Using SAT-Solving Techniques in the Game of Go

Seth Cooper and **Kevin Wampler**

Abstract

With the success of computer algorithms to play games such as chess and checkers, the ancient game of go has stood out more and more prominently as the last remaining frontier for artificial intelligence in deterministic zero-sum complete knowledge games.

We present a method by which goals in a game of go can used to prune the moves which are considered to those which may lead toward the realization of a goal. We formulate these goals in a form similar to prepositional logic, which allows us to use a DPLL style solver for a minimax search algorithm, as well as to potentially create heuristics for the search based on an analysis of the formula.

Introduction

Current AI techniques have met with little success when applied to the game of go. The best computer players still achieve only a novice level of play, and in large part resemble expert systems designed to play the game, rather than being able to play well from little more than the rules or learn good play from data.

This stands in sharp contrast to other deterministic zero-sum complete knowledge games such as Othello, checkers and chess. In each of these games a respectable level of play can be achieved by combining a simple evaluation function with an alpha-beta minimax search. In the case of chess, the most difficult of these games for computers to perform well at, even this simple technique is sufficient to look 10 moves ahead, which is significantly more than an average human player.

These techniques have had far less success with playing go. The most basic reasons for this are twofold, presenting problems both for performing a minimax search and for creating an evaluation function.

In contrast to Othello, checkers, and chess, all of which are played on an 8×8 board, a standard game of go takes place on a 19×19 board (although the smaller sizes of 13×13 and 9×9 are sometimes used for beginners). During the game, pieces are allowed to be played at almost any unoccupied intersection. This results in the average branching factor for a game of go being around 200, as opposed to about 35 for chess. Even in the case of a 9×9 board the branching factor still greatly exceeds that of chess. This problem is made even worse by the ability of human players to reliably read ahead on the order or 60 plies. Since the stones do not move around, even beginners can reason roughly about the effects that a play may have on board states well over a hundred plies later.

Even if a computer program could read deeply in a game of go, the evaluation function poses tremendous difficulties. Since the pieces in games such as Othello, checkers, or chess operate relatively independently, it is easy to create a simple evaluation heuristic which gives a reasonable analysis of which player is ahead at a given point.

The situation in go is anything but this simple. The goal in go is to control territory, which is a task that by its very definition involves many pieces. Furthermore, pieces in go, even early in play before any territory is solidly staked out, rarely act independently. Rather pieces are either directly connected (and thus inseparable from) a larger group of stones, or are only meaningful within the context of other stones on the board. The intricacies of these interactions are subtle enough to make the creation of even a remotely reasonable evaluation function a daunting task. The creation of an evaluation function suitable for high level play remains a problem with little prospect for solution in the near future.

In this paper we will not aim to solve all of these problems, but rather show a method by which the branching factor can be reduced by considering only moves which may work toward the attainment of some goal. The goals are further formulated in a manner which is general enough to express a very wide variety of possible strategies, rather than being limited to the expression of some small set of higher level actions.

Using SAT Techniques

We guide the search for an optimal move by formulating the problem in a form in which it possible to apply some of the techniques that are used in satisfiability solvers. The basic idea used is that only certain board configurations are useful in achieving a goal, so we can create a logical formula describing these configurations. Finding an optimal move can then be phrased in terms of the satisfiability of this formula and the search restricted to moves which will aid in this. It is worth mentioning that although many satisfiability techniques are applicable to this formulation, our method does not result in a standard satisfiability problem, and still more closely resembles minimax (or a restricted form of QBF).

Motivation

A blind search in the game of go only allows a very shallow search. In our implementation, we found that even on a 9×9 board we could only search up to 3 ply in a reasonable time. At a depth of 3 ply a blind search has to consider over 511,000 moves, and at 4 ply this increases to nearly 40 million moves. On a full 19×19 board the number of moves which must be considered at 3 and 4 ply are over 45 million and over 16 billion respectively.

This level of search is far, far to low to be useful, even with a good evaluation heuristic (which, of course, is itself not known). The principal problem with the blind search algorithm is the very fact that it is blind. Such an algorithm is attempting to find tactical solutions by brute force, and has no other means at its disposal. This method fails as miserably when applied by programs to go as it does when applied by humans (this is basically the strategy that many people have their first time playing the game).

This hints at a deeper relationship between tactics and large scale, longer term strategies in the game of go. This relationship is well stated in a popular book on go problems (Davies 1995):

> The first principle in reading is to start with a definite purpose. There is no better way to waste time than to say to yourself, 'I wonder what happens if I play here,' and start tracing out sequences aimlessly. Tactics must serve strategy. Start by asking yourself what you would like to accomplish in the position in question, then start hunting for the sequence that accomplishes it. Once you have your goal clearly in mind the right move, if it exists, will be much easier to find.

It is precisely this view of tactics as serving strategy that we take to better guide our statical search.

Overview

In order to use strategy to guide the search for a tactical solution, it is first necessary to have a way to describe a strategic goal, representing what we would like to accomplish in a given situation. To the best of the authors' knowledge, attempts to achieve this have only been done in the context of adversarial planning, where goals are formulated in terms of higher level concepts, such as 'connect groups A and B' or 'capture group A' (Willmott *et al.* 1999). This sort of representation for goals by its very nature greatly limits the types of goals that can be represented. If one wishes to be able to have a new king of goal, say 'build a wall' then the definition of this goal must be programmed by hand. Furthermore, it may be difficult to learn such specific goals from data of actual play between humans, so there must be essentially an expert system to provide these goals.

Although we do not actually provide a system to learn these goals from data, we do focus on a formulation of goals which is flexible enough to cover many diverse circumstances, and for which it at least looks hopeful to be able to learn goals from example games.

We think of a goal as a constraint on a future board state. For example, if we wish to capture a piece then we would formulate a goal of having a future board state in which that piece is not on the board. This view captures all of the essence of what a goal is, namely it is some set of constraints on the state of the board that we would like to be able, through correct play, to bring the game into agreement with. The sorts of goals which we are able to represent is then merely determined by the expressiveness of how we express hypothetical future board states.

Because of its simplicity and relatively powerful expressiveness, as well as its closeness to the actual game of go, we choose a language very similar to prepositional logic to formulate these goals in. Each square on the board can have one of three states, either it can be black, white, or empty. These states are similar to the values that true, false, and unassigned might have in a standard satisfiability equation, but it is useful to retain the notions of black and white since, for example, only one player can place a piece of a given color. At a square at coordinates x, y, we will represent the constraints that black or white occupy that square as B_{xy} and W_{xy} respectively.

A goal is a logical formula involving these sorts of variables. We express these logical formula in disjunctive normal form, because of its usefulness in specifying actual goals in the game (note that since this is more closely related to QBF than to standard satisfiability, we cannot solve a DNF equation in polynomial time). For example, if we want a white piece at $E5$ to be captured or a white piece to be at $E4$ and a black piece at $F5$, we would create the goal $(\neg W_{E5}) \vee (W_{E4} \vee B_{F5})$.

Merely having these goals is not, however, sufficient. We must also be able to use them to find, if it exists, play which will lead to the attainment of the goal. We do this in two steps. First we create an expression denoting the ways in which it is possible to achieve the given goal within some set limit of moves, and then we attempt to find an adversarial satisfiability solution to the resulting expression which will give the correct play (or determine that none exists). In order to simplify these parts, we we consider only a restricted subset of go in which no captures (other than the one specified in the goal itself) are allowed. This is a major restriction on the rules, but many of the routes of play in this form are identical to in actual go. In any case, we suspect that, since the assumption of no captures in go holds for most of the moves in a game, the techniques described for this restricted rule set will be applicable in largely their present form to the full rules of go.

Generating Pruning Expressions

The first step in this algorithm, once a goal has been supplied and we wish to determine if it is actually realizable, is to generate an expression which gives the conditions under which the goal can be satisfied. This serves to massively prune the spaces which must be searched, and so we refer to such an expression as a *pruning expression* Since there may be arbitrarily complex and deep ways in which any particular goal can be satisfied, we set a limit $maxDepth$ on the number of plies we are willing to consider in a solution.

To aid in the discussion of this algorithm, we will define the following terms:

string A group of stones of the same color where each piece is connected to the others by being adjacent either horizontally or vertically another piece in the group.

liberty An empty square which is either horizontally or vertically adjacent to a string.

capture When all of the liberties of a string are filled with pieces of the opposite color, it is removed from the board.

depth In this algorithm, the depth of a solution will refer to the number of moves which the *opponent* makes before a goal is achieved. Thus the ply of a solution is $2depth + 1$.

We will also adopt the convention that black is always the player attempting to achieve a goal and white is always the player attempting to prevent that goal from being realized.

Since blindly searching over the entire board is prohibitively slow, we will prune the configurations which are considered to those which have a potential of leading to a realization of the goal within a depth less than or equal to $maxDepth$.

This is achieved through a process similar to iterative deepening. For each depth (starting at depth 0), we first derive which conditions would need to obtain for black to accomplish the goal, assuming white is not allowed to make any further moves. This essentially gives the conditions that need to obtain for black to immediately achieve its goal.

At each iteration, once black has created an expression describing the conditions that would need to obtain for its goal to be achieved in some depth, d, white then attempts to find a move which will push the depth required to achieve the goal to $d + 1$.

Once white has derived a way (if there is one) to push this depth back to $d + 1$, we branch on all of the possible ways white could thwart the goal at depth d and iterate the process on each of these branches for depth $d + 1$.

To further prune the board configurations which we consider, any ways of obtaining the goal which require too many moves to be accomplished in depth $maxDepth - d$ are not considered. For each depth d this amounts to pruning those configurations which require more than $maxDepth - d + 1$ moves by black or $maxDepth - d$ moves by white.

During this process, we keep track of all configurations which we have encountered by which the goal could be obtained. These configurations are stored as a conjunction of the states of the squares on the board (black, white, or empty). If any one of these configurations actually obtains, black's goal will be accomplished, so we create a single disjunction of all of these configurations. Thus the ways of accomplishing a goal can be stored as a expression in disjunctive normal form.

An example of how this process proceeds will be illustrative of what is going on. Consider the board shown in figure 1.

Suppose that black formulates as a goal the capture of the white piece at E5. This goal would be stated as the following

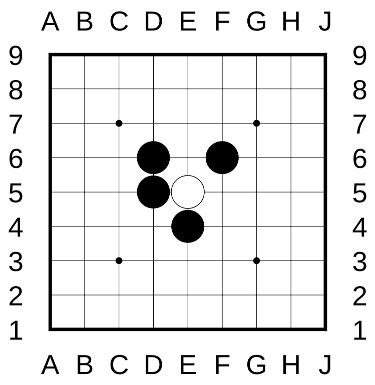

Figure 1: black to capture white

constraint:

$$\neg W_{E5} \tag{1}$$

We will consider the steps taken by our program in creating a pruning expression to satisfy this goal with $maxDepth = 1$.

Since there is currently a white piece at E5, the only way for black to achieve this goal in depth 0 (white makes no moves) is to surround it by filling in its liberties as shown in figure 2.

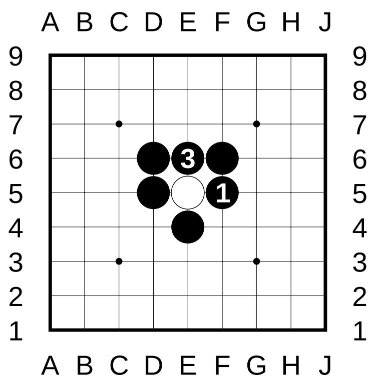

Figure 2: an immediate capture

The expression corresponding to this capture is:

$$B_{E6} \wedge B_{F5} \tag{2}$$

[1]

If white is to avoid being captured in depth 0, it must invalidate this expression. Since it is taken to be an axiom that a white piece cannot be on the same square as a black piece ($\neg(B_{xy} \wedge W_{xy})$), and since we are not considering captures by white, the only way for white to do this is to have either W_{F5} or W_{E6}.

[1]The actual expression is $B_{E6} \wedge B_{F5} \wedge B_{D5} \wedge B_{E4}$, but for the sake of brevity we will omit the variables corresponding to pieces which are already on the board in this initial state.

Our algorithm recurses on each of these terms. Let us consider W_{F5} first. The board state ofter this move is illustrated in figure 3

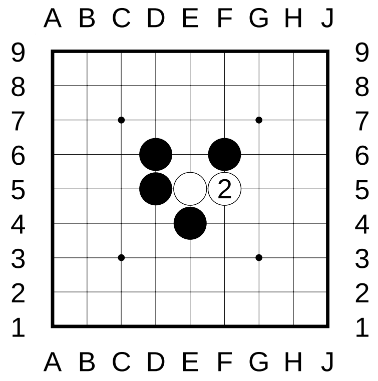

Figure 3: white escapes

If black wants to capture white in this configuration, as illustrated in figure 4, then black must be able to satisfy the expression:

$$W_{F5} \wedge B_{E6} \wedge B_{G5} \wedge B_{F4} \tag{3}$$

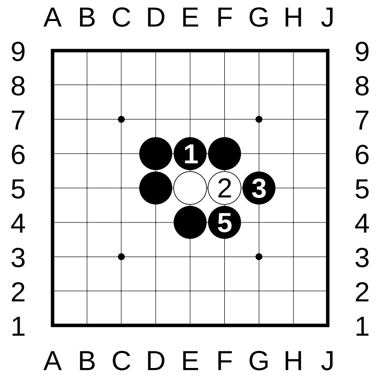

Figure 4: requirement to capture in depth 1

The number of variables remaining to be set in this equation (equivalently the number of moves required to capture) exceeds the limit set by $maxDepth$, so this state can be removed from consideration.

At this point we have exhausted this branch and can now consider white's other option, W_{E6}, as illustrated in figure 5.

In this case, to realize the goal of $\neg W_{E5}$ black only has to satisfy the formula:

$$W_{E6} \wedge B_{F5} \wedge B_{E7} \tag{4}$$

This capture is shown in figure 6.

Since the number of unset variables in this equation is only two (B_{F5} and B_{E7}), it is attainable within $maxDepth$.

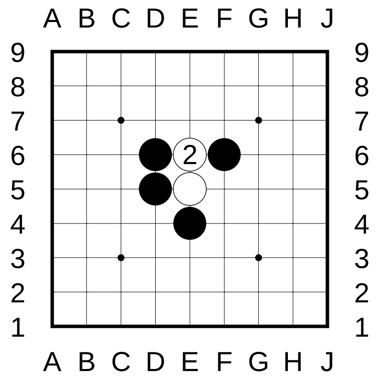

Figure 5: white escapes depth 0 capture

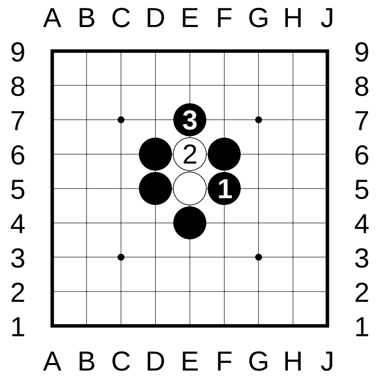

Figure 6: black captures

This exhausts all possibilities for attaining the goal within depth 1, so the final expression indicating the possible ways in which black can achieve this goal is returned as the disjunction of each of the configurations in which it could be attained:

$$(B_{E6} \wedge B_{F5}) \vee (W_{E6} \wedge B_{F5} \wedge B_{E7}) \tag{5}$$

This expression is then given to an algorithm which attempts to find a sequence of moves by the players in which black can force it to be satisfied. In this case, it can be seen the by black setting B_{F5} this can be done[2]. The algorithm for determining this is described next.

Solving Pruning Expressions

Our algorithm for solving for satisfiability in go pruning expressions derives from the DPLL satisfiability algorithm and the minimax adversarial search algorithm (Russell & Norvig 2003). We are aware of some concurrent research in applying adversarial search techniques to satisfiability problems (Zhao & Mueller 2004); however, it is recent enough that

[2]It is worth mentioning that at 3 ply this example is essentially the most difficult sort of problem that a blind minimax algorithm could solve.

we have not been able to find the paper, and it has not been applied to the game of go.

Our algorithm proceeds as follows. There are two players, analogous to MAX and MIN in minimax; let us call them TRUE and FALSE for clarity's sake. The players are given a pruning expression, and alternate turns assigning values to variables in the expression by making moves. A move sets a variable to true if it has the the same space and color as the variable. A move sets a variable to false if has the same space and a different color as the variable. TRUE attempts to have the expression evaluate true, and FALSE attempts to have the expression evaluate false.

We had initially planned to give the algorithm two expressions: one which TRUE would try to set true and FALSE would try to set false, and one which FALSE would try to set to true. In this way we could constrain the moves TRUE made by limiting them only to those which did not set the second expression to true. As it turned out, for our restricted rule set we only needed the first expression. As the players take turns, the pruning expression is updated to reflect the new state of the board.

In the computation we update by using DPLL on our expressions, which are in DNF. If a variable is set to true, it is removed from its clause; if it is set to false, its whole clause is removed from the expression. This continues until all clauses are removed, in which case the expression has become false, or there is an empty clause, in which case the expression is true. In this way, the available moves for each player are encoded in the expression as the turns progress. In our implementation, we are able to prune the moves TRUE considers to only moves which correspond to variables of if its color. FALSE will consider all moves corresponding to variables. Moves are considered in an arbitrary order; we do not employ any heuristics to guide their expansion. It is very likely that heuristics, such as looking for pure symbols, would improve the performance of our algorithm.

Results

Because of their particular use in simply illustrating the sorts of problems which out method can solve, we will focus primarily on problems in which, through correct play, black can capture a white piece. One simple example of such a result was given above in the description of the generation of pruning expressions. Another standard problem involving a capture is a *net*. The most simple form of a net is shown in figure 7.

The goal for this capture is stated as $\neg W_{E5}$.

The correct solution, as found by our program is shown in figure 8.

Once black has moved here, white cannot escape and is eventually doomed.

Two slightly more difficult net examples are illustrated in figures 10 and 12, and the solutions found by our program are shown in figures 11 and 13.

A substantially less easy to read solution to a capturing problem found by our program is shown in figure 14.

The solution found by our program is somewhat surprising (particularly since we did not realize there was a solution), and is shown in figure 15

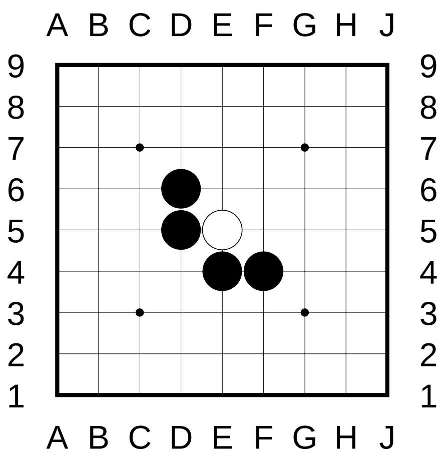

Figure 7: black can capture white in a net

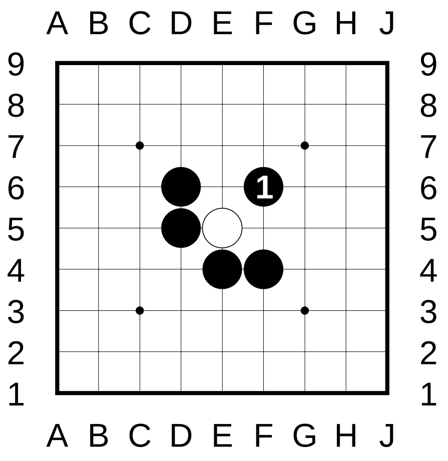

Figure 8: the correct move

This solution does indeed work, if white attempts to jump out at $E5$, our program caps at $E6$ and the white pieces have no escape. See figures 16 and 17.

Since we represent goals in a language similar to prepositional logic, we are not limited to goals which simply involve the capture of a single group of pieces. This is a very useful trait as far as actual play in the game of go is concerned. Even if we still restrict ourselves to goals involving captures, it is not too uncommon to see cases where either one of two white groups can be captured, but in which black cannot choose which one. The most simple example of such a case is when black can cause a *double atari*. Such a situation is shown in figure 18.

In this case we form a goal as any board state in which either one of the two white pieces has been captured:

$$\neg W_{E6} \vee \neg W_{F5} \tag{6}$$

Our program was easily able to find the solution to this, after black moves at $F6$ white can save either piece, but not both (see figures 19 and 20).

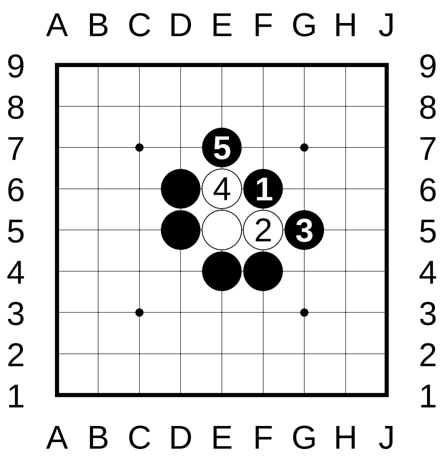

Figure 9: white attempts to escape, but cannot

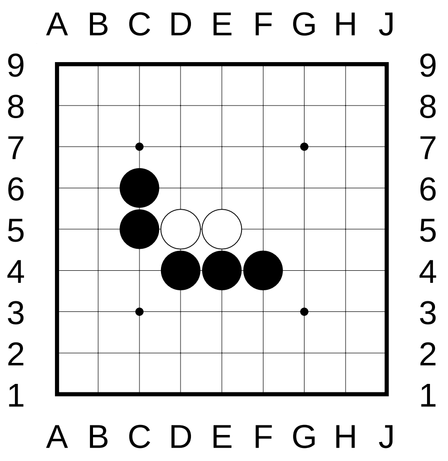

Figure 10: a slightly harder net

As a final example of the ability to solve goals other than captures, we consider the situation shown in figure 21.

The white piece at $D5$ is safe for the time being, but black can do well by threatening a capture in order to build a wall along the top. This goal can be expressed as the disjunction of a state representing the capture, and another representing the wall.

$$\neg W_{D5} \vee (B_{D6} \wedge B_{E6} \wedge B_{F6}) \tag{7}$$

White cannot prevent this goal from being attained once black moves at $D6$, though two possible attempts to do so are shown in figures 22 and 23.

All of the problems given here were not only solved by our system, but the solution was always calculated in a fraction of a second. We found that our program was able to quickly solve problems with a depth of up to 9 ply, taking about the same amount of time as a 2 ply blind minimax search.

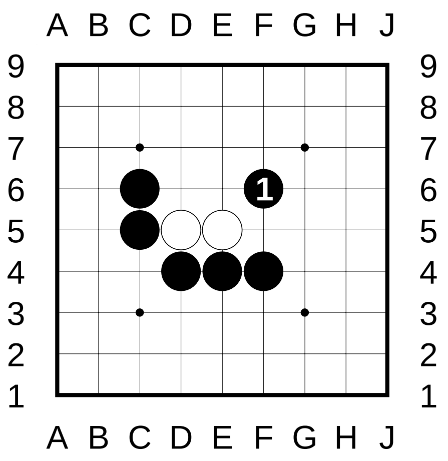

Figure 11: correct play

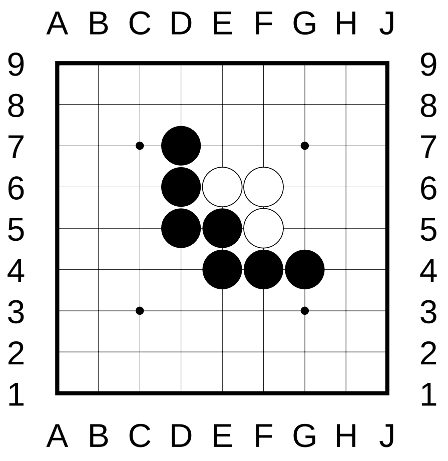

Figure 12: another net example

Conclusion and Future Work

We have presented a method whereby goals in a game of go can be specified and used to guide a tactical search. In the examples we tested, we were able to increase the search depth that can be done in under a second from 2 ply to 9 ply. Furthermore we specify these goals in a simple, general, and flexible form. Although our method is currently restricted to a subset of the rules of go, we suspect that the ideas shown can be applied without too much modification to the full rule set.

There is a great deal of room for future work on this method. It of course would be necessary to extend these methods to the full rule set of go. In addition, goals must currently be supplied by the user. It would be very desirable to create a system which could propose goals for a given board state by learning from examples of games played by humans. Toward this end, it would also be useful to be able to specify goals in a probabilistic manner, so that a goal represents a soft constraint on a future board state rather than

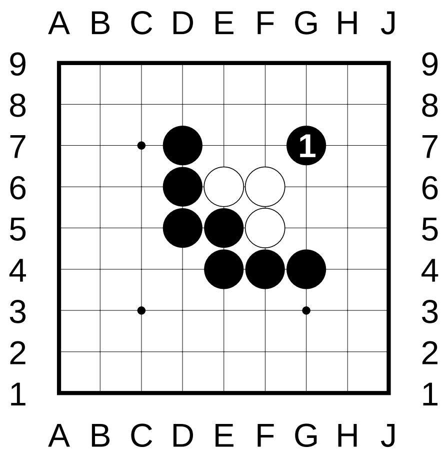

Figure 13: black captures

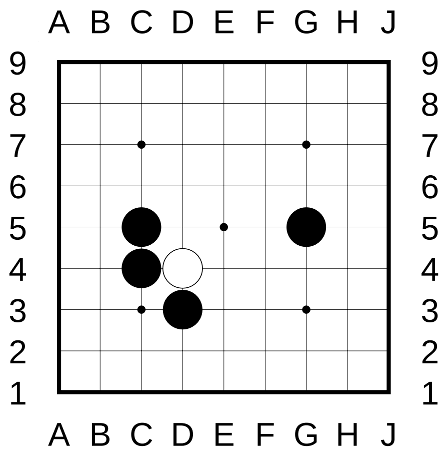

Figure 14: can black capture the white piece?

a hard constraint. Done correctly, this would allow goals to be only partially achieved, and the utility of how well a goal is achieved compared to others that could be achieved to be expressed.

The current bottleneck in the execution of our program is in solving the pruning expressions. We use a very simple algorithm based on DPLL, but methods derived from other more advanced SAT solving procedures, or tailored to work well in go, should be investigated. It may be particularly fruitful to consider stochastic methods, such as walksat, since even in experienced human play it is often the case the all possible moves which could alter the outcome are not considered, but only those which seem to have the best chance of achieving the goal.

References

[1] Davies, J. 1995. *Tesuji, Elementary Go Series, Volume 3*. Kiseido Publishing Company.

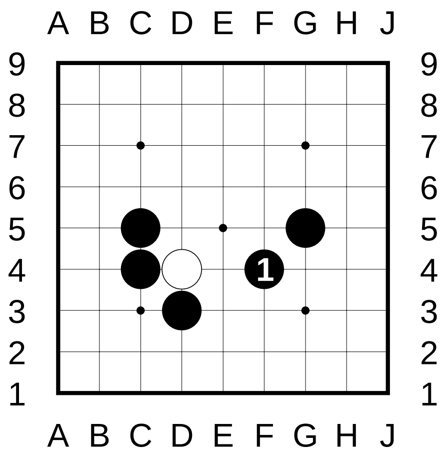

Figure 15: the correct move

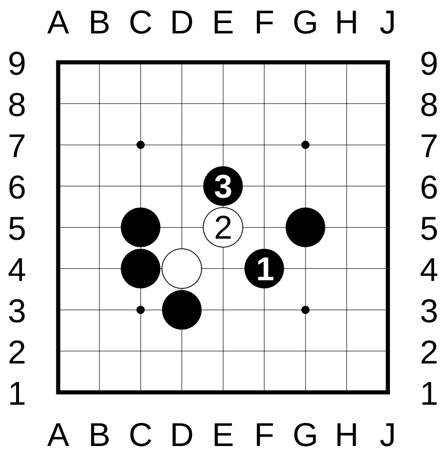

Figure 16: black caps to prevent white's escape

[2] Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

[3] Willmott, S.; Richardson, J.; Bundy, A.; and Levine, J. 1999. An adversarial planning approach to go. In *Proceedings of the First International Conference on Computers and Games*, 93–112. Springer-Verlag.

[4] Zhao, L., and Mueller, M. 2004. Game-sat: A preliminary report.

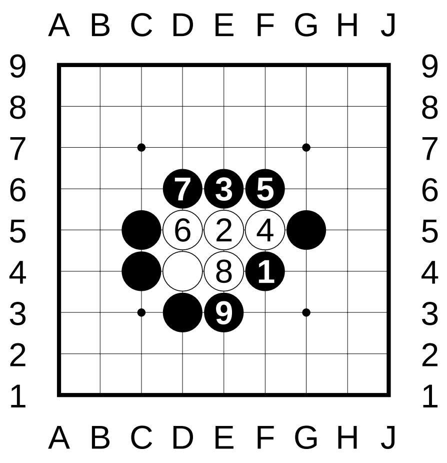
Figure 17: a possible continuation

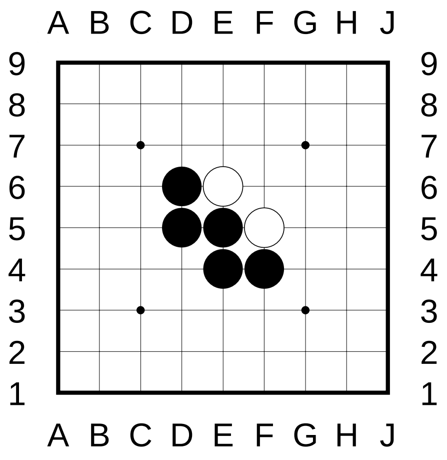
Figure 18: black can capture one of the two white pieces

Appendix A

Work done by Seth Cooper[3]

- DPLL based pruning expression solver
- Minimax framework
- GUI for solving puzzles

Work done by Kevin Wampler[4]

- coding of the rules of go
- pruning expression generator
- Minimax plugin for go

Work done by both authors:

- General framework for game playing
- Discussion of algorithms

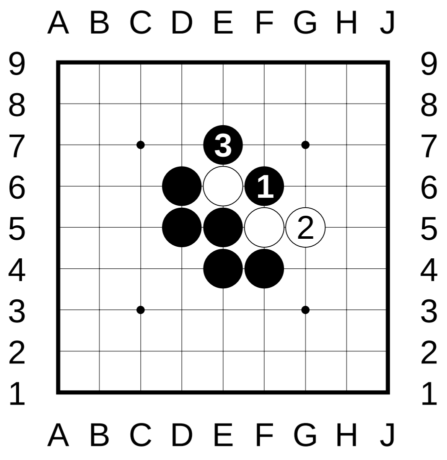
Figure 19: white saves the top piece

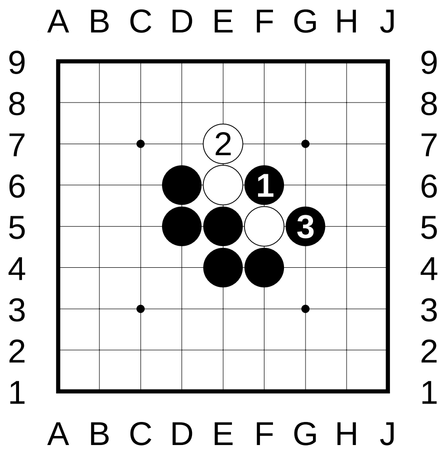
Figure 20: white saves the other piece

[3]Puny human number 1

[4]Puny human number 2

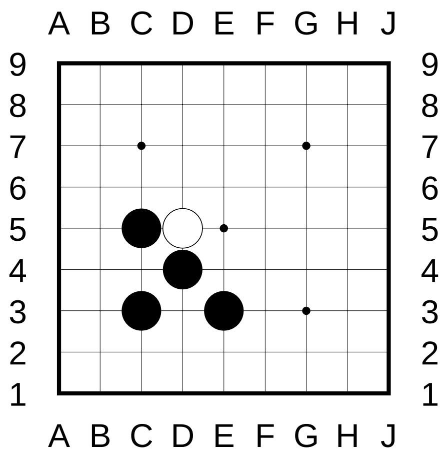

Figure 21: white cannot be easily captured

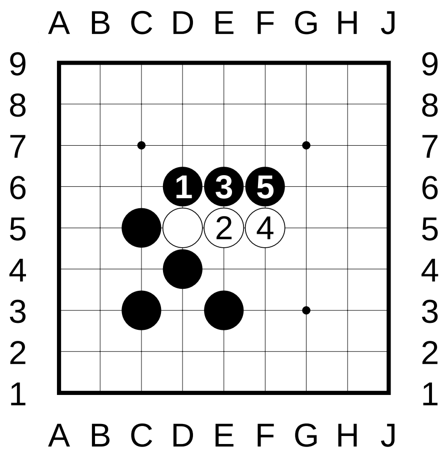

Figure 22: white avoids capture, but black gets a wall

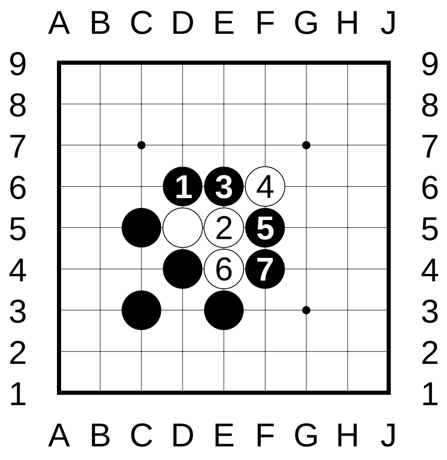

Figure 23: white attempts to spoil the wall, but is captured.