Board state evaluation in the game of Go - Preliminary WPE report

James Parker
jparker@cs.umn.edu

December 4, 2015

Abstract

The game of Go is very interesting from a machine learning point of view since it has been shown to be PSPACE hard to solve explicitly [9]. A major factor for the complexity is due to being played on a 19x19 board where moves can not be considered using only spacial locality. Heuristics are difficult to make for Go because much of the game is qualitative and it is hard to assign quantitative values to such situations. This paper extends GNU Go [6], a hybrid heuristic based Go program, in three ways: decomposition of patterns for influence propagation, adaptive strategy for move generation and risk identification and resolution for move generation. Pattern decomposition is analyzed and shown to be inferior to the original patterns for influence propagation. Adaptive strategy and risk identification and resolution are shown to improve the performance of GNU Go by a statistically significant amount.

1 Introduction

Go has a large state space and a branching factor of over 300 for most of the game, creating difficulties for any search that does not prune the search tree very heavily. Moves in Go quite often do not exhibit locality, where stones on opposite sides of the board can have drastic impacts on each other, so pruning in this fashion does not work. Additionally, Go has many qualitative properties when analyzing board positions that are hard to quantify reasonably in a computer. It is often very difficult to balance searching for more good moves and closely examining good moves already found under time constraints. All these factors are large hurdles for most searches to overcome and by studying Go many new and more efficient ways to search huge abstract spaces have been found.

Computer Go has been studied since the 1960’s with the first program being created Albert Zobrist and a much stronger program in 1972 by Reitman and Wilcox. Zorbist hashing [16] is still widely used when turns can be represented as a change of a few states. This is quite often true in board games where a single piece either appears or moves from one spot to another. Despite active research for about fifty years, the best Go program running on supercomputers is still at an amateur level of play. The current most popular algorithm is a Monte-Carlo tree search [5] that relies on efficient state evaluation, random play, large amounts of simulations and substantial processor power. Although computer Go is still not at a professional level, much progress has been made over the last few years in creating heuristics and searches that exploit computer parallelism efficiently.
Figure 1: A 19x19 board of Go in the starting position where black has not played yet.

There are many diverse approaches on how to evaluate the state of a Go game. Some approaches rely heavily on generally good patterns of play, while others offer only minimalistic patterns and attempt to learn the rest. Some programs rely heavily on simulation data while others go off purely the evaluation of the current state. Surprisingly playing randomly when there is a lack of heuristics and evaluating the results has yielded great results for the Monte-Carlo algorithm.

This paper provides three novel contributions to GNU Go [6]. First, the influence function of GNU Go is changed from a large database of complex patterns to a single stone decompositions of these patterns. Second, the winning margin of the player is used to create an adaptive strategy that plays defensively when winning and aggressive when losing. Third, every move is evaluated by a risk heuristic and if the move is identified as risky, then further searching is done. The pattern decomposition performs poorly and the reasons why this is true are explored. Both adaptive strategy and the risk heuristic show statistically significant improvements over an unmodified GNU Go agent.

First a brief overview of the rules of Go and the GNU Go program is given in section 2. Next, we review the current approaches to play Go in section 3. Then the novel contribution to the GNU Go agent are detailed in section 4. The results for these modifications are explained in section 5. Lastly the conclusions and future work end this paper in section 6.

2 Go Overview

2.1 Brief Go introduction

Go is played by two players (white and black) taking turns placing stones on the intersections of the Go board shown in Fig 1. Board positions are described as a letter and number pair corresponding to the letter found by following the vertical line and following the horizontal line to the edge of the board. For example, the black stone in Fig 2(a) is on the board position C3. For shorthand,
(a) The black stone at C3 will be captured when the four white transparent stones surround the adjacent intersections.

(b) To capture this group of stones the sixth transparent stone is needed.

(c) When white plays the transparent stone neither white nor black will have any free intersections. In this case, the attacking player (white) captures the enemy stones.

(d) Black cannot be killed in this shape since white cannot play both C1 and A1 in the same turn.

Figure 2: The basics of capturing stones. Entire stone groups must be surrounded to capture them. This makes “living” much easier than “capturing” in general. Some patterns cannot be killed once formed.

The player’s color may be incorporated in the board position, so black playing C3 and B-C3 are indicating the same black stone. Each intersection around an enemy stone must be occupied by the player’s stones to ”surround” and ”capture” the enemy stone. For example in Fig 2(a), white needs four stones to fill in all the intersections around the black stone at C3: C4, D3, C2, B3. Similarly if a player has multiple stones in adjacent intersections, this is called stones being ”connected” or a ”group” of stones. A group of stones cannot be killed individually and now the all the combined intersections of the group must be filled before the whole group is captured. This is shown in 2(b) as white needs a sixth stone at D2 before the two black stones can be captured. Captured stones are taken off the board and each stone is one point at the end of the game. At the edge or corner of the board there are less free intersections since stones cannot be played outside. This means edges have three intersections and corners only have two. When a player is capturing a enemy stones, the enemies’s filled in intersections are counted first, so as shown in Fig 2(c) although white has no open intersection when playing A1 neither will black. In this situation since white is putting down the attacking stone, black’s stones are killed and the white stone at A1 has two free intersections: A2 and B1. Some shapes cannot ever be captured once formed, for example black in Fig 2(d). If white plays at A1, it has no free intersections while black has one free intersection at C1, so white’s stone is captured by black. Similarly if white tries to play at C1, black will still have a free intersection at A1, so white’s group of stones at C1 and D1 will be captured. Thus black cannot be killed since both A1 and C1 cannot be filled in a single turn and white will get captured immediately playing in either space. An area completely surrounded by the player’s own stones is called an ”eye”. In Fig 2(c) black has one eye at A1 since all the intersections are completely surrounded by the same black group. Similarly in Fig 2(d) black has two eyes, one at A1 and another at C1 and D1. Any pattern with one eye is still able to be killed, but having two or more eyes ensures that these stones cannot be captured.

A group of stones is considered ”dead” if the player has no way of making their stones into
2.2 GNU Go engine overview

GNU Go [6] is a hybrid heuristic based model that uses large databases to try and identify shape and good moves. There are three main steps used to determine where to play: gathering information about the board, listing possible moves, and then deciding the best move.

The first step of information gathering is to identify groups of stones on the board. To do this, stones that are directly connecting to each other are identified as "worms". After all worms have been identified, GNU Go tries see if it is either possible or impossible to connect stones together. Worms that are possible to connect together are then grouped into larger "dragons". When the
dragons have been formed, it is possible to compute the overall influence on the board and remove stones that are assumed to be dead from any computation. A small amount of information is stored about dragons and worms to make it computationally easier to reconnect them on the players next turn.

The influence function is built to be exponentially decaying from any stone as shown in Fig 3(a). In addition, both ally and enemy stones block influence from spreading, though ally stones have influence of their own as seen in Fig 3(b). Once the board has estimated influences, it is possible to see if any groups are being surrounded without eyes and are in danger. A heuristic that many professionals use is the convex hull of surrounding enemy stones. This means that if a line is drawn between all stable nearby enemy stones and an ally group is fully enclosed, this ally group is in trouble and thus "surrounded". If a group is almost certainly surrounded it is assigned a "critical" status. If the enemy groups enclosing an ally group are rather far away, the ally group is marked as "weakly surrounded". If the enemy groups influence is far too low, the ally group is assumed to be "safe".

Moves are not actually all generated at once. Throughout the entire information gathering phase, possible moves are marked and will be evaluated later in addition to any moves saved from previous information gathering. Many interesting moves are found when running a specialized (binary values for connected or disconnected) minimax search to see if worms can be connected into dragons. If a dragon is in danger of being surrounded by the opposing player, then possible moves to escape will searched up to a specified depth. There are 10 other depth parameters that effect the search size in other areas such as ability to see if enemy worms can be disconnected.

The final thing is to analyze all the possible moves listed previously and decide the best one. In addition to possible moves being identified, each move passes heuristic reasons why it was picked. For example moves could picked because they attack enemy stones, expand your own territory or connect your groups together. The move and reason pairs are then evaluated based on a heuristics that assigns numeric values and the maximum valued move is chosen.

3 Related work

Almost all the top Go programs are based off the Monte-Carlo tree search approach, such as Mogo [8]. The basics behind this approach is to generate a large amount of pseudo-random test moves and efficiently evaluate which samples are good. A discovery in a new UCT (Upper Confidence bound applied to Trees) algorithm by Kocsis and Szepesvári [7] has greatly increased search efficiency with a good memory/time trade-off. UCT computation is also extremely parallel gaining access to the benefits of multi-core or grid computing [2]. The Monte-Carlo approach is weak when there are long games or a good heuristic, but the lack of decent heuristics in Go for both move formulation and state evaluation find the randomization of Monte-Carlo appealing [8]. However, as Kishimoto et al. [15] discovered this Monte-Carlo search suffers from diminishing returns, causing later passes to yield less information than earlier ones. The reduced benefit for doubling processing power for large computations was verified by explicit testing by Bourki et. al [1].

Another method that is reinforcement learning [11] [10] that tries to incorporate locality. Silver et. al’s program [11] uses patterns that are recognized "good" among professionals and tries to
learn how the enemy professional responds. By learning responses, there is no need to precompute large databases of useful moves to be searched. Another positive factor of response learning is that the programmer’s bias is not carried over to the program. Like all learning algorithms, this requires time to generate important patterns and many games need to be observed before results can be extracted. Since only local patterns are analyzed, this program is designed to run on a smaller 9x9 board, which has much greater locality than a normal 19x19 board.

Neural networks are similar to the standard reinforcement learning techniques and are often implemented using the perceptron model. Once again, this type of learning requires a delay before the system can play efficiently but does not require precompiled databases of moves. One major strength in neural networks is accurately evaluating likely end game positions [14]. The perceptron is also skillful at using many single stones to generate a overall strategy, although it falls short when stones get intertwined and it is unclear who is capturing who [4].

Chen et. al’s [3] work excels at these local fights by using a heuristic based approach. Heuristic models are probably most intuitive since you have direct control over what you feel are important qualities. This control allows easier debugging to see why the program is playing a certain way. The major drawback of this approach is the programmer often introduces their own bias and play style to the code. Additionally, assigning quantitative values to situations in Go can be quite challenging. It is often easy for humans to see which move is better than another, but only very rarely are numbers used to describe these reasons.

4 Novel contributions

4.1 Influence pattern decomposition

As mentioned before whole board evaluation of moves in 19x19 games is quite hard, so it is often done on a more local scale. However, stones on the exact opposite side of the board can have a direct effect on local strategy as seen in Fig 4. This situation is called a "ladder" and is very common to have multiple of these form in a game, so almost all advanced programs check for this explicitly regardless of what other strategies are implemented. There have been many ways of trying to use local information to generate moves [12] [13] which have moderate success on 9x9 boards, but on 19x19 sized boards these methods perform poorly.

GNU Go’s influence function is based off local information. A single stone has influence worth 100 at its center, and then is radiated out exponentially decreasing (by a factor of 3) uniformly in all directions as shown in Fig 3(a). This influence is computed for every source (stone or pattern match) on the board and a total influence map is formed by summing all the individual influences and used to generate values for moves. GNU Go is heavily dependent on patterns like the ones shown in Fig 5. The faded out stones can be either a stone of that color or empty and an empty intersection means no stone is present. If stones match this pattern under any rotation or reflection, a new influence source is added at the location and strength indicated by the number.

Over 50 percent of the patterns in the influence database are built out of eight basic subshapes shown in Fig 6. Instead of using the Japanese or other commonly used names for these two stone shapes, we assigned short descriptive names due to the inability to rotate the patterns. For example both k2pat and k2mpat are commonly called a "knight’s move" in reference to how the chess
(a) After W-S19, B-T19 and white is dead. (b) With this single white stone at S18, after B-R19 white still has two spaces open. This means white can play B5 and kill black stones before black can kill white.

Figure 4: Example of whole board evaluation needed called a "ladder". The solid colors are the starting position and the faded out stones are stones that are going to be placed. In both figures it is white’s turn to move and must play B3 or black will and kill the stone.

(a) A GNU Go defined pattern (b) The influence spreads perpendicular to the line connecting the stones

Figure 5: A GNU Go pattern is matched up to all 8 symmetries of rotations and reflections. The 30 represents a new influence source that will spread influence similar to a stone. Stones have a staring influence of 100.
Figure 6: The eight single stone pattern matches for generating vertical influence. Notice that although k2pat and k2mpat are rotations of each other, the influence spreads differently vertically vs horizontally so they require separate patterns. My patterns are only matched with the 4 reflections (2 for a1, a2, and a3) about the stone generating influence. Each pattern will be matched twice, once for each stone when generating its influence spread.

piece moves, but we require different weights for these two patterns since they spread their vertical and horizontal influence differently. For example, when spreading influence horizontally k2pat has a stone in the direction of the influence spread, but k2mpat has a stone perpendicular to the influence spread. The a1pat is short for "Adjacent pattern, distance 1" and a2pat is short for "Adjacent pattern, distance 2". Similarly "Diagonal pattern, distance 1" is shortened to d1pat, "Knight pattern, distance 2" is shortened to k2pat and "Knight major pattern, distance 2" is shortened to k2mpat.

As seen by these patterns, stones when grouped together do not emit influence uniformly in all directions. The pattern shown in Fig 5(a) indicates that influence spreads fairly perpendicular to a line connecting the stones as shown in Fig 5(b). This is a common feature across almost all the patterns for extending influence in the GNU Go database. Building upon this, we assigned both a vertical and horizontal influence value to each pattern and spread them in their corresponding direction. Notice that in Fig 6 there are no stones straight above or below the influencing stone because this stone would greatly block the vertical influence spread. Let $d_i$ be the horizontal distance to the influence source and $d_j$ be the vertical distance. Then the calculation for the modified influence is done by Algorithm 1, where $h$ and $v$ subscripts denote horizontal and vertical directions respectively. The "for $j = (1, 2, ..., 8)$" implies $j$ is being used to index the eight patterns in Fig 6.

Using $cos^2$ and $sin^2$ for Factor$_h$ and Factor$_v$ respectively ensures that $Spread_p \geq 1$ since we assume for $i = (h, v)$, $j = (1, 2, ..., 8)$ Pattern$_i$Value$_j \geq 1$. Here the $j$ indexes correspond to the eight possible pattern matches, while the $i$ indexes define whether this is a horizontal or vertical pattern. The patterns are combined as a product instead of a sum to help overcome the exponentially decreasing influence spread. However, the values of Pattern$_i$Value$_j$ are fairly close to 1 and normally stones are spread loosely thus not matching too many patterns simultaneously, so there should not be a large difference between the two methods. The values for the Pattern$_i$Value$_j$ on all eight of my patterns are quite sensitive and we could not ad hoc assign them values and get good results. To compute the values, we initialized all Pattern$_i$Value$_j = 1$, thus not spreading any extra influence. Then we did a floating random walk as described in Algorithm 2.

If SmallRandomNumber() pushes Pattern$_i$Value$_j$ in the right direction, it will have a greater chance of winning and thus being saved as the most recent. This means that probabilistically the saved values are closer overall to the correct values. The value of SmallRandomNumber() is initially larger valued but then decreased as it converged to a local maximum. Since these are
Algorithm 1 Calculate adjusted influence spread

1 for $i = (h, v)$, for $j = (1, 2, ... 8)$ isPattern$\_i$Match$\_j() = \begin{cases} 1 & \text{if match} \\ \frac{1}{\text{Pattern}_i\text{Value}_j} & \text{otherwise} \end{cases}$

2 $\text{Pattern}_h() = \prod_{j=1}^{8} \text{isPattern}_h\text{Match}_j()\text{Pattern}_h\text{Value}_j$ Comment: See Alg 2 for $\text{Pattern}_h\text{Value}_j$.

3 $\text{Pattern}_v() = \prod_{j=1}^{8} \text{isPattern}_v\text{Match}_j()\text{Pattern}_v\text{Value}_j$ Comment: See Alg 2 for $\text{Pattern}_v\text{Value}_j$.

4 $\text{Factor}_h = \frac{d_i^2}{d_i^2 + d_j^2}$ Comment: $\cos^2$

5 $\text{Factor}_v = \frac{d_j^2}{d_i^2 + d_j^2}$ Comment: $\sin^2$

6 $\text{Spread}_p = \text{Factor}_h * \text{Pattern}_h() + \text{Factor}_v * \text{Pattern}_v()$

7 $\text{Influence}_\text{new} = \text{Spread}_p * \text{Influence}$

Algorithm 2 Floating random walk

Initialize:

1 for $i = (h, v)$, $j = (1, 2, ... 8)$ $\text{SavedPattern}_i\text{Value}_j = 1$

2 while(not converged)

3 for $i = (h, v)$, $j = (1, 2, ... 8)$ $\text{Pattern}_i\text{Value}_j = \text{SavedPattern}_i\text{Value}_j$

4 for $i = (h, v)$, $j = (1, 2, ... 8)$ $\text{Pattern}_i\text{Value}_j = \text{Pattern}_i\text{Value}_j \pm \text{SmallRandomNumber}()$

5 result=$\text{RunGoGame}()$

6 if result == win

7 then $\text{SavedPattern}_i\text{Value}_j = \text{Pattern}_i\text{Value}_j$

8 decrease range $\text{SmallRandomNumber}()$

9 end if

10 testStatisticalConvergence()

11 end while
basically Bernoulli trials, the mean of the saved history of wins converges to the correct local maximum with variance decreasing proportional to \( \frac{1}{\sqrt{N}} \) with \( N \) being the number of recorded successes.

### 4.2 Adaptive strategy

Professional Go players constantly estimate which player is winning and change how they play based on this estimate. Since a 1 point win is worth just as much as a 30 point win, Go professionals will often not kill their opponent’s stones when they are winning even if they think they can. This is because killing decently sized groups can start fierce battles that are very complex and risky. Instead Go professionals allow their opponent to live and keep the game stable with them in lead. GNU Go cannot estimate or read nearly as well as professionals, but dynamic strategy adaptation is still possible.

After gathering information about possible moves, GNU Go then evaluates them on a number heuristics, some examples are:

- Territorial value
- Strategic value
- Follow up value
- Reverse Follow up value
- Shape value
- Threat value

These values are combined a very complicated way to evaluate a move’s overall score based on data from the information gathering phase. By adding weights to some of the heuristic values, the final score can be biased towards different moves than it normally is. To do this the following weights were created and paired with their respective values during the score computation:

- Attack weight - Moves that attack the opponent’s small groups
- Defend weight - Moves that defend your own small groups
- Strategic attack weight - Moves that attack the opponent’s large groups
- Strategic defend weight - Moves that defend your own large groups
- Territorial weight - Weight of your expected points from enclosed intersections
- Strategic weight- Pattern matched moves that strengthen your own large group or weaken your opponents
- Attack dragon weight - Weight if a move is needed when your opponent plays next to an already dead group (safety move)
• Follow up weight - Weight for if the opponent ignores our move and we get 2 moves in a row in the same local area

• Minimum value weight - Minimum final value factor for patterned sequence of moves.

• Maximum value weight - Maximum final value factor for patterned sequence of moves.

• Invasion malus weight - Reduction in score for playing isolated moves close to large enemy groups (High weight makes the player not invade)

In addition to these weights, the game state must be analyzed to determine what strategy should be implemented. Two factors are considered for our adaptive strategy: PercentGameFinished and score. The PercentGameFinished variable estimates how much of the game has elapsed on a scale of 0 to 1. Values near 0 mean the game is in the "opening", values near 0.5 mean the game is in "midgame" and values near 1 mean we are in the "endgame". The difference between two players score is our score variable so we can track who is winning and by what margin. The PercentGameFinished and score are recomputed after every move and adjustments to the weights are made if necessary.

There are three distinct categories of playing: even, uneven and desperate. Regardless of the PercentGameFinished if the scores are fairly even use the default weights (all 1). If the score is too uneven and we decrease and increase various factors as shown in Table ??). If a specific weight is being decreased when losing, that same weight will be increasing when winning and vice versa with the exception of minimum weight and maximum weight. The uneven playing style is modeled linearly with the weights being a response to the difference in score, so large score discrepancies will cause very shifted weights. The desperate playing only happens when we are losing and the game is about to end. At this point, the invasion malus weight is decreased so the program will try to make any desperate attacks within enemy territory if it thinks there is a small possibility of succeeding.

The weight adjustments are ad hoc heuristics that attempt to create the following conditions. If we are losing attack and strategic attack will become more appealing and similarly defend and strategic defend will be valued less since we will not win by defending. When losing we need to increase our score which is done by increasing territory owned, thus the territorial weight is increased. The strategic weight is the opposite, it sets up pattern generated good positions for potential points later, but if we are already behind we need the points now and not hopefully in the future. If an opponent plays in a spot that we think is already dead and we are losing, we should ignore their hopefully wasted turn and gain points elsewhere reflected in decreasing attack dragon. If we are being this aggressive when losing, the opponent is probably going to respond to us, so we balance this out by decreasing the follow up weight. The minimum and maximum weights are only really used for for patterns of play normally agreed upon to be the best outcome for both sides. By reducing the maximum weight, it allows winning players to value moves with greater stability and losing players to value more risky moves than the standard pattern. The reverse is also true, by increasing minimum the standard patterns will be picked in the absence of any other good moves. These standard patterns have been well researched in the Go community, so both a winning and losing player will want to trust them in the absence of any other good moves. The invasion malus is one final effort to try and do something drastic by playing in areas heavily influenced by the opponent and trying to live, thus reducing their territory and score.
4.3 Risk identification and resolution

Many programs assign a fairly fixed amount of time/processing for each move. The only variation in speed for GNU Go is a decrease in performance if a tournament game’s time is running out. Humans have a large variation in the time spent thinking about moves, because they have learned to understand which moves are easy and which moves are hard. The lack of dynamic time allocation in Go programs is detrimental because one mistake at a critical time can completely reverse the outcome of a game. When a human is playing Go, they are normally quite aware of the dangers and gains in the moves they are considering. Normally it is the moves a player did not consider that cause a loss and not a misinterpretation of what was at stake. If a player is doubtful of a situation, they will spend extra time thinking more carefully about the move until they either feel satisfied in their response or pressured on time. Computers might take into account what is at stake as part of a move’s value, but they do not allocate time differently for high or lower valued moves.

Here we propose a heuristic for evaluating the "risk" of a current move to adjust the exploration and exploitation of a move. To identify the risk a best move will be found at some level before the end of the search depth called the "potential best move," \( \text{Best}_p \). The program will then finish the searching to its normal depth and find the actual best move, \( \text{Best}_a \). We propose identifying risky situations based on three factors: \( |\text{Value(}\text{Best}_a\text{)} - \text{Value(}\text{Best}_p\text{)}| \), \( |\text{Position(}\text{Best}_a\text{)} - \text{Position(}\text{Best}_p\text{)}| \), \( |\text{Value(}\text{Best}_a\text{)}| \). Resolution of the dynamic time allocation will be based on: \( \text{PercentGameFinished} \) and \( \text{PercentTimePassed} \).

The first step in allocating time dynamically to risky situations is identify when a risk is present. Risky situations are classified into three types: large move values, move uncertainty and a sudden realization of game direction. Large move values will indicate a large change in score, so time spent thinking carefully about this move will give a high utility. Uncertain moves happen if a move was originally thought to be optimal, but this move turns out to be suboptimal after thinking more. In this case more time should be spent exploring other move options if the difference in the move’s value is not small. A sudden realization of game direction happens when a move that looks poor suddenly seems good after further analysis or vice versa. If a moved looked good, but after consideration this move can actually be outplayed by the opponent, more effort should be put into finding a new move. Likewise, if a move seemed poor yet optimal, but after contemplation this move is seems good, the player should check to make sure the opponent really has no escape before committing.

After the risk has been identified, the question of how much extra time should be allocated to move searching still remains. If every move was given a large amount of time the Go agent is likely to run out of time, so some sort of time management is needed. The opposite is also true; extra time at the end of the game adds nothing to the player’s score, so utilizing it all is important. If the game has progressed faster than our time limit (for example 50% of the moves have been played, but 75% of our time remains), we create a "reserve time" of the difference (in this case 25% of the total time). This situations corresponds to when \( \text{PercentGameFinished} = 0.5 \) and \( \text{PercentTimePassed} = 0.25 \). The next move that causes the agent to think uses up the whole reserve and then a new reserve is slowly accumulated every subsequent move until the next risky move is encountered. The risk identification and evaluation implementation is detailed in Algorithm 3.
Algorithm 3 Risk identification process and evaluation

1. depth = default\text{depth}
2. branch = default\text{branch}
3. Start depth search
4. Gather potential best move, Best\text{p}.
5. Finish normal depth search and find best move, Best\text{a}.
6. Record percent time passed, PercentTimePassed.
7. Estimate percent game finished, PercentGameFinished.
8. if (|Value(Best\text{a})| > Threshold\text{v})
9.   depth = depth\text{v}
10.  branch = branch\text{v}
11. endif
12. else if (Best\text{a} ≠ Best\text{p} AND |Best\text{a} − Best\text{p}| > Threshold\text{m})
13.   depth = max(depth\text{m}, depth)
14.   branch = max(branch\text{m}, branch)
15. endif
16. else if (|Best\text{a} − Best\text{p}| > Threshold\text{d})
17.   depth = max(depth\text{d}, depth)
18.   branch = max(branch\text{d}, branch)
19. endif
20. time\text{a} = AllocateTime(PercentGameFinished, PercentTimePassed)
21. if (time\text{a} > 0)
22.   RecomputeBranch(branch, time\text{a})
23.   RecomputeDepth(depth, time\text{a})
24.   ExtendMoveGeneration()
25. endif

AllocateTime(PercentGameFinished, PercentTimePassed)
1. if (PercentGameFinished ≤ PercentTimePassed)
2. return : 0
3. else
4. return : PercentGameFinished − PercentTimePassed
5. endif
This risk resolution adjusts both the exploration and exploitation of the Go program based on a set of predetermined weights. The \( \text{branch}_i \) and \( \text{depth}_i \) for \( i = (v, m, d) \) correspond to the three cases when a risk is identified and they are weights used for altering the branching and depth factors for the search on Line 24. These weights are set heuristically, although they could be optimized with any reinforcement learning algorithm. The \( \text{AllocateTime}() \) function is an ad hoc heuristic of reserve time. Line 8 corresponds to the risk identification by a "large" value, we define "large" as a difference greater than \( \text{Threshold}_v \). Similarly line 12 checks if the \( \text{Best}_a \) is a completely different move than \( \text{Best}_p \), corresponding to move uncertainty. The first possibility is \( \text{Best}_a \) and \( \text{Best}_p \) are very close in value and the extra reading depth just slightly favored \( \text{Best}_a \). In this case there might be many moves of similar value and any of these moves are acceptable so \( \text{Best}_a \) is picked, causing the extra "AND" condition in line 12. If the moves are different and the values of \( \text{Best}_a \) and \( \text{Best}_p \) have a difference of \( \text{Threshold}_m \), that means \( \text{Best}_a \) suddenly got a big boost in score after finish the search. This variance in the move values probably indicates that the best move is still possibly unstable and deeper reading should be done. Line 16 checks if the difference in values is large (greater than \( \text{Threshold}_d \)) between \( \text{Best}_a \) and \( \text{Best}_p \) and both moves are in the same spot, then this is classified as a sudden change in game direction. \( \text{Threshold}_i \) for \( i = (v, m, d) \) are all heuristically set and the exact values are given in Section 5.4.

5 Results

5.1 Setup overview

The basic GNU Go [6] version 3.8 was used as a baseline for both implementation and testing. The actual program was ran on a number of different machines with capabilities between 2.4-3.2 GHz, at least 2 cores (though only one was used at a time), 2GB - 1TB of RAM all running Ubuntu 8 or 9. There is a rather large range on RAM and a little spread of processor power, but this did not effect my results for the following reasons. Each instance of GNU Go ran only used about 20MB of RAM, so even on the smallest machine this was 1% of the total capability. GNU Go did use all of the processor power, but GNU Go does not have a search or process limit per move based on time. It would simply takes longer on a slow machine to generate the same move it would have on a faster machine with the same current seed. The only time game duration was important in my results is for the risk evaluation, and the same machine (the fastest) was used in all these calculations.

The bulk of my simulation testing was done through an online Go service called KGS. Each machine would start two instances of GNU Go, which would log on and play each other. This ensured the timing results are accurate since matched opponents were always run from the same machine. This did not cause an issue with hardware capacity since GNU Go does not prospectively compute anything while it is the opponents turn. So the active player’s agent would get 100% of the CPU power and the 40MB of memory also is not a limitation. Various configurations of the bots would log on and repeatedly play against each other until a significant amount of games were simulated. The agents were randomly assigned starting colors to further ensure no bias was present. Games were set to the default KGS settings of 19x19 board, 6.5 komi, 30 minute games, 30 second byo-yomi time, 5 byo-yomi periods, no handicap and Japanese scoring.
Table 1: Means and standard deviations of pattern weights. Standard deviations are applicable since the data is following a fairly normal trend. As seen the perpendicular patterns are higher weighted with the exception of \textit{a2pat}.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1pat</td>
<td>1.305</td>
<td>0.08634</td>
</tr>
<tr>
<td>a2pat</td>
<td>1.065</td>
<td>0.04707</td>
</tr>
<tr>
<td>a3pat</td>
<td>1.175</td>
<td>0.07313</td>
</tr>
<tr>
<td>d1pat</td>
<td>1.081</td>
<td>0.04850</td>
</tr>
<tr>
<td>k2pat</td>
<td>1.020</td>
<td>0.02636</td>
</tr>
<tr>
<td>k2mpat</td>
<td>1.163</td>
<td>0.07907</td>
</tr>
<tr>
<td>k3pat</td>
<td>1.069</td>
<td>0.04962</td>
</tr>
<tr>
<td>k3mpat</td>
<td>1.179</td>
<td>0.08835</td>
</tr>
</tbody>
</table>

5.2 Influence pattern decomposition analysis

The influence function did very poorly with a pitiful winning percent against the base agent of 2.469%. Heuristic patterns in GNU go are categorized to convey what Go aspect they are trying to mimic. The categorization of "CLASS E" patterns in GNU Go are called "extensions" which add extra influence sources. An example of a "CLASS E" pattern was presented in Fig 5 back in Section 4. Half of the "CLASS E" patterns can be decomposed into the eight sub patterns shown in Fig 6, and the other half of the patterns involve walls to add incentive to play near them.

The results of the floating random walk are shown in Table 5.2. As seen in Fig 7, most of the variables are showing a fairly normal spread allowing us to use a normal approximation for the Bernoulli trials. This approximation was used to compute the convergence of the various patterns. As seen in Fig 7(g) most of the values are around 1. When running the floating random walk algorithm, the values were not allowed to run below 1 since this would represent a stone somehow having a negative non-blocking effect on the spread of influence. The true value since value for \( k3pat \) is near 1, so Fig 7(g) represents a normal distribution with the left half truncated.

There are two main reasons why the single stone pattern decomposition approach does poorly against GNU Go's pattern database. First, multiple stone patterns do not decompose to be the combination of multiple single stone patterns. An example of this is shown in Fig 8(a) where this is a horizontal pattern spread match of both \( a1pat \) and \( d1pat \). However \( a1pat \) and \( d1pat \) are mutually blocking each other, so the marked stone only gains benefit from one of them at a time, depending on whether the influence is spreading left or right. The second reason the "CLASS E" patterns do much better is because the influence points that are added are sometimes very far away. This makes stones feel safe farther away from the player's own stone and near walls. Due to the exponentially decreasing influence spread, the basic patterns would need extremely large weights to spread influence as far as some "CLASS E" matches as shown in Fig 8(c) and ??.. In these figures, a weight of 12 (optimized weight was 1.065) would have to be assigned to \( a2pat \) to extend the influence of my basic patterns as far as the GNU Go patterns.

Both of these reasons cause the influence agent to play too locally. The double counting for influence adds value to thick sets of stones as shown in Fig 8(a), but playing thickly is bad unless the opponent is threatening an important group. The lack of extended spread from "CLASS E" patterns also makes the agent play too locally because it thinks more stones are needed before the area is secure enough. An example game where this is exemplified is shown in Fig 9. Without the wall specific patterns, the agent does not extend along the wall nearly as well. In this example game black has solidified the upper left corner, but at the cost of the rest of the board. After removing the wall specific patterns in "CLASS E" (so 100% of the patterns used could be decomposed into the single stone patterns), the modified agent's win percent went from 2.469% to 5.056% which is
Figure 7: Results of the floating random walk. As seen most are fairly normal with a definite peak around locally optimal values.

double yet still very poor. These results indicate that patterns must be matched at their highest level that allows only one pattern match. If multiple patterns can be matched, only one of these may be picked.

5.3 Adaptive strategy analysis

The results in this section yielded a 55.00% winning percent against the base agent in exchange for small computational and memory overhead. The only calculations that are done are a score evaluation and an approximation of what percent of the game is done, PercentGameFinished. The score estimation is already calculated and used in other parts of GNU Go (any many other programs), so this value can simply be stored and reused. The PercentGameFinished is part of the base GNU Go and computed by seeing what percent of the board is considered solid territory along with how strong the influence is at each board position, a similar process to the score evaluation.

The tested values of the strategies are shown in Table 2. At every turn the current weights for a strategy were computed by linearly interpolating the two points shown in Table 2, except for the minimum and maximum weights. These interpolations normally do not have the intercept going through weight = 1, however this is not a problem since a threshold is required before the weights are used. The minimum and maximum weights are linearly interpolated from the absolute value
(a) Stones are too close, thus blocking influence.

(c) To achieve the same effect the basic patterns would need high weights and "over influence" close areas. The colors are simply to make it easier to tell which number belongs to which intersection.

Figure 8: Stone influence propagation scenarios. The decomposed patterns "double count" tightly pack stones and needs high weights to spread far.

Figure 9: Black has used 11 moves in the top left, while white has only used 3. This allowed white to map out most of the board while black is confined to the corner.
### Table 2: The ad hoc values for weights. To determine the current weight for score is the linear interpolation of the winning and losing score shown, except for minimum and maximum weight. For these two the absolute score is used as an indicator.

<table>
<thead>
<tr>
<th></th>
<th>Attack</th>
<th>Defend</th>
<th>Strategic attack</th>
<th>Strategic defend</th>
<th>Territorial</th>
<th>Strategic</th>
<th>Attack dragon</th>
<th>Follow up</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Losing (-50 score)</td>
<td>1.5</td>
<td>0.9</td>
<td>1.5</td>
<td>0.9</td>
<td>1.4</td>
<td>0.8</td>
<td>0.7</td>
<td>0.7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Even (no adaptive)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Winning (+50 score)</td>
<td>0.8</td>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>0.9</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
<td>0.6</td>
<td>0.6</td>
</tr>
</tbody>
</table>

of score with the point score = 0 → weight = 1. If the score difference was less than 10 or the game had not progressed past 35% of the whole game, then the weights were left unbiased all equal to 1. If 70% of the game had passed and the difference in score was over 30, we reduced the invasion malus weight down to 0.4 to allow a final effort for a risky invasion to even out the game. These values were generated ad hoc and despite being unoptimized yield good results. Any reinforcement learning algorithm could yield better values, but the importance of adaptive strategy is shown well enough.

Analyzing the real effect of the adaptive strategy in a game is difficult. After seeding the agents to make their play static, games can be found where the adaptive agent won while the base agent lost. In each of these games it is easy to see when the modified agent’s play started to differ, however all the moves after this point are normally different between games. Because the games end in very different fashions, it is difficult to point out which move sequences exactly changed the game from a lose to a win. For this reason, simply looking at the first move difference is insufficient to decide that this was the game deciding move, instead this move is simply the start of a different sequence that lead to an overall win. Even if we can not analyze the games individual games well, we can see the effect of the adaptive strategy on the total win ratio. As shown back in Fig 11(a) there is a statistically significant improvement in win ratios since we can assume the default agent would win 50% of the time.

#### 5.4 Risk identification analysis

GNU Go uses heuristics and patterns to evaluate the game state. This makes our risk identification much harder since one cannot compare an earlier "depth" of a pattern or heuristic with a later "depth". The primary place depth and branching factors appear in GNU Go is in the tactical reading sections, which tests to see if groups can be connected, killed or saved. Incorrectly reading out sequences of this type can have a large impact on the game as shown in Fig 10. Here white is threatening the large black group at G3, but black does not read far enough ahead to see that the situation is very dangerous if white connects with the stone at H4. Due to black’s inability to see the danger, black plays elsewhere and white does connect by playing H3 and proceeds to kill the black group. After losing such a large group, the difference in points increased by over 30 and black has very little chance of coming back.

There are many depths for the various parts of the tactical reading, most of which have a default value of around 10, except for the connection reading which has a default depth of 64. When the tactical readings were about 75% complete, the current potential best move $Best_p$ was found and stored with its value $Value(Best_p)$. The tactical reading then read to the end of its normal depth and found the new best move $Best_a$ and its value $Value(Best_a)$. The three separate tests for adjusting the depth are as follows:
Figure 10: White has just played J4 and black ignores it and moves at P3, which kills the black group at G3.

- If the threshold for the difference of score values between $Best_p$ and $Best_a$ was set to 4 ($Threshold_d$).
- If the score difference is greater than 25 ($Threshold_v$).
- If the moves $Best_p$ and $Best_a$ were different, and worth more than a 2 score difference ($Threshold_m$).

If any of the above were true, $Best_a$ replaced the old perspective value $Best_p$ and the depth and branching factors were all increased by 3, and new moves were generated with the new best move stored in $Best_a$. The new $Best_a$ was compared with the new $Best_p$ using the same three tests above, and if there was still a large enough difference, the depth and branching factors were all increased by another 3 (for a total of 6). Once again new moves were generated and the current best move was picked as the correct one.

Allocating time for the extra branch and depth processing described above was difficult in GNU Go. Since GNU Go does move generation by fitting patterns in one pass, the only way to reduce time would be to cut out whole classes of patterns or only match part of every class of patterns. Neither of these options are very desirable so instead of allocating the time spent searching, a less desired approach was taken to ensure the agent did not lose due to time. To avoid losing on time, the $PercentGameFinished$ was compared to the percent of time passed in the current game, $PercentTimePassed$. The depth and branching factors were only increased if $PercentGameFinished > PercentTimePassed$.

GNU Go as a whole does not fully utilize the dynamic depth checking due to the patterns and heuristic matching. However, since patterns and heuristics cannot be searched at a larger depth,
the only time moves meet any of the three conditions above is when the move is part of a tactical reading. This means although GNU Go as a whole does not benefit from the depth modification, the part of the program that is causing the discrepancy between move values is getting full benefit.

5.5 Results overview

For the experiments, we matched up seven modified configurations of GNU Go against the base agent. The seven configurations were all combinations of the three modifications suggested earlier: "influence", "strategy", "risk", "influence + strategy", "influence + risk", "strategy + risk" and "influence + strategy + risk". My slowest configuration which used all three modifications took on average 70% more time thinking about moves and the variation was also large. Sometimes the configuration took only a minute more, sometimes it took 20 minutes more.

The number of games played between configurations varied depending on how interesting the results were. The base agent played against itself for 100 games to ensure there was no strong bias in the white vs black win ratios. The results turned out to be 50 wins for both black and white, so we can assume that there is not a strong bias towards starting colors. The "strategy", "risk" and "strategy + risk" simulations were all run over 300 times each, while "influence", "influence + strategy", "influence + risk" and "influence + strategy + risk" were all run around 100 times. As seen from the results in Table 5.5 and Fig 11, "strategy", "risk" and "strategy + risk" all showed statistically significant improvements over the base agent.

The configuration that showed the best results was the "strategy + risk", which won against the base GNU Go agent 56.35%. However, "strategy" alone had a 55.00% win ratio against the base agent and "risk" had a 55.86% win ratio. This means there is some diminishing returns since the combined "strategy + risk" was not too much more than just the "risk" configuration. The confidence intervals are still fairly wide at 300 games, so some of the diminishing returns might be hidden in some unlucky results. There must be a fair overlap where either searching deeper or being more aggressive were able to turn the game around, but since the game will only count as one win the double benefit is hidden.

6 Conclusions and future work

6.1 Conclusions

This paper presents the limitations of decomposing larger patterns into smaller subpatterns in GNU Go’s influence function. Due to the player’s own stones blocking the spread of influence and greater extension capabilities of GNU Go’s larger patterns, single stone patterns cannot perform well even with asymmetric influence propagation. This shows that the net effect of stones is not equal to the

<table>
<thead>
<tr>
<th></th>
<th>Base (by color)</th>
<th>I</th>
<th>S</th>
<th>R</th>
<th>I + S</th>
<th>I + R</th>
<th>S + R</th>
<th>I + S + R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>50%</td>
<td>2.469%</td>
<td>55.00%</td>
<td>55.86%</td>
<td>5.714%</td>
<td>7.453%</td>
<td>56.35%</td>
<td>10.48%</td>
</tr>
</tbody>
</table>

Table 3: Table of averaged winning percent. "I" stand for "influence," "S" for "strategy" and "R" for the "risk" modification. The plus shows when multiple modifications were used together.
Confidence intervals for the good configurations

Confidence intervals for the poor configurations

Figure 11: 95% confidence intervals of the true mean winning percent. The circles are the means and the blue bars are the confidence intervals. As seen the "strategy" and "risk" modifications are statistically significant improvements, while the "influence" modifications do poorly.

Adaptively adjusting the agent’s aggressiveness based on the current win margin is shown to yield a higher success rate than a static strategy. This allows the agent to take chances when losing to try and pull ahead and to sacrifice points for stability when winning. Since winning by 1 point is not any worse than winning by 30 points, this added stability mimics what Go professionals do to control the course of the game. The overhead of computing an adaptive strategy was very minimal in the GNU Go framework, but it would possibly be more difficult in settings where moves are not computed as a function of factors since this strategy depends on being able to weigh these factors differently in the move value function.

A process for risk identification when generating moves is described and used to increase the search if a move is classified as risky. This helps the Go agent manage its time more effectively, since one wrong move in Go can have huge game changing effects. Thankfully not every single move has such a large effect, so identification of which moves are very important is necessary. All of these results are implemented in a modified GNU Go and played against the base agent. Statistical analysis shows all these results to be significant.

6.2 Future work

The adaptive strategy relies on being able to use heuristics and pattern matches to evaluate a move as a function of various factors. Changing the weights on these factors is what allows the current
method to adapt the program’s strategy. In a more simulation based environment like a program using Monte-Carlo, heuristics are substantially less used and the moves are normally decided by which branch has highest simulated win ratio. To accomplish the adaptive strategy in this setting, either more detailed information about the playout branches is needed or some heuristic analysis at various depths of the search. In the current model of adaptive strategy a linear relationship is used for simplicity, but a separate model for winning and losing would probably be more beneficial. This would allow the aggressive agent to have a high affinity for territory while not reducing the defending strategy’s affinity for territory drastically. One possibility would be to have two different linearly functions with an intercept of one. This would require learning the same number of parameters, yet have asymmetrical effects.

The risk identification results were already effective in GNU Go’s framework where extra processing time had diminished payouts. Implementing this strategy to balance the simulation count in a Monte-Carlo algorithm would likely yield much stronger results. Monte-Carlo simulations are also able meet time allocations for moves much easier than GNU Go, which allows more advanced time allocation algorithms. Extending the model to have a quantitative risk identification factor instead of a simply binary one allows the risk factor to be incorporated into the time allocated for better time management. The time allocation based on the progression of the game could also be modeled from professional’s move time allocation instead of the reserve policy used in this paper.

One thing that should be verified is that the adaptive strategy and risk identification and evaluation actually do improve the GNU Go’s base performance. Sometimes quirks can happen when two configurations of a single agent is playing against itself that are not evident in playing heterogeneous agents, such as a Monte-Carlo based Go program or a human. Playing against different configurations allows easy baseline comparisons, but the modified agent should be more thoroughly tested against more fundamentally different opponents.

References


