Life-and-Death Problem Solver in Go

Byung-Doo Lee
Dept. of Computer Science, Univ. of Auckland, New Zealand
blee026@cs.auckland.ac.nz

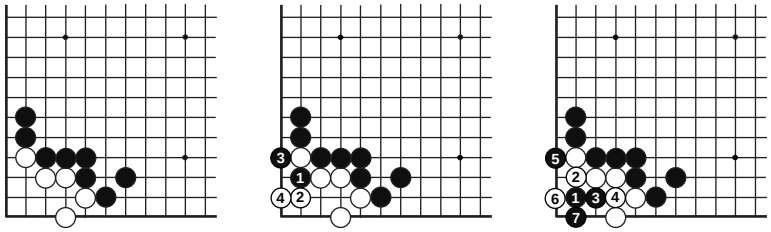
Abstract

The life-and-death problem in Go is a basic and essential problem to be overcome in implementing a computer Go program. This paper proposes a new heuristic searching model which can reduce the branching factor in a game tree. For constructing the first level of a game tree, we implement pattern clustering and eye shape analysis to get the set of the first moves, thereby reducing the branching factor of the game tree. The empirical result for game tree searching with these methods is promising. We also suggest several problems to address for making game tree searching more robust, such as: coping with situations where the number of legal moves in the surrounded group is more than 8, creating an accurate heuristic evaluation function, and dealing with *ko* fighting.

1 Introduction

In the game of Go, the life-and-death problem is a fundamental problem to be overcome when implementing a computer Go program. The life-and-death problem is a local problem in which the attacker tries to kill an opponent's group [19]. In the middle game and the end game, life-and-death problems usually happen as the game unfolds. In the life-and-death problem, there are two main techniques [10, 16] for the defender to make two *eyes* needed to attain life for the surrounded group: enlarging the base or playing at a vital point. Conversely, the attacker also has two main techniques [10, 16] to attack a surrounded group: reducing the base of a group or playing on a vital point. Intuitively most Go players select the first move in a few seconds, which may be a candidate vital move to kill the surrounded group, by comparing it to the similar patterns which they have remembered. And then they play out a sequence of moves to kill the surrounded group.

Figure 1(a) shows a life-and-death problem that does not have a completely defined boundary between the black and white groups (there are gaps at the edges of the board). Each commercial computer Go program has its own veiled algorithms. Most of them use tactical searching, which is selective goal-oriented searching [3]. The simplest method for solving the life-and-death problem is to explore all the possible (legal) moves until all terminal moves are encountered,



(a) (b) (c)

Figure 1: Illustration of the correct and an incorrect move sequence to attack the white group. (a) A life-and-death problem. (b) The white group is alive because of black's incorrect move sequence. (c) The white group is dead because of black's correct move sequence.

and then to determine the best move sequence for both players by a heuristic evaluation function. This approach requires enormous computing time, and thus we cannot use this kind of exhaustive searching in reality.

2 Experimental approach

In the game of Go, similar life-and-death problems (patterns) often have similar solutions, i.e. a similar first move to kill the surrounded group. There are two methods we could use to categorize patterns: pattern classification and pattern clustering. In general, pattern classification is used to classify similar patterns. There is a difference between pattern classification and pattern clustering. Pattern classification is a process of assigning a new input pattern into a predefined cluster (class, group or category), and pattern clustering aims to group the input patterns into different clusters without having any predefined clusters [1, 5, 8]. The drawback of pattern classification is that we could encounter new input patterns that are not members of any predefined cluster. Pattern clustering, however, generates new cluster dynamically when new input pattern cannot be classified into the existing clusters.

We chose pattern clustering method because there were no predefined pattern clusters. With pattern clustering, similar patterns are grouped into the same cluster, which keeps a set of first moves extracted from each pattern. The main reason to apply pattern clustering is to use the cluster's first moves to kill the surrounded group as the candidate first moves. After classifying a new input pattern we can use the cluster's candidate first moves for game tree searching.

Another approach for solving life-and-death problems is to analyze the *eye* shape

of the surrounded group and then to find the vital first move to kill it. An *eye* shape is the shape of the empty board space that is completely surrounded, by stones that are all connected [10]. In life-and-death problems, the *eye* shape of the surrounded group contributes to determining whether the group is alive, dead or unsettled. Alive means that the surrounded group does not need to be defended because it cannot be killed, i.e. it is unnecessary (indeed pointless) to play a stone in the surrounded area to secure (or to attack) the surrounded group. Unsettled is a situation where, if the owner of the surrounded group does not play a stone in the surrounded area, the group can be killed by the opponent. Dead is a situation in which the surrounded group cannot survive, even by playing a friendly stone in the surrounded area. If an *eye* shape cannot be reduced to fewer than seven points, we will assume that the group has secure territory rather than an *eye* shape [10]. The main reasons for analyzing the *eye* shape of the surrounded group are: (1) to find to what extent the surrounded group is dead or unsettled, and then (2) to generate possible vital moves to compensate for the weakness of pattern clustering by adding the possible vital moves to the candidate moves generated by pattern clustering.

Figure 2 shows the two basic steps to getting a sequence of best moves: (1) move generation, and (2) game tree searching. Pattern clustering and *eye* shape analysis were used for the move generation, and game tree searching was used for finding the best sequence of moves in a game tree. When a new pattern is input as a life-and-death problem, the pattern classifier in Figure 2(I) generates a set of candidate first moves from the selected cluster.

For constructing a classification (or clustering) engine, either unsupervised or supervised learning may be used. Since Kohonen's self-organizing maps have become a promising clustering technique [1, 12], we implemented Kohonen Neural Network (KNN) based clustering for unsupervised learning. For supervised learning, Euclidean distance based clustering and vector product based clustering were implemented.

Before analyzing the *eye* shape of the surrounded group, the *eye* shape analyzer in Figure 2(II) had to create a virtual boundary to distinguish the surrounded group from the surrounding group. Contour tracing with $influence$ was used for creating a virtual boundary, and then the $influence$ function calculated the $influence$ value at each point in the surrounded group. Finally, the $influence$ value determined the *eye* shape of the surrounded group to determine whether the group was alive, dead or unsettled. When the surrounded group was unsettled, the *eye* shape analyzer generated a set of possible vital moves as the first moves to kill the surrounded group.

Figures 1(b) and 1(c) show how the first move and the sequence of moves are important to solve the life-and-death problem in Figure 1(a). Two first moves, such as black 1 in Figures 1(b) and 1(c), may be generated by the pattern classifier and the *eye* shape analyzer. Based on the generated first moves, finding

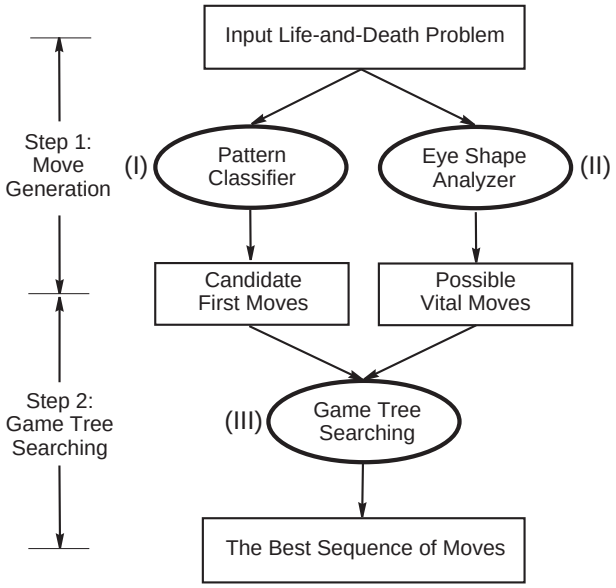


Figure 2: Basic components of a local problem solver.

the move sequence in the game tree is the role of game tree searching in Figure 2(III). As Minimax searching with α-β pruning has been used as a standard searching algorithm for two-person zero-sum games with perfect information [2], Negamax searching was used for finding the sequence of best moves for both players, and the α-β pruning algorithm was used for cutting off nodes in a game tree.

3 Pattern classifier

Move generation is a part of a computer Go engine that mainly includes board representation, position (move) evaluation and searching. We explained that the first move and the next move sequence are very important when attacking a surrounded group in life-and-death problems. To generate a set of first moves, we implemented three learning methods and then compared them to find which method is the most suitable for solving life-and-death problems.

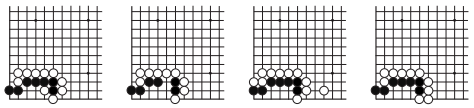
Clustering is a prior step to classifying data, and aims to group (cluster) the input data into sets that have strong internal similarity [11]. Clustering has been emphasized as an important part in data mining, machine learning, computer vision, and other engineering disciplines. Different clustering methods analyzing the same data set can yield very different results. Furthermore, a clustering algorithm which works well on one specific data set may have very poor results

with another data set [14, 20].

We implemented Euclidean distance based clustering and vector product based clustering for supervised learning, and KNN based clustering for unsupervised learning. Each clustering method was applied to the original data set and the transformed data set generated by Principal Component Analysis (PCA). This is because the transformed data set, rather than the original data set, has been used extensively to measure the similarity and the dissimilarity among the input patterns [9].

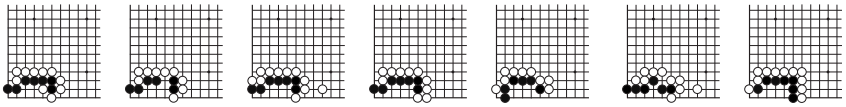
3.1 Euclidean distance based clustering

As a supervised learning method, Euclidean distance based clustering was applied. The basic idea is to calculate distances between the input pattern and the weighted center of each cluster, and then find the closest cluster within the range of the threshold ρ. Figures 3 and 4 show that the accuracy of clustering is dependent on the threshold ρ.



(a) (b) (c) (d)

Figure 3: A pattern set clustered by Euclidean distance based clustering ($\rho = 3$).



(a) (b) (c) (d) (e) (f) (g)

Figure 4: A pattern set clustered by Euclidean distance based clustering ($\rho = 9$).

For instance, Figure 4(f) slightly differs from the patterns in Figure 3. That is, with a low threshold, the extent of similarity within each cluster is high and the input patterns are likely to be split into a large number of different clusters. On the other hand, with a high threshold, the extent of similarity is low and the input patterns gather into a few different clusters.

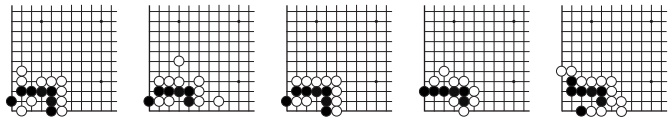
We tested the Euclidean distance based clustering method with the original input data set ($N = 588$) and its transformed data set, and then found the

number of clusters is 196 for the original set and 201 for the transformed set ($\rho = 11$). We also found the percentage of the patterns clustered differently by each data set is less than 3% when $\rho = 11$.

3.2 Vector product based clustering

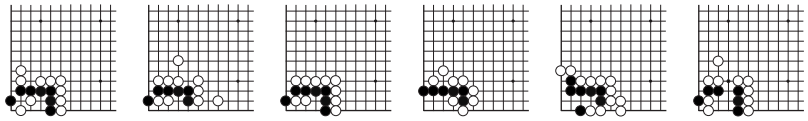
As a non-distance metric and a supervised learning model, the vector product (centroid) based clustering method was implemented. The method for computing similarity is similar to the Euclidean distance based clustering method. The similarity degree ($\cos\theta$) between an instance vector and the centroid vector of each cluster measures the similarity between two vectors. That is, if $\cos\theta = 1$, then the two vectors are exactly the same. Meanwhile, if $\cos\theta$ = -1, then the two vectors are totally different.

Like Euclidean distance based clustering, we applied vector product based clustering to the original data set and the transformed data set. The number of clusters with the original data set was 202 ($\cos\theta = 0.65$) and 220 for the transformed data set ($\cos\theta = 0.55$). Figure 5 shows the clustered patterns in a cluster from the original data set and Figure 6 from the transformed data set.



(a) (b) (c) (d) (e)

Figure 5: A pattern set clustered from the original data set ($\cos\theta = 0.65$).



(a) (b) (c) (d) (e) (f)

Figure 6: A pattern set clustered from the transformed data set ($\cos\theta = 0.55$).

From Figures 5 and 6, we can see the outcome of vector product based clustering is very similar for the two data sets, and vector product based clustering is sensitive to the threshold (the similarity degree), as in Euclidean distance based clustering.

3.3 Kohonen neural network based clustering

As an unsupervised learning method, we used the Kohonen Neural Network (KNN) which is based on the concept of self-organizing maps (SOM) and retains the topology of a multidimensional representation within a two-dimensional (plane) array of neurons [5, 21]. The basic idea of the KNN is to select the winning neuron, which is the neuron in the active output layer that is the closest to the input vector. Then the weights of the winning neuron and its neighboring neurons are updated inversely proportional to their distance from the winning neuron.

To apply the KNN, we constructed a two-layered network: an input layer and an output layer. In the input layer, we provided two sets of 121 attributes, representing black and white stones. For the output layer, we constructed 900 neurons (in a 30×30 grid) so that it was large enough to cover the worst situation where each of the input patterns is in a different cluster. We split the original 588 patterns into the 500 training patterns (the actual number of the training patterns was 1,000 by reversing the colors) and the 88 validation patterns for performing the cross validation. We trained and generalized the KNN with an early stopping method using cross validation to avoid overfitting to the training set. After 190 epochs, the 1,000 training patterns were clustered into the 219 different clusters.

We classified 88 validation patterns with the trained network to get a set of the first moves. Among the 88 patterns, 78 patterns (88.7%) were classified into 25 different clusters created from the training patterns and 10 patterns (11.3%) were not classified. The main reason why there were unclassified patterns was that the domain size (the 11×11 attributes) was to some extent too large. We then checked to what extent the 78 classified patterns had correct first moves, which were extracted from the first move set retained in the clusters, as first moves for attacking the surrounded group. The accuracy of selecting the first move for the test patterns was 62.8% as shown in Table 1. We compared this result with the experimental results found by Sasaki *et al.* [19] as shown in Table 1. They trained a supervised neural network with the 2,000 patterns and then extracted a set of first moves for each of 1,000 test patterns. The resulting accuracy (34.9%) in Table 1 is the value when we only consider the highest output value in the output layer. When we consider the first 5 highest output values, the resulting accuracy is 65.0%.

Table 1 shows that KNN based clustering is a promising method for picking first moves for solving life-and-death problems. We found that (1) the KNN as unsupervised learning ideally finds a set of first moves if we train the network with plentiful training patterns, and (2) the pattern clustering method with the KNN can compete with the pattern matching method performed by supervised learning with a neural network.

Network type	Back-propagation neural network	KNN
Domain size	9×9	11×11
Learning type	Supervised	Unsupervised
Applied methodology	Pattern matching	Pattern clustering
Number of training data	2,000	1,000
Number of test data	1,000	88
Accuracy of picking the correct first moves	34.9% (first answer only)	62.8%

Table 1: Comparison of the performance between supervised learning with a neural network conducted by [19] and unsupervised learning with a KNN.

Using the original 588 input patterns, we then compared the accuracy of KNN based clustering with the result of Euclidean distance based clustering. Since Euclidean distance based clustering is dependent on the threshold value ρ, we set the threshold value to 15, which generated 191 clusters. This number of clusters was similar to the number of clusters (194) created with 25×25 output neurons in the output layer of the KNN. Figures 7 shows the similarity of the clustered patterns in a cluster generated by KNN based clustering. We cannot see a clear similarity among the clustered patterns. Since there were some patterns that should not be clustered in the same cluster, we conclude that KNN based clustering is, to some extent, not adequate to be applied to generating the candidate first moves.

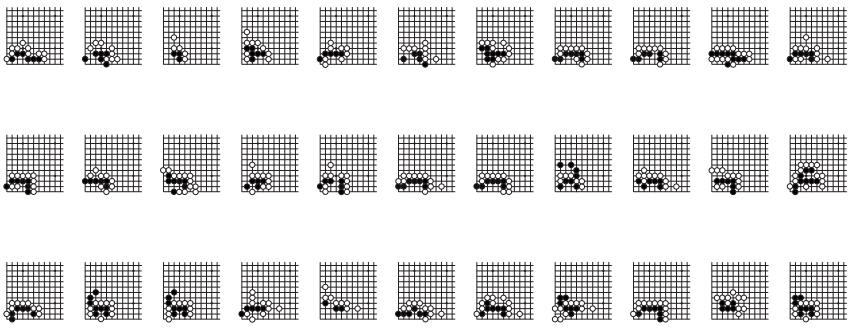
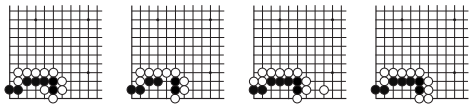


Figure 7: A pattern set clustered by the KNN on (25×25) output neurons.

3.4 Experimental results in the pattern classifier

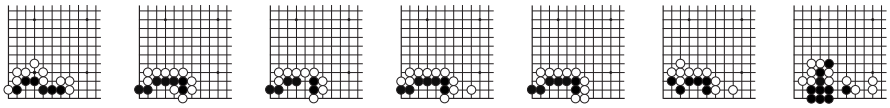
For solving even a local Go problem, an important consideration is how to tackle the huge branching factor and the depth in the game tree. We need a heuristic method to tackle these problems. Since similar patterns have a similar first move in life-and-death problems, clustering (or classification) is a useful method for reducing the initial branching factor. The set of first moves, which are recognized by the pattern classifier, contributes to solving life-and-death problems. Our goal was to find to what extent the pattern clustering methods are able to be applied in the real game of Go.

Our empirical result showed that there is no real advantage in using KNN based clustering; rather, Euclidean distance clustering is more suitable when the threshold value is reasonably adjusted. For example, Figure 8 shows a very similar pattern set clustered by the Euclidean distance clustering method, whereas the three patterns in Figures 9(a), 9(f) and 9(g) have little similarity in the cluster created by the KNN based clustering method.



(a) (b) (c) (d)

Figure 8: A pattern set clustered by Euclidean based clustering with the transformed data set ($\rho = 3$).



(a) (b) (c) (d) (e) (f) (g)

Figure 9: A pattern set clustered by KNN based clustering with the transformed data set (projected on 50×50 output neurons).

In conclusion, when pattern clustering method is applied for solving life-and-death problems, it is recommended that Euclidean distance based clustering be used with a low threshold value (e.g. ≤ 3) and with transformed data sets generated by PCA.

4 Eye shape analyzer

We applied a novel method, *eye* shape analysis with a heuristic $influence$ function, to generate the candidate first moves. The main objective of analyzing the *eye* shape of a surrounded group with a heuristic $influence$ function was to find in a short time whether the surrounded group is alive, dead or unsettled. If there is a high possibility that the surrounded group is dead or unsettled, then we can generate the possible vital points (moves) to save enormous amounts of computing time in searching for the best sequence of moves in the complete game tree.

4.1 Eye shape

A typical Go board can be represented as a 19×19 array of intersections. Each intersection is a vertex (from now on called a point) of the board graph. There is an edge which connects two neighboring intersections vertically or horizontally (not diagonally). That means a point cannot have more than 4 edges. When two points are connected by an edge, we call them *neighbors*, which share $influence$. The arrangement of the set of empty neighbors in a life-and-death problem shows the *eye* shape of the board graph, and can be represented as a quadruple (a,b,c,d), where a is the number of points with 4 neighbors, b for 3 neighbors, c for 2 neighbors, and d for 1 neighbor [17]. In Figure 10, the points A and B have one neighbor, the point C has 4 neighbors, the points D, E and F have 2 neighbors, and thus the graph is represented by (1,0,3,2).

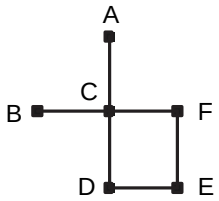


Figure 10: Neighbors in a graph. The points A, B, D and F are neighbors of the point C, E is not.

The set of neighbors determines whether the *eye* shape is alive, dead or unsettled. In life-and-death problems, the numbers of points of concern are 3, 4, 5 and 6. An *eye* shape which has less than 3 points is unconditionally dead. An *eye* shape of more than 6 is alive when all stones are solidly connected [10]. Figure 11 shows *eye* shapes categorized by the number of points: three-point, four-point, five-point and six-point.

We represented the *eye* shape of the surrounded group as (a,b,c,d:e), where a indicates the number of points with 4 neighbors, b for 3 neighbors, c for 2 neigh-

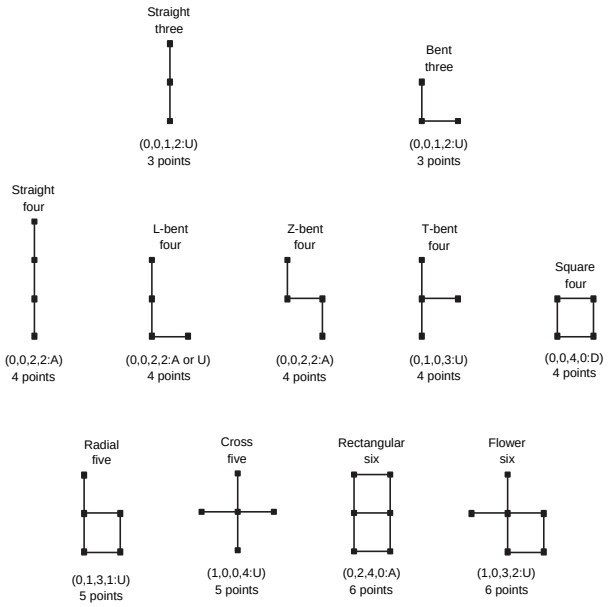


Figure 11: The main *eye* shapes concerning life-and-death problems with the name (above) assigned to each. The notation (a,b,c,d:e) shows the number of points which have specific numbers of neighbors and the life-and-death status of the shape.

bors, and d for 1 neighbor. And e is represented by A when the *eye* shape is alive, D for dead, and U for unsettled. Table 2 shows the relationship between the number of points, the number of types, and the life-and-death status of the *eye* shapes.

Generally, the status of three-point *eye* shapes is unsettled, and the four-point *eye* shapes are all alive except the *T-bent four* (unsettled) and the *square four* (dead). Meanwhile, in the five-point *eye* shapes, all except the *radial five* (unsettled) and the *cross five* (unsettled) are alive. All six-point *eye* shapes are alive except the *flower six* (unsettled). All *eye* shapes having seven or more points are alive [10]. In four-point *eye* shapes, there is one exception when determining the life-and-death status of an *eye* shape: *L-bent four*. If the *L-bent four* is formed in the corner of a Go board, the shape is an unsettled shape.

4.2 Heuristic influence function

When a group of stones is surrounded, we assume that the surrounding group radiates strong $influence$ into the open surrounded area, and meanwhile the surrounded group radiates very weak $influence$ to the surrounding group. That means $influence$ of the surrounding group is not diminished by the surrounded group, except in some special cases (e.g. a point which becomes an *eye* using

Number of points	Number of types	Representative *eye* shapes
1	1	(0,0,0,0:D)
2	1	(0,0,0,2:D)
3	1	(0,0,1,2:U)
4	3	(0,0,2,2:A) (0,0,4,0:D) (0,1,0,3:U)
5	4	(0,0,3,2:A) (0,1,1,3:A) (0,1,3,1:U) (1,0,0,4:U)
6	8	(0,0,4,2:A) (0,1,4,1:A) (0,1,2,3:A) (0,2,0,4:A) (0,2,2,2:A) (0,2,4,0:A) (1,0,1,4:A) (1,0,3,2:U)

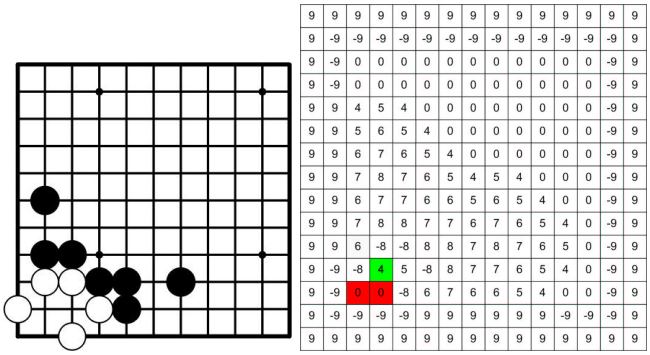
Table 2: *Eye* shapes varying with the number of points.

a *tiger's mouth*, which is a set of three stones of the same color in a V-shape around an empty point).

When solving life-and-death problems, we did not consider the connection problem between the surrounded group and other allied groups nearby (to which it might be able to connect in order to live). We also did not consider life-and-death problems where the surrounded group already had an *eye*. Figure 12(b) shows the resulting $influence$ of the life-and-death problem in Figure 12(a). Figure 12(c) shows the calculated *eye* shape of the surrounded group, (0,0,1,2:U). Figure 12(d) illustrates the best move sequence; black 1 is the vital point of the *eye* shape in Figure 12(c), to attack the surrounded group.

There are two steps for finding the *eye* shape of the surrounded group: (1) calculating $influence$ of the surrounding group and then (2) calculating the number of neighbors of the zero $influence$ points. Firstly, the virtual boundary is necessary to completely surround the surrounded group, because the surrounded group is loosely enclosed by the surrounding group. Initially, we set the outer boundary to 9 for the surrounding group and to -9 for the surrounded group, as in Figure 12(b). We then calculated $influence$ from the surrounding boundary, which includes the surrounded stone group and the virtual boundary, to the surrounded area using the contour tracing method. The contour tracing method is a technique for finding boundaries in digital images and is also known as border tracking. As an algorithm, we used the radial sweep algorithm [6], which is similar to the Moore-Neighbor tracing algorithm.

We tested to what extent the heuristic $influence$ function can correctly determine the life-and-death status of 30 problems in [13]. Table 3 summarizes the calculated *eye* shapes compared to the solutions given in [13]. The results of the calculated *eye* shapes are classified as alive, unsettled and dead. The result (36.7% correct) shows that the *eye* shape analyzer with a simple heuristic $influence$ function can contribute to quickly finding the life-and-death status of the surrounded group.



9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	9
9	-9	0	0	0	0	0	0	0	0	0	0	0	0	-9	9
9	-9	0	0	0	0	0	0	0	0	0	0	0	0	-9	9
9	9	4	5	4	0	0	0	0	0	0	0	0	0	-9	9
9	9	5	6	5	4	0	0	0	0	0	0	0	0	-9	9
9	9	6	7	6	5	4	0	0	0	0	0	0	0	-9	9
9	9	7	8	7	6	5	4	5	4	0	0	0	-9	9	
9	9	6	7	7	6	6	5	6	5	4	0	0	-9	9	
9	9	7	8	8	7	7	6	7	6	5	4	0	-9	9	
9	9	6	-8	-8	8	8	7	8	7	6	5	0	-9	9	
9	-9	-8	4	5	-8	8	7	7	6	5	4	0	-9	9	
9	-9	0	0	-8	6	7	6	6	5	4	0	0	-9	9	
9	-9	-9	-9	-9	9	9	9	9	9	9	-9	-9	-9	9	
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	

Figure 12: The *influence* map and the calculated *eye* shape. (a) A life-and-death problem. (b) Generated *influence* values with the virtual boundary. The black stones are represented as 8 and white stones as -8. Generated *influence* values are represented as between 7 and 4. The set of points with zero *influence* in the surrounded group determines the formation of the *eye* shape. (c) The number of neighbors at each point within the *eye* shape. (d) The actual move sequence to kill the surrounded group as the solution.

Correct status (number of problems)	Calculated *eye* shape status (number of problems)	Percentage correct
Alive (1)	Unsettled (1)	0.0%
Unsettled (29)	Alive (14), Unsettled (11), Dead (4)	37.9%
Total (30)	Correct answer (11)	36.7%

Table 3: Results of applying *eye* shape analysis to 30 life-and-death problems. Calculated *eye* shapes were the shapes calculated by the heuristic *influence* function.

4.3 Experimental results in the eye shape analyzer

We analyzed the *eye* shape in surrounded groups to determine the life-and-death status of each group. With a heuristic $influence$ function, we tested 30 life-and-death problems with problem domain sizes of about 10 or less, and where the boundaries between the surrounded and surrounding groups were not completely blocked. We found that from the calculated *eye* shape, which is the set of points having zero $influence$ in the surrounded group, we can determine correct point to play to capture the surrounded group with 36.7% accuracy.

That is, the *eye* shape analyzer with a heuristic $influence$ function can quickly assess the life-and-death status of the surrounded group to some extent. The vital points generated by the *eye* shape analyzer contribute to a set of possible correct moves to reduce the branching factor in a game tree, and the drawback (low accuracy) of the *eye* shape analyzer is compensated by the pattern clustering method.

5 Game tree searching

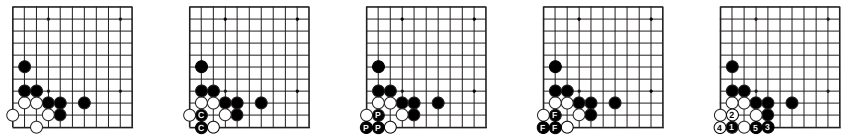
A game tree is composed of a set of all legal moves in a hierarchical data structure. To find the best sequence of moves, one method is to search all nodes in the entire search domain. Another method is to analyze the situations to a limited depth in the tree, to reduce the enormous computing resources (e.g. computing time and memory space). But this method has no guarantee of tackling the horizon effect such as the *ladder* problem in Go.

In general, even local Go problems (such as simple life-and-death problems) need a complicated searching method to find an accurate solution. We tried to solve life-and-death problems using a conventional searching method, Negamax searching with α-β pruning. The important matters to consider in game tree searching are deciding (1) where to start, (2) when to stop, (3) how to evaluate and (4) how to search the game tree.

5.1 Where to start

Human players select a few possible vital points in a few seconds by intuition, which is learnt from experience. And then they deeply examine the best sequence of moves for both players based on the possible vital point.

Using the pattern clustering and the *eye* shape analysis, we can extract a set of first moves. The set of first moves is composed of both candidate first moves from pattern classification and possible vital moves from *eye* shape analysis, and contributes to reducing the branching factor at the first level in the game tree. Note that we applied the set of first moves only at the first level, not throughout the entire depth of the tree. Figure 13 shows a life-and-death problem and a set of first moves generated by the pattern classifier and the *eye* shape analyzer.



(a) (b) (c) (d) (e)

Figure 13: Illustration of a life-and-death problem and a set of first moves. (a) A life-and-death problem. (b) Candidate first moves. (c) Possible vital moves. (d) A set of first moves for game tree searching. (e) The correct sequence of moves to kill white's group.

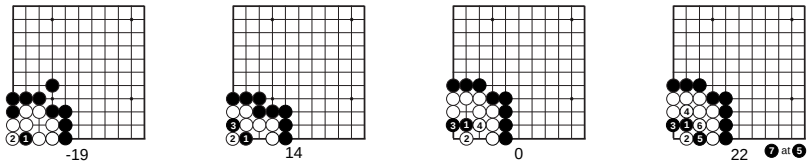
The pattern classifier compares an input pattern (a life-and-death problem) with the representative pattern of each cluster, and then classifies the input pattern into the cluster which has the closest similarity with the input pattern. Figure 13(b) shows the two points which are the candidate first moves in the selected cluster. Meanwhile, the *eye* shape analyzer generates a set of possible vital moves. Figure 13(c) shows the three points which are the set of possible vital moves generated by the heuristic $influence$ function. The combined set of candidate first moves and possible vital moves becomes a set of first moves in Figure 13(d), for solving the life-and-death problem in Figure 13(a). Figure 13(d) illustrates the set of first moves to create the nodes in the first level of the game tree. These moves contribute to reducing computing time and memory space in searching and generating the game tree.

5.2 When to stop

When playing Go, there are two restrictions on placing stones on the board: *suicide* and *ko*. *Suicide* does not allow one to play on a point which causes an ally string to have no *liberty* (except the situation where it captures one or more stones by playing at that point). *Ko* means you cannot instantly put a stone on the point where the opponent player has just captured a single stone. After playing a stone elsewhere, you can return to the *ko* to capture that stone [7]. To generate the nodes in the game tree, we applied these two basic restrictions to explore all the possible (legal) next moves.

Furthermore, if the *eye* shape of a surrounded group had more than 6 points or two *eyes*, we stopped generating nodes and regarded the current node as a terminal node. Meanwhile, if the points in the *eye* shape were between 3 and 6 points, we analyzed the *eye* shape and then generated nodes with all legal moves until encountering terminal conditions as: alive, *ko*, *seki* and dead. Note that we assumed that black plays first. In Figure 14, we can see four terminal

conditions that stop the generation of nodes, with the numbered stones denoting the move sequence for both players.



(a) White alive (b) *Ko* (c) *Seki* (d) White dead

Figure 14: Illustration of terminal conditions representing terminal nodes in a game tree. The evaluation value of each terminal node is illustrated below the Go board from black's perspective.

5.3 How to evaluate

The next matter for managing a game tree is to create an evaluation function for evaluating terminal nodes. We categorized the results of the terminal *eye* shapes as alive, *ko*, *seki* and dead. Ko is difficult to evaluate in computer Go. We could not evaluate the absolute outcome of *ko*. Instead we stopped generating next nodes when we encountered *ko*, and then calculated the evaluation value of current terminal node. That is, we did not calculate the outcome of retaining the initiative (*sente*) or losing the initiative (*gote*) resulting from the *ko*. The evaluation function is extremely important in searching for the best sequence of moves in a game tree. If the evaluation value is not accurate, we cannot expect to find the best sequence of moves for solving life-and-death problems.

Since there is no evaluation function for evaluating the board position, we set a simple evaluation function that only depends on the size of territory and the number of captured stones. Let us assume that black surrounds a white group and that the white group finally dies. Then we calculate the evaluation value of a terminal node with the following evaluation function:

$$B_{eval} = D_W \times 2 + E + C_W - C_B$$

where B_{eval} means the evaluation value to black, D_W is the number of dead white stones remaining on the board, E is the number of empty points in the surrounding group at the terminate node, and C_W and C_B denote the number of captured white and black stones, respectively.

Meanwhile, if the white group finally lives, the evaluation value to black with

the evaluation function as follows:

$$B_{eval} = -A_W \times 2 - E + C_W - C_B$$

where A_W is the number of alive white stones in the surrounded group.

In Figure 14, evaluation values are illustrated below each diagram, and are shown as positive for black and negative for white.

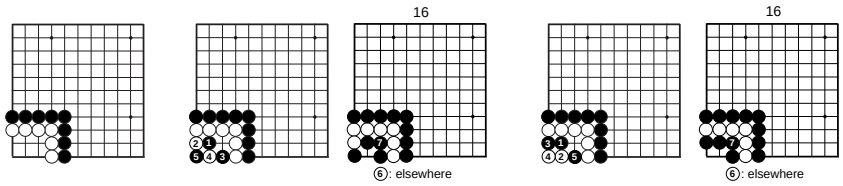
5.4 How to search

Game tree searching works by taking the root node (or usually the current node) and then generating all possible child nodes by applying legal moves until encountering terminal conditions. The terminal nodes generated by terminal conditions are each assigned an evaluation value, based upon an evaluation function, and then these values are filtered back up the tree toward the root node to find the best sequence of moves for both players [4]. The simplest way to search the game tree is the Minimax searching method. This method searches all terminal nodes in a game tree, calculates all resulting evaluation values, and then uses these values to work back from the terminal nodes to the root node. The number of nodes to explore in a game tree increases exponentially by the number of legal moves and the depth. Without applying a pruning method, it is intractable to apply it even in a small local domain.

Minimax searching without pruning has the disadvantage of needing to examine all nodes in a game tree. For solving a simple life-and-death problem, Figure 15(a) for instance, which has 6 legal moves, we have to generate approximately b^d (i.e. 6^6=46,656) nodes where b and d are the number of legal moves and the depth, respectively. That is, the size of the tree will grow enormously when the search domain (the number of legal moves) increases. For solving life-and-death problems, we need very deep searching to find a very accurate solution. Minimax searching without pruning does not allow very deep searching because there is a memory space problem in the computer system caused by the extremely high branching factor and depth. That means we need a pruning method to reduce the branching factor and depth. We thus used α-β pruning which does not affect the accuracy of naive Minimax (or Negamax) searching.

5.5 Experimental results in game tree searching

So far, we have seen four important matters to consider for dealing with the game tree: where to start, when to stop, how to evaluate and how to search. That is, for handling the game tree, we generated a set of first moves for where to start, set terminal conditions for when to stop, built up an evaluation function for how to evaluate, and used Negamax searching with α-β pruning for how to search.



(a) (b) (c)

Figure 15: Illustration of a life-and-death problem and move sequences. The value above the Go board denotes the evaluation value. (a) A life-and-death problem. (b) An incorrect move sequence and the evaluation value generated by game tree searching. (c) The correct move sequence and the same evaluation value.

Table 4 gives a performance comparison between searching with candidate first moves and searching with the entire set of next moves, when the initial number of legal moves in the life-and-death problem is 6. We can see that the game tree searching with candidate first moves is better than the game tree searching without candidate first moves, in terms of computing time and the number of generated nodes.

Comparison items	With candidate first moves	Without candidate first moves
Computing time (seconds)	< 1	≈ 3
Generated nodes	614	3,574
Pruned nodes	608	3,568
Existing nodes	6	6

Table 4: Comparison of the searching performance with and without candidate first moves after applying the pruning method to Figure 15(a).

For life-and-death problems that had a complete boundary between black and white, the generated move sequence was almost correct, except that it did not deal with *ko* fighting. Meanwhile, the game tree searcher did not generate a correct sequence of best moves for problems that had an incomplete boundary between black and white. We realized how important the evaluation function is, and that dealing with *ko* fighting is also important to pick the correct move sequence in life-and-death problems. Figure 15(b) shows an example of an incorrect move sequence generated by the game tree searcher. Most cases of finding an incorrect move sequence were caused by an inaccurate evaluation function.

6 Conclusion

If there are b legal moves and the depth is d in the problem domain, the state-space complexity for game tree searching is roughly $O(b^d)$ [15, 18]. Without initially reducing the branching factor, generating an entire game tree and applying a brute-force searching method is intractable for solving life-and-death problems. Furthermore, if the number of legal moves in a life-and-death problem domain is greater than about 8 or 9, the problem of searching the game tree gets considerably more onerous.

To reduce the branching factor at the first level of the game tree, we implemented pattern clustering and *eye* shape analysis to get a set of first moves for game tree searching.

- Firstly, the basic idea of pattern clustering is that similar patterns have similar solutions (first moves). We suggest implementing Euclidean distance based clustering (a supervised learning method) with the transformed data set rather than KNN based clustering (an unsupervised learning method) for solving life-and-death problems.
- Secondly, the *eye* shape in the surrounded group determines whether the surrounded group is alive, dead or unsettled. To get the *eye* shape in the surrounded group, we implemented the *eye* shape analysis method with a heuristic $influence$ function. We showed that possible vital moves generated by the *eye* shape analyzer contribute to finding the life-and-death status of the surrounded group in a short time.
- Finally, we generated the game tree with the first moves generated by pattern clustering and *eye* shape analysis. During game tree generation, we applied terminal conditions and Negamax searching with α-β pruning to cut off the nodes in the game tree, and then found a sequence of moves for both players to solve life-and-death problems.

When there is a complete boundary between black and white, the sequence of moves generated for both players was almost always correct, except that it did not deal well with *ko* fighting. Meanwhile, game tree searching usually did not generate a correct sequence of moves for life-and-death problems that had an incomplete boundary between black and white. We suggest several problems to address for making game tree searching more robust, such as: (1) coping with situations where the number of legal moves in the surrounded group is more than 8, (2) creating an accurate heuristic evaluation function and (3) dealing with *ko* fighting.

References

[1] K. Asanobu, *Data Mining for Typhoon Image Collection*, Proc. of the Second International Workshop on Multimedia Data Mining, 2001.

[2] A. D. Bruin and W. Pijls, *Trends in Game Tree Search*, Conference on Current Trends in Theory and Practice of Informatics, pages 255-274, 1996.

[3] J. Burmeister, *Studies in Human and Computer Go: Assessing the Game of Go as a Research Domain for Cognitive Science*, PhD thesis, University of Queensland, 2000.

[4] R. Cant and J. Churchill and D. Al-Dabass, *Using Hard and Soft Artificial Intelligence Algorithms to Simulate Human Go Playing Techniques*, International Journal of Simulation, vol 2, no 1, pages 31-49, 2001.

[5] K. Das and W. Perrizo and Q. Ding, *Using Neural Networks for Clustering on RSI Data and Related Spatial Data*, Proc. of International Conference on Information Reuse and Integration, 2000.

[6] A. G. Ghuneim, *Contour Tracing*, Retrieved from http://www.cs.mcgill.ca/∼aghnei on 28 January 2004.

[7] A. Hollosi and M. Pahle, *Sensei's Library*, Retrieved from http://senseis.xmp.net/ on 14 January 2004.

[8] M. Jeon and H. Park and J. B. Rosen, *Dimension Reduction Based on Centroids and Least Squares for Efficient Processing of Text Data*, Proc. of SIAM Conference on Data Mining, 2001.

[9] I. T. Jolliffe, *Principal Component Analysis*, Springer-Verlag, 1986.

[10] J. Kim and S. Jeong, *Learn to Play Go: Battle Strategy*, Good Move Press, 1997.

[11] A. Kitamoto, *Data Mining for Typhoon Image Collection*, Proc. of the 2nd International Workshop on Multimedia Data Mining, pages 68-77, 2001.

[12] T. Kohonen, *The Self-Organizing Map*, Proc. of the IEEE, vol 78, no 9, pages 1464-1480, 1990.

[13] C. Lee, *Chang-Ho Lee's Conventional Baduk 8: For Elementary Life-and-Death Problems*, Samho Media, 1998.

[14] J. Mao and A. K. Jain, *A Self-Organizing Network for Hyperellipsoidal Clustering (HEC)*, IEEE Transactions on Neural Networks, vol 7, no 1, pages 16-29, 1996.

[15] J. Matthews, *generation5*, Retrieved from http://www.generation5.org/ on 28 January 2004.

[16] Y. Moon, *Discovery of Go: Understanding of Modern Go*, Booki Press, 1998.

[17] M. Müller-Prove, *Lebende Blöcke Beim Go: Ein Formaler Ansatz unter Verwendung von Petri-Netzen*, Technical report, University of Hamburg, 1995.

[18] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[19] N. Sasaki and Y. Sawada and J. Yoshimura, *A Neural Network Program of Tsume-Go*, Computer and Games 1998, pages 167-182, Springer-Verlag, 1999.

[20] K. Yeung and W. Ruzzo, *An Empirical Study on Principal Component Analysis for Clustering Gene Expression Data*, Bioinformatics, vol 17, no 9, pages 763-774, 2001.

[21] Z. Zupan and Z. Gasteiger, *Neural Networks in Chemistry and Drug Design*, Wiley-VCH, 1999.