

# DETECTING THE COMMUNITY STRUCTURES IN THE GAME OF GO

Yuri Malitsky  
Cornell University

Christopher Fellows  
Cornell University

Gregory Wojtaszczyk  
Cornell University

**Abstract**—Go is a complex board game that stands impervious to the present computational intelligence. This project continues our composite approach, aiming to integrate the strengths of proven heuristic algorithms with AI techniques for boosting performance of Computer Go. In previous research, we explored Support Vector Machine (SVM) supervised training of a move evaluation function based on a collection of expert games. In this paper, we present a graph-based model for describing the board position and an application of Newman-Girvan edge betweenness clustering algorithm for detecting the Go dragons, “communities” of loosely connected stones.

**Key Words:** Clustering, Interaction Network, Computer Go

## 1. INTRODUCTION

WHILE computers are playing many of the classic board games at the grandmaster level, progress in Go remains elusive. The large branching factor in the game makes traditional adversarial search intractable, while the complex interactions of stones prevent the construction of the reliable evaluation function. As a result, most of the existing programs rely on expert-based heuristics and pattern matching techniques. There are also a number of AI approaches aiming to derive generic mathematical models [1]. Yet none of these solutions perform better than an amateur player.

The goal of our research is to introduce a composite approach integrating the strengths of proved heuristic rules, AI-based learning techniques, and knowledge accumulated in expert games. In previous work [2], the effort was directed to replacing a hand-tuned move evaluation function with an adaptable neural network structure trained on a collection of expert games. As a basis, we exercised with the publicly available Gnu Go [3], currently one of top-ranking Go programs, and Fast Artificial Neural Network Library (FANN[4]). The conventional approach with back-propagation however turned out impractical in the Go application environment since the expert moves provided only the maximization conditions rather than absolute values of all possible moves. To step around this problem, we transformed the learning task into an optimization application maximizing the difference between the correct move and all others. In this approach, each game position could potentially have up to 381 moves, resulting in exceedingly large number of equations. This complication

had been resolved with SVM-light [5], a program specifically designed for handling large data sets.

After optimizing a linear kernel on over 7000 equations, the SVM successfully approximated the Gnu Go hand-tuned heuristics, leading to an identical level of play [2]. Further extension of this approach to the unresolved Gnu Go problems however was futile despite various enhancements to the kernel structure. This suggested for us to go down into the engine and explore the lower level algorithms supplying input values for the Gnu Go evaluation function. Analyzing the unresolved endgames [6] with the Cornell Go Club members [7] directed us to the part of Gnu Go associated with the definition of dragons, groups of loosely connected stones. Location and structure of these dragons have a forefront importance in the estimation of strategic and tactical characteristics of possible moves. The groups that Gnu Go finds however are sometimes too large, leading the engine to assume that the related stones are no longer worth saving. To address this problem we follow our composite model and introduce a graph based approach combining Gnu Go’s connection patterns with graph clustering algorithms for detecting the community structures of loosely connected stones.

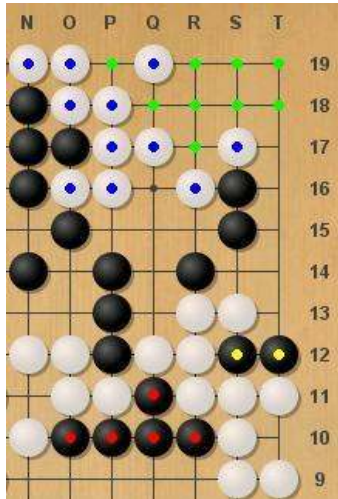
The paper is organized as follows. Section 2 and section 3 provide a brief introduction into the game of Go and the Gnu Go engine. Section 4 introduces the graph based model and application of the Newman-Girvan edge betweenness algorithm. The final section presents the summary and future work.

## 2. THE GAME OF GO

Go is a deterministic, perfect information, zero sum game between two players. The rules are simple, the two players alternate placing black and white stones on a 19x19 grid board trying to surround as much territory (empty intersections) as possible. The only time a stone is removed from the board is when it is completely surrounded by the opposing player’s pieces, like if black plays T13 in Figure 1. The game ends when both players pass on successive turns. However, despite the simplicity of these rules, the game is very difficult to master.

The strategy for Go is centered on surrounding maximal amount of territory with groups of worms, dragons, that are unconditionally alive. The rules of the game state that if any group of stones becomes completely surrounded it is removed from the board. However, if a strongly connected chain of stones (worm, Figure 1) surrounds two independent

empty intersections, the group cannot be killed. This is because even if the chain is surrounded, the opposing player cannot fill both these empty intersection simultaneously, and playing in an intersection that is already surrounded is suicide. These independent empty intersections are called eyes, and any group of stones having two or more eyes is unconditionally alive.







-  - Worm (a group of strongly connected stones)
-  - Dragon (a group of loosely connected worms)
-  - Territory controlled by white dragon
-  - Black stones that will be removed when white plays T13

Fig 1: An example of worms and dragons in a corner of a 19x19 Go board

From a computer standpoint, the problem with Go is two fold. First, because the game is played on such large board the branching factor of the game tree is around 300, which is 10 times that of chess. Because of this, brute force searching is not applicable. Secondly, there is a complex interdependence between the stones and it is not uncommon for a stone on one side of the board to influence a group in the opposite corner. In such situations it is also possible that a single move disrupts the dependence. This volatile interdependence makes it difficult to evaluate a board position, or even determining which of the players is winning. As a result, the best computer programs play only at the level of a moderate beginner.

### 3. THE GNU GO ENGINE

Gnu Go [3] is a publicly available Go program that is currently ranked as the fourth best computer engine. To select the played move, the program follows a multistage approach. First, an inner representation of the board, consisting of the chains of connected stones (worms), is

compiled. These chains are then analyzed by a connection pattern matching algorithm to create the dragons. Once the dragons are established, a collection of *move generators* analyze the board using a mix of combinatorial game theory, life-and-death pattern matching, and expert heuristics. If the generators determine that a move fulfills a certain criteria, like finishing a known combination or attacking an opponent's chain of stones, the move is assigned a corresponding *reason*, for example ATTACK\_MOVE. When the generators are finished, each move's reasons are evaluated to produce twelve *characteristic values*, such as the territorial and strategic value of a move. The values are then evaluated and combined using a set of hand-tuned rules, and the move with the highest resulting value is played.

In our previous research, the SVM approach confirmed that Gnu Go was performing near optimally in its high level calculations of the move values. However, extensive studies of unresolved GnuGo problems suggested that problems are located in lower layers dealing with creating and analyzing of dragons, groups of loosely connected worms. In Gnu Go, the algorithm calculating these dragons is based on pattern matching, and is described in the following pseudo code.

```

Create Dragons()
  Declare all worms individual dragons
  For each board intersection p of the current problem
    For every connection pattern c in database
      For every transformation t of the pattern
        If pattern c anchored at p is satisfied
          If the pattern involves two dragons of the
            same color
            Agglomerate the two dragons

```

The code works by first declaring that every worm is a dragon. It then iterates through every intersection of the board, checking if a pattern exists that could join two previously unconnected dragons. For example, Figure 2 shows the pattern that Gnu Go would use to join the white stones Q17 and R16 from Figure 1.

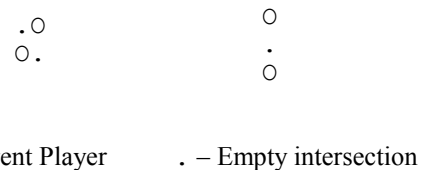


Fig. 2: Example patterns that when rotated 90 degrees connect the worms Q17, R16, and S17 into a dragon

The collection of connection patterns is open and configurable. It is kept in an external text file which is compiled in an array of the corresponding C structures for efficient processing in pattern matching procedure. This approach effectively identifies the groups of associated worms, however, it doesn't take into consideration the degree and strength of each pattern. As a result, it leads to

the formation of overly large dragons, like the white dragon with the red dots in Figure 3. Technically the dragon does exist since the worms in the blue box can be connected by the patterns shown in Figure 2.

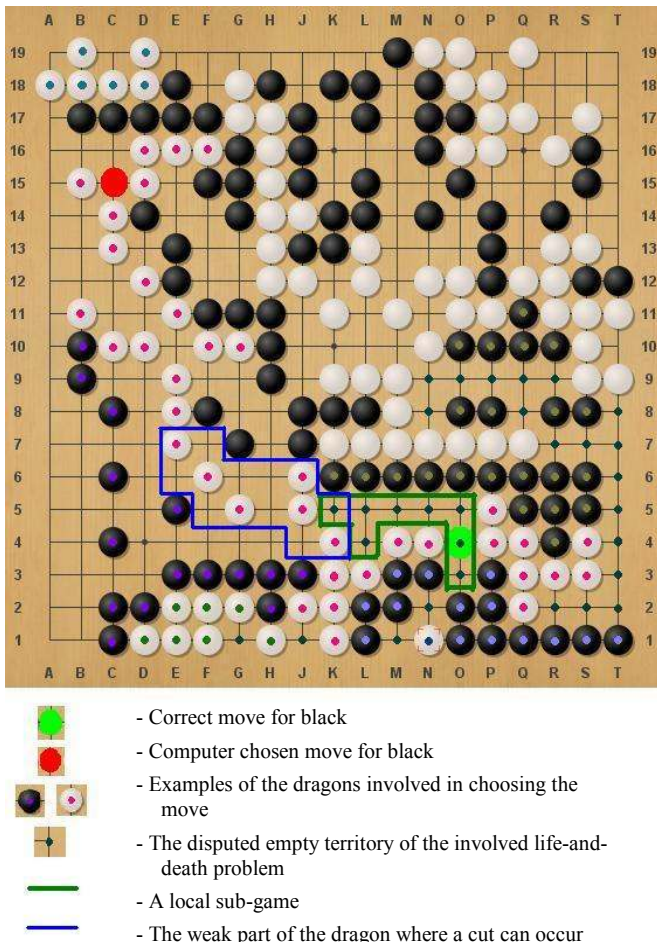


Fig. 3: Example of an unsolved board position and the involved features

What the algorithm fails to notice however is that this dragon has two strongly connected groups of stones joined by a very weak strand of single connections (blue box in Figure 3). Because of this weakness, a Go player [7] would judge the lower right corner of the board as a separate sub-game instead of considering the entire dragon like Gnu Go.

#### 4. GRAPH BASED APPROACH

In its initial analysis of the board, Gnu Go creates dragons that are too large leading to inaccuracies further in the calculations. Identifying the precise weak spots of the dragon however is complicated by the complex structure of the Go engine, which is organized around a flat board container. There are almost a dozen of arrays describing the full board, some keeping track of the worms and dragons, others on whether a given move fits under a cutting pattern that would separate two worms. Such a flexible and extendable

approach however, complicates the development and application of algorithms since all information is spread out and compartmentalized. Therefore we decided to consolidate all the data into a unified graph-based model.

In this model, each worm (strongly connected chain of stones) is represented as a colored node. The color of this node is either black or white depending on which player it belongs to. The size of the node then corresponds to the number of stones in each worm (see Figure 4). The edges in this graph represent all the connection patterns found by Gnu Go for the given board position. Thus, if there are multiple edges linking two worms it means that there are more than one way to connect stones.

For constructing this graph we selected the Java Universal Network/Graph Framework (JUNG [8]), a software library that provides an extensible API for modeling, analyzing, and visualizing the graph data. Figure 4 is the graph representation of the board position described in Figure 3.

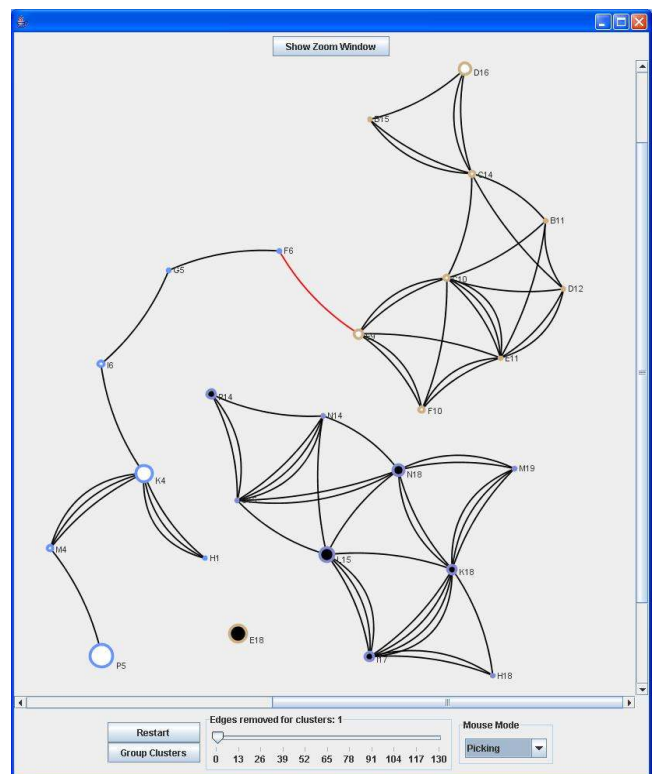


Fig. 4: Graph representation of the red marked white dragon from the board position of Figure 3

Judging from the produced graph we focused on analyzing clustering algorithms for separating the two subgroups of the white dragon. Specifically we considered the three clustering algorithms implemented in the present version of JUNG library: Bicomponent, Weak Component, and Edge Betweenness. Bicomponent Clustering finds the maximal subgroups where all nodes will remain connected, even if one node is removed. This algorithm however is too exclusive to be applied to Go because it is common in the

game for otherwise unconnected groups to link to a single worm a case that should be considered as a dragon. The Week Component clustering approach finds groups where each node is reachable from all nodes in the group. However, since the Go graph is undirected, all nodes communicate, so the algorithm doesn't separate anything.

The Edge Betweenness algorithm, suggested by Grivan and Newman [9], focuses on finding the community structures using the betweenness property:

$$C_B(e) = \sum_{s \neq t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}}$$

The algorithm analyzes the graph  $G = (V, E)$ , and assigns the value  $C_B$  for each edge  $e \in E$ , where  $\sigma_{st}$  is the number of shortest geodesic paths from  $s$  to  $t$ , and  $\sigma_{st}(e)$  is the number of shortest geodesic paths going from  $s$  to  $t$  that pass through edge  $e$ . Therefore the edges that are between two communities have higher values, because there is only one path that connects the nodes of one community to the nodes in the other. Removing this high ranked edge, we get two strong community structures, which could be naturally associated with the dragons in the game of Go.

For the white dragon in Figure 3, the Edge Betweenness algorithm is able to successfully find and separate the two communities, as is shown by the red edge in the graph of Figure 4. This result is also typical in the other unsolved endgame and life and death problems from the Computer Go Test Collection [6]. The Edge Betweenness algorithm is especially applicable since in the cases where the dragons do not have multiple weakly connected communities, removing a single edge does not break apart the dragon, because in these situations there are multiple connections between the involved worms.

## 5. CONCLUSION

As an extension of our previous research, this project continued to tackle the problem of Computer Go. In this paper, we focused on creating a graph based approach combining Gnu Go components with a graph clustering algorithm. Using this model, we were able to visualize the miscalculations predicted by the Cornell Go Club, and then apply the Newman-Girvan edge betweenness clustering algorithm to solve the problem.

Despite the immediate benefits of this application, we consider this model as the groundwork for future development. The next step would be to extend the model with the concepts of eyespace and the values based on the combinatorial game theory suggested by Berlekamp and Wolf [10]. The new structure would then allow for the application of "importance" algorithms, such as HITS and PageRank, to calculate the move values.

## ACKNOWLEDGMENT

We would first like to thank Professor Thorsten Joachims for his continual guidance and support.

We would also like to thank the Cornell Go Club for their deep insight into the wonderful game of Go.

## REFERENCES

- [1] M. Enzenberger. Computer Go Bibliography. [http://www.cs.ualberta.ca/~emarkus/compgo\\_biblio/](http://www.cs.ualberta.ca/~emarkus/compgo_biblio/)
- [2] C. Fellows, Y. Malitsky, and G. Wojtaszczyk. *Exploring GnuGo's evaluation function with a SVM*. AAAI-06 conference, 2006.
- [3] D. Bump, et al. 2005. GnuGo, <http://www.gnu.org/software/gnugo/gnugo.html>
- [4] S. Nissen and E. Nemerson, Fast Artificial Neural Network Library, <http://leenissen.dk/fann>
- [5] T. Joachims, 1999, SVM-light, <http://svmlight.joachims.org>
- [6] M. Mueller, 1995. Computer Go Test Collection <http://www.cs.ualberta.ca/~games/go/cgtc/>
- [7] A. Jackson et. al, Cornell Go Club, private communication.
- [8] J. O'Madadhain, et.al, JUNG: Java Universal Network/Graph Framework. <http://jung.sourceforge.net/>
- [9] M. Girvan and M. E. J. Newman. *Community structure in social and biological networks*. Proc Natl Acad Sci USA. 2002;99:7821–7826. doi: 10.1073/pnas.122653799.
- [10] E. Berlekamp and D. Wolf. *Mathematical Go: Chilling Gets the Last Point*. A K Peters Ltd. May 1997