosed zones by dealing with dividers. The left side of Figure 4 shows two related black zones. Their interior points are marked by A and B. *Inner dividers* which are adjacent only to related zones are added to the merged zone. In the example, **d3** is a divider between A and B and is added to the merged zone M shown on the right side of the figure.

As a final step, the opponent's connectible points are removed from the areas to avoid trivial attacker "wins". For example, in the right side of Figure 4, **h1** is originally part of the merged area M. However, the attacker can directly play there and is connected to the outside through **j1**.

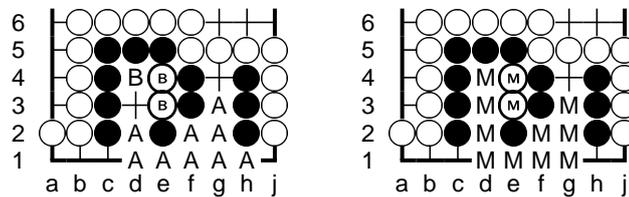


Fig. 4. Merging related zones

4 Multiple Searches for Related Goals

Since *df-pn(r)* only resolves boolean questions, the re-search technique described in [7] is used to distinguish between wins, seki, and losses. However in order to provide solutions for different search goals, further searches with different goal settings may be required. One important case is switching the first player. In a given board position, assume that Black plays first. The zone processing step discussed in the previous section provides the solver with a list of Black and White areas as inputs. For White areas, performing seki re-searches when Black plays first as the attacker can distinguish between a Black win, Black loss and a seki. However for Black zones, a second search with a different first player may be required. The detailed steps are as follows:

1. Set the first player to White and perform the first seki re-search to determine the outcome. If the result is a loss, White can not do anything in this area and the black area is safe. In this case further search is unnecessary.

2. If the outcome is win or seki, the zone is unsafe when White plays first. A second seki re-search is performed to check whether the zone is safe if Black plays first. If the second outcome is a win, Black can successfully defend the zone. If the second outcome is seki, then the zone is unsafe, and the best result for Black is to form a seki. If the outcome is a loss, the Black zone is not safe.

Further searches can be used to determine when external liberties affect the safety status of an area. Figure 5 on the left shows a Black bent-four area with no outside liberties. If White plays at **b1** the result is ko. Black playing first can defend the territory. The same example with three external liberties is shown on the right of Figure 5. If White tries **b1** again, Black can live with **B:a1, W:a2, B:c2** because of the external liberties. In this scenario the question is, if White starts taking external liberties such as **c3**, when does Black need to respond? To answer such questions, a search taking external liberties into account is required. The detailed steps, explained using the example from Figure 5 are as follows:



Fig. 5. External liberties affect zone's safety status

- Step 1: After **W: c3** it is Black's turn. The first search is local within Black's area and pessimistically assumes no external liberties. If the outcome is a win, the area is safe without using any external liberties. No further search is necessary.
- If the outcome is seki or loss for Black, then a second search using the real external liberties is performed. In the left of Figure 5, the first search returns a loss. The second search establishes that with two external liberties the Black area is safe. In the future, after White plays **b4** or **d3**, another search will show that Black needs to defend the territory to avoid ko.

5 Forward Pruning Techniques

For an enclosed area, all legal moves inside the area are generated for both players. In an open boundary area, many external moves must also be generated. This leads to a larger search space, with the increase depending on how many external moves have been added. In order to improve the performance, two forward pruning techniques for defender moves are used in *SAFETY SOLVER 2.0*.

5.1 External Moves

The first technique is used for generating external moves. Figure 6 shows an open boundary example. First, the program generates all legal moves on interior points. In this example, all 12 interior points will be generated for either attacker or defender.

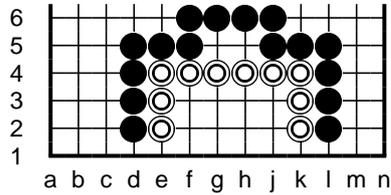


Fig. 6. external move generation for an open boundary area

Second, the generator needs to identify all external liberties for all boundary blocks of the zone. In this example, the white boundary block has 4 external liberties at **e1**, **g5**, **h5**, **k1**. Among them **e1** and **k1** are dividers of the area, and **g5**, **h5** are pure external liberties not related to dividers. Attacker plays on these external liberties can affect the safety of the area. However, defender plays on these liberties can be pruned since they can only decrease the liberties of defender blocks, and can neither be used to form an eye nor to create a seki. *SAFETY SOLVER 2.0* focuses on proving the local safety status, and ignores possible outside connections or counterattacks on outside attacker stones. Therefore pure external liberties moves such as **g5**, **h5** can be pruned for the defender.

Third, the program generates moves around dividers. In this example, if White plays **d1** Black can block by playing at **c1**. Moves such as **d1**, **l1** are called *layer one moves* and moves **c1**, **m1** are called *layer two moves*. The defender must generate layer one moves but not layer two moves because playing directly in layer two moves will not help to prove the inside area safety (the connection problem is ignored). Therefore layer two moves are pruned for the defender. For the attacker, moves in both layers around dividers will be generated. In this example, the program generates 16 moves for the defender. For the attacker, since layer one and dividers are all empty points, layer two moves are pruned, giving 18 attacker moves including **g5** and **h5**.

5.2 Inner Eyes

During search, the defender might be able to form eyes inside the area. In this paper, only one-point eyes inside the area are called *inner eyes*. Inner eyes are very helpful for the evaluation function to evaluate the safety status. Since the defender should not fill them, such defender moves are pruned. For example, in Figure 7 the move **f3** can be pruned for the defender. Since the attacker playing at **f3** is illegal, **f3** will also be pruned for the attacker. In this example, five moves are generated for the attacker (**g5**, **h1**, **h5**, **j1**, **j3**) and only three for the defender (**h1**, **j1**, **j3**).

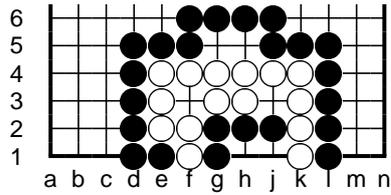


Fig. 7. Pruning inner eyes for the defender

6 Experimental Results

Three test sets were created for three experiments. Set 1 contains a total of 120 test positions, including 60 original open boundary problems and 60 modified problems that are versions of the original problems with some external liberties added. Most original positions are taken from the classic *Guan Zi Pu* [9]. The remaining problems are collected from several resources, including [10] and positions created by the authors. Set 2 contains 20 positions from computer-computer games. Since *SAFETY SOLVER 1.0* can not handle open boundary problems, it can not solve any positions from Set 1 and 2. Set 3 contains 100 final positions of 19×19 games played by amateur and professional players. All three test sets are available at: <http://games.cs.ualberta.ca/go/open/>.

Experiments were performed on a Pentium IV/2.4GHz with 1024 Mb memory. The time limit is 200 seconds per position in experiment one and two, and 10 seconds per local area in experiment three.

6.1 Experiment One: Correctness Test

The purpose of this experiment is to test the correctness of the solver. The difficulty of positions varies from easy to hard. The standard search setup described in the end of Section 2.2 is used for all these positions. The solver correctly proves that the territories of all 60 original problems are not safe, and generates White's best invading or reducing move. In the 60 modified problems, the solver proves that the territories are safe due to the external liberties added.

Figure 8 shows four Black open boundary positions from the 60 original problems in Test set 1. White as the attacker plays first.

The two black zones on the top are easy problems. On the left, after White's best invading move at **a18**, Black can choose to either let White connect back through **a17**, or form a seki as shown in the Figure. Either way the territory is not safe. The solver finds the best invading move **a18** in 0.01 seconds. On the right, the solver finds the best move **r19** in 0.1 seconds.

The problem at the bottom left corner is moderately difficult. It takes 51 seconds for the solver to find the best invading move **b2**. White can form a seki inside Black's zone. The problem at the bottom right corner is hard. The solver finds the best invading move **r2** in 179 seconds. Figure 8 shows strongest move sequences for each problem.

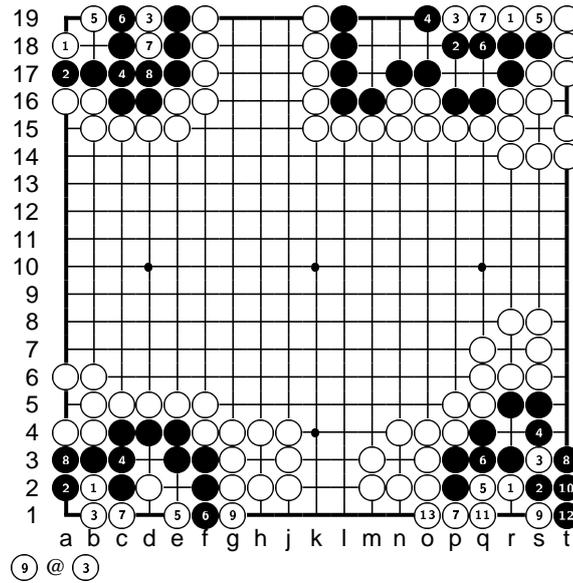


Fig. 8. Four open positions in Test set 1

6.2 Experiment Two: Game Play Test

The purpose of this test set is to test whether Explorer enhanced by the safety solver is able to play the correct defending or invading move in 20 real game positions. Before integrating the safety solver, Explorer failed all these test cases. The standard setup described at the end of Section 2.2 is used. The time limit is 200 seconds per position. The results show that current Explorer correctly identifies the problems and generates best first moves for all of them.

Figure 9 shows an example in Test set 2. Before using the safety solver, Explorer as Black played **k1** to capture three white stones marked by \odot . White played **d2** to capture one black stone and finished the play in this local area with a safe territory of 9 points. After integrating the safety solver, Explorer finds the move **d1**, which leads to a seki in the corner. Black is still able to capture the three White stones \odot .

6.3 Experiment Three: Comparison of Solvers

This experiment compares the performance of four solvers Benson, Static, *SAFETY SOLVER 1.0* and *SAFETY SOLVER 2.0* on 100 completed games. The proven safe points are computed by each program starting from the end of the games, then backwards every 10 moves. Figure 10 shows the average number of proven safe points by solvers. At the end of the game, *SAFETY SOLVER 2.0* proves 202 points on average, while *SAFETY SOLVER 1.0*, Static and Benson prove 137, 85 and 46 points respectively. *SAFETY SOLVER 2.0* outperforms *SAFETY SOLVER 1.0* by a large margin, proving 47% more points. More importantly, the solver is useful much earlier in the

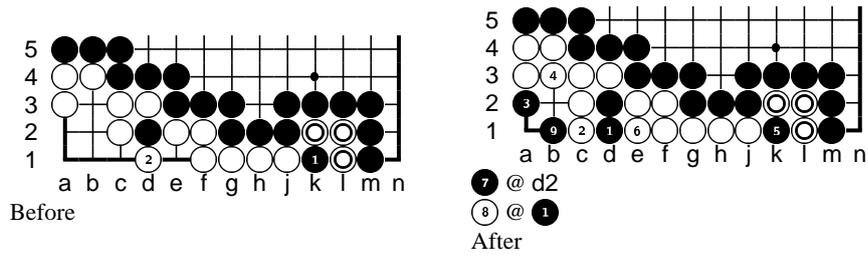


Fig. 9. An example from Test set 2

game. Even 80 moves before the end of games, *SAFETY SOLVER 2.0* can prove 42 points safe on average, while the other solvers can hardly prove any.

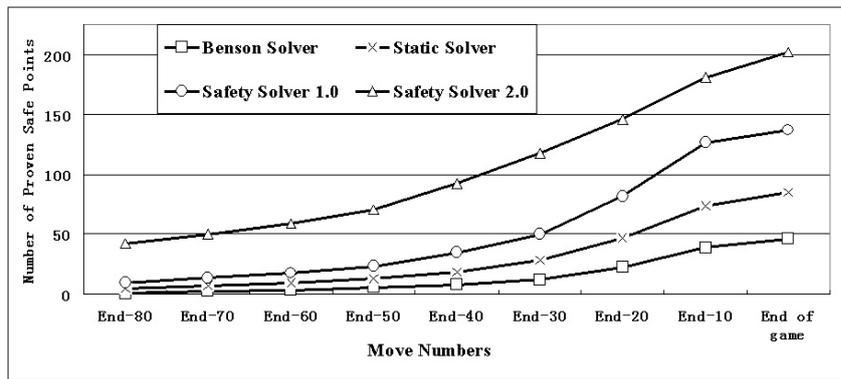


Fig. 10. Comparison of four solvers on 100 games

7 Conclusions and Future Work

SAFETY SOLVER 2.0 is an improved safety solver that provides exact evaluations for the safety status of open boundary areas. By using re-search techniques, it can compute different search goals in the same position. In addition, it has been used in Go program Explorer to identify and solve open boundary problems in real game positions. Although experimental results are very encouraging, there are still numerous ideas on how to improve the performance.

One major limitation of the current safety solver is the size of an open area. The search space for an open area is often much larger than for an enclosed area, depending on the number of dividers and external liberties. An open area with 15 interior points can easily have over 20 legal moves for each player. In practice the current size limit

for the solver is around 15 interior points. Another problem is time control. Under tournament time conditions, it is not feasible to spend too much time on one local area. In addition, searching every unsafe zone using the same amount of time is also not wise. Concentrating the effort on areas where good invading or defending moves are most likely to exist would greatly improve the effectiveness of the solver in game play. Developing a flexible time control scheme that uses heuristics to select suitable problems to solve is an interesting topic for future research.

Further ideas for improvements include:

- Store a subtree of search results of local problems in a hash table, then look up solutions when they are needed.
- Use the most promising move from the search as the best try, even if a search runs out of time.
- Integrate other tactical solvers into the safety solver. For example, if an open area has been proved as locally unsafe, it should be checked whether its boundary blocks can be connected to other outside blocks.

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, New York 1988.
2. K. Chen and Z. Chen. Static analysis of life and death in the game of Go. *Information Science*, 121:113–134, 1999.
3. A. Kishimoto. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. PhD thesis, Department of Computing Science, University of Alberta, 2005.
4. A. Kishimoto and M. Müller. Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games. Many Games, Many Challenges*, pages 125–141. Kluwer Academic Publishers, 2003.
5. 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.
6. M. Müller. Counting the score: Position evaluation in computer Go. *ICGA*, 25(4):219–228, 2002.
7. X. Niu, A. Kishimoto, and M. Müller. Recognizing seki in computer Go. In *proceedings of the 11th Advances in Computer Games*, 2005. To appear in Springer Verlag Lecture Notes in Computer Science (LNCS).
8. X. Niu and M. Müller. An improved safety solver for computer Go. In J. van den Herik, Y. Björnsson, and N. Netanyahu, editors, *Computers and Games: 4th International Conference, CG 2004. LNCS 3846*, pages 97–112, Ramat-Gan, Israel, 2006. Springer.
9. S.Y. Tao. *Guan Zi Pu*. 1689. Reprinted in Wei Qi Ji Qiao Da Quan, M. Jiang, Z. Jiang (Editors), Shu Rong Qi Yi Press, Cheng Du, China, 1996.
10. E. van der Werf. *AI techniques for the game of Go*. PhD thesis, University of Maastricht, 2005.
11. R. Vilà and T. Cazenave. When one eye is sufficient: a static classification. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 109 – 124. Kluwer, 2004.
12. T. Wolf. About problems in generalizing a tsumego program to open positions. In *proceedings of the Game Programming Workshop in Hakone/Japan*, pages 20–26, 1996.