Go Game in Computer Science

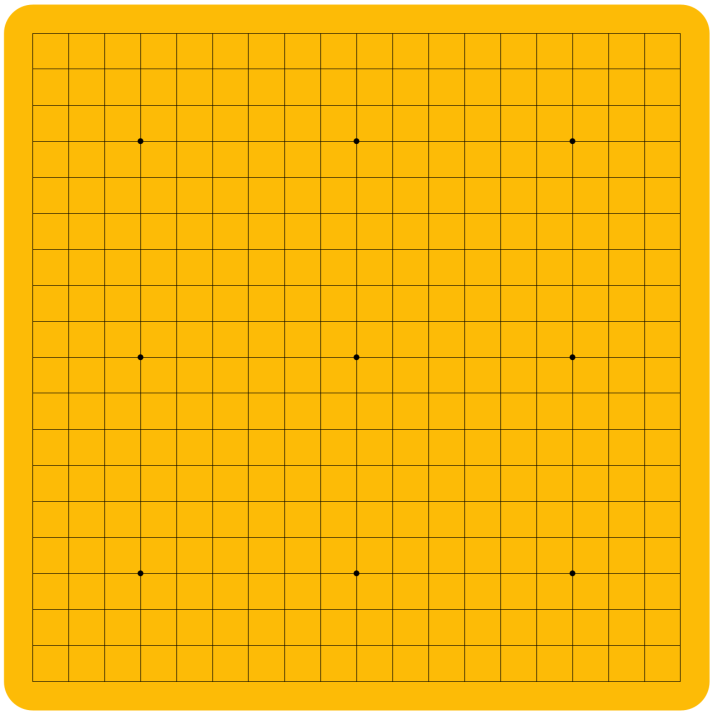
Your enemy's key point is your own key point.

1 Goal

The goal of this paper is to dive into the algorithm of endgame decision in a Go game(evaluation of winning/losing status in the endgame) using graph theory. Furthermore, this paper also dives into potential conjectures of how territorial estimations are carried out, and further research topics to create a better mathematical model.

2 What is Go?(A quick puzzle)

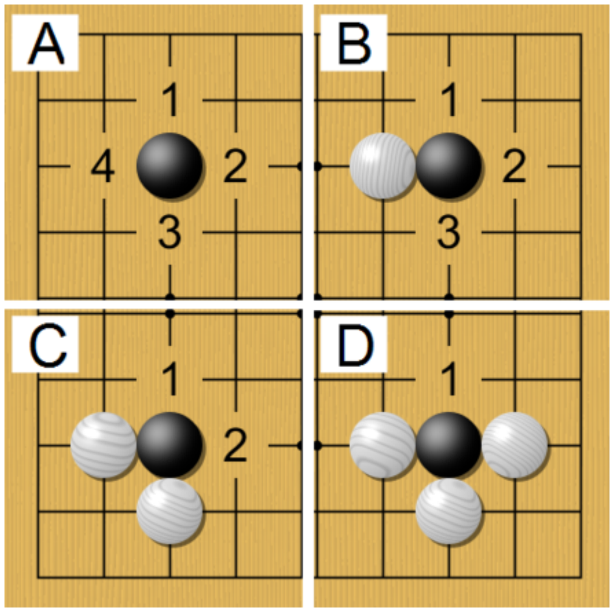
Below is a standard 19×19 Go board, which has 361 crosspoints. Each player will be able to play on a cross points. Mathematically, Go game is complex due to its tremendous amount of variations. In fact, there are 3^{361} possible outcomes of a Go board because each cross point can be played by a black stone, a white stone, or no one plays. Each outcomes are independent from the other ones. Thus, multiply together all the crosspoints gives 3^{361} variations.



3 Basic rules of Go

Before we dive into the mechanisms of my project, it's a better idea to introduce what Go is. Go is an extremely complex game that involves strategic thinking. Each point on the board that depicts the intersection of two lines is called a "cross-point", which are the points a stone is allowed to land.

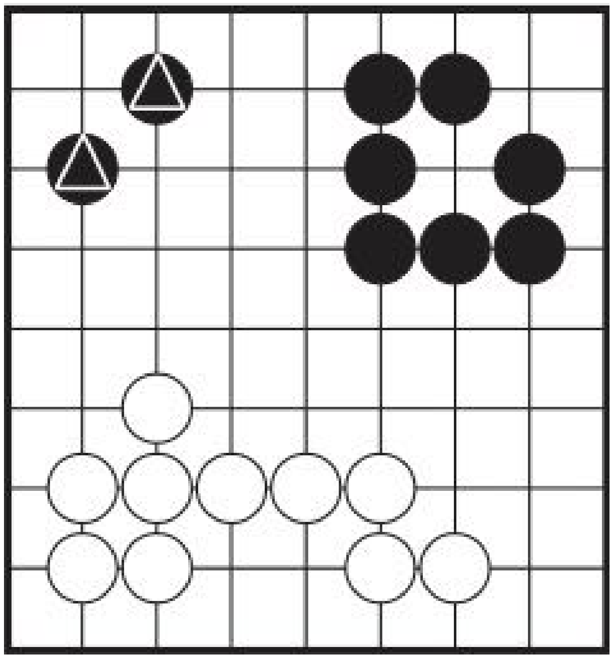
In contrast with the rule of chess, black gets to go first in Go. The overarching purpose of Go is to enclose territories and capture stones, which are essentially the two important concepts that will whoven throughout the later excerpts of this paper. Let's start with stone capturing:
To begin, we define an important concept called *liberty*, which for a single stone, represents the number of adjacent crosspoints that are unoccupied by the opponent's stone. For example:



In the diagram above, the black stone in part A has 4 liberties since all the adjacent cross-points are empty. Same idea applies for B, C, D, except $1, 2$ and 3 adjacent cross-points are occupied by a white stone. Now, a natural question would arise: ***What happens when all liberties are being occupied?***

This directly leads to an **important lemma in Go**: A stone or a string of stone is *dead*, or in a more formal way, being captured, when it's liberty is 0. It's relatively simple to count the liberties when there is one single stone, but difficulty starts to increase when multiple stones with the same color are adjacent to each other, forming a *group*.

As you connect more stones adjacent to each other, then it becomes intuitive that the adjacent stone can cover up 1 liberty of the other stone, resulting in over-count. As shown below, for the string of white stones, there are 11 stones in total, none of which are touching black's stone. However, the white string only has a total of 15 liberties, while there are a total of 11 stones. Clearly, there are 29 liberties that are being "overcounted".

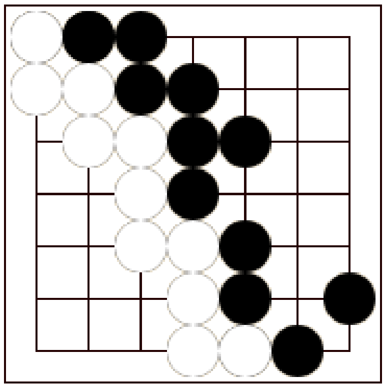


During an actual Go game, both players can easily compute the number of liberties, say, this string of white stones by manually counting. However, computer would hardly implement such human-strategy. The problem becomes how to effectively traverse through the board to determine the state of every stone on the board?(State is defined whether it's captured or alive, depending on the amount of liberty of the Go string that it connects to).

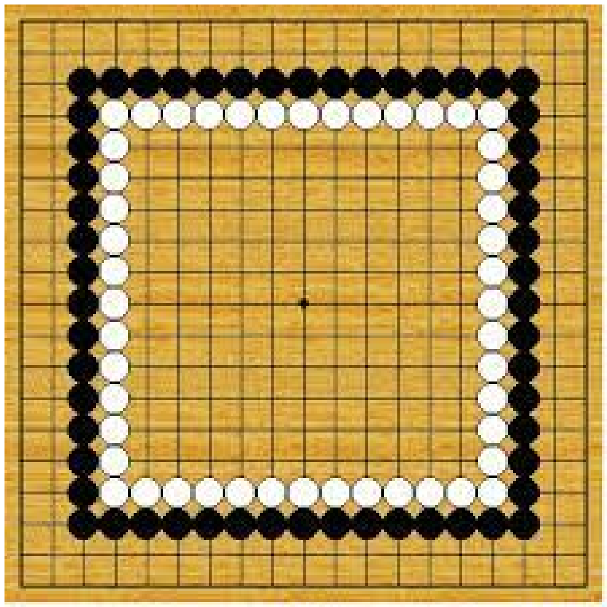
After knowing these basic rules, a natural question arises–How to determine which side is winning at the end of the game? A simpler version of an ended game is shown in the graphic below. The winner is the side with the most territory. The word "territory" seems to be a bit ambiguous in this scenario. This introduces to our second lemma:

Lemma 2: A cross point on the board is considered to be a *point* of a side if it is completely enclosed by stones of exactly one color, and the boundaries of the board.

For instance, in the graphics below, white has a total of 11 points as territory because all the 11 points on the bottom left corner are within the boundary of white stones and the west, south boundaries of the game board. Analogous with black.



Remark: The same problem arose again: During a game, a player can clearly count the number of territories as shown in the graphic, but how would computer execute? In the case at the bottom, the shapes of both black's territory and white's territory is pretty neat–Both composed of squares and rectangles, which are easy to calculate their area. However, when we consider the graphic 1.3, which the shapes aren't that nice to deal with.



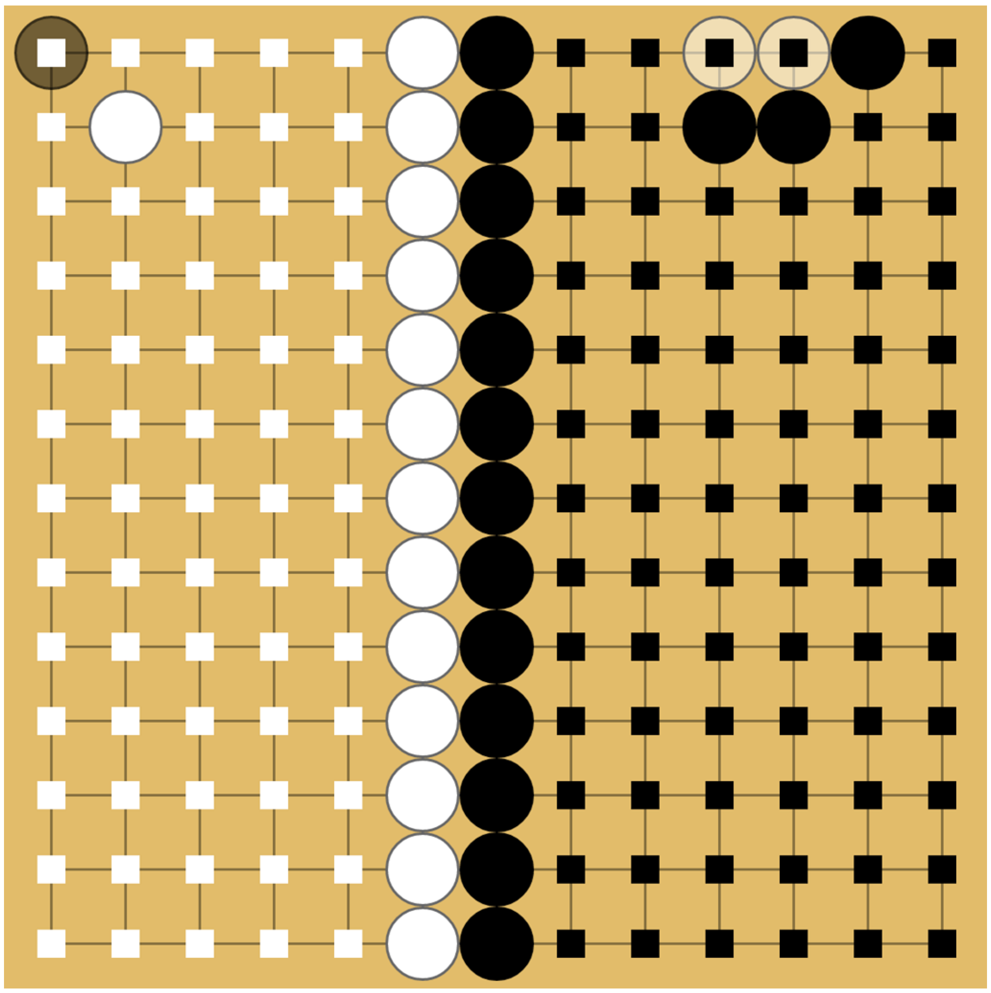
At this moment, I encourage you as readers to actually think about how computer can possibly execute both of the problems I listed above, which involves stone capturing and territory counting, respectively. ***Think outside of the box.***

As a hint, both of the problems are solved by the same algorithm/idea. Draw out some similarities between these two common problems, and research into data structures and algorithms. The detailed solution is presented from page $6-14$, which includes both the code implementations and the algorithm ideas/intuitions.

4 Go and Graph Theory

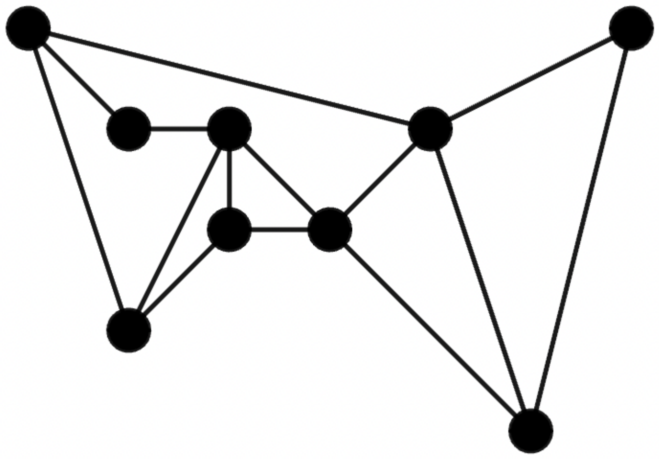
How are Go and graph connected?

A really large part of this Go project, or any large Go playing website, such as TYGEM, FOX and CGOBAN, is the mechanism of capturing and territory counting.

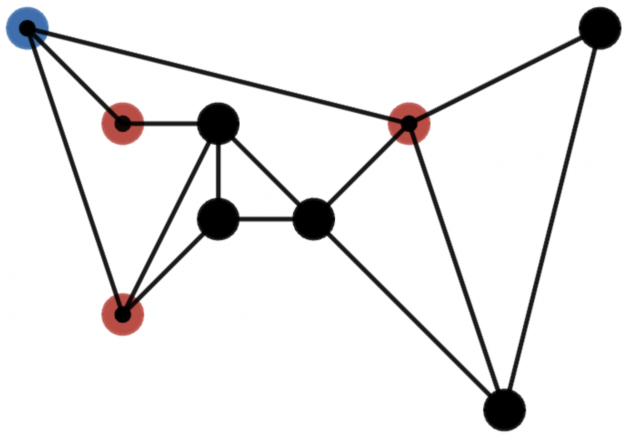


First, we will start with territory counting. A territory is all tiny black squares, and all territories, at endgame, must follow that it's **COMPLETELY** enclosed by the border of the game board and the stones of the **SAME** color. A good example would be that in the diagram above, all crosses on the left of the white stones are all white's territory, because its enclosed by the white stones in the middle and the three sides of the board. Same for the black territory at right.

So, all we want to do is to recursively search for territories until we reached the border of the game board OR stone of a color. This can be implemented by Graph traversal algorithms, specifically, Depth first search.



In any graph, Depth first search can be implemented to find ALL connected components within a graph. The aspect that we are going to use in the Go project is ***Floodfill***. Consider a pointer on the top left point. How floodfill works is that its going to search all nodes in a graph that is adjacent to the current node, and the important part is that **IT's NOT GOING TO RESEARCH THE SAME NODE TWICE**. To animate that, the following figure demonstrates how floodfill works. In the bottom diagrams, all visited nodes will be marked green, and all the unvisited nodes will be marked black, and all nodes that are adjacent to the current node that is not visited yet will be marked red. The current node is marked green.



The computer will continue this process, and all nodes will be marked green in this case eventually when you start traversing from the blue point, since it is reachable to all other points from the blue point.

5 DFS Without Dead Stones

```
public void searcher(int x, int y, ArrayList<Cell> arr){
        if(!this.cellArr[x][y].getState() && !this.cellArr[x][y].getMark()){
            this.colors.add(cellArr[x][y].getcolor());
            return;
        }
        if(this.visited[x][y]){
            return;
        }
        if(this.cellArr[x][y].getMark()){
            this.currterri++;
        }
        this.visited[x][y] = true;

        arr.add(this.cellArr[x][y]);

        this.currterri++;

        if(x + 1 <= 18){
            this.searcher(x+1, y, arr);
        }
        if(x - 1 >= 0){
            this.searcher(x-1, y, arr);
        }
        if(y + 1 <= 18){
            this.searcher(x, y+1, arr);
        }
        if(y - 1 >= 0){
            this.searcher(x, y-1, arr);
        }



    }
```

From my implementation, I essentially created a two dimensional array, ***visited***, that stores a boolean value "ischecked" for all the 19×19 crosspoints. From this code, once the searcher reaches a point, let's say, the point which the x-index is x_1 and the y-index is y_1, then visited[x][y] is marked to be true. The part that checkes "getmark" would be explained here(dead stones), however, the general case when there isn't a dead stones is basically demonstrated.

To simulate a Go board, we will start by creating a 19×19 matrix filled with zeroes. Zero represents the boolean "false" and one represents the boolean "true".

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & . & . & . & 0 \\ . & . & & & & & & & \\ . & & . & & & & & & \\ . & & & . & & & & & \\ 0 & & & & & & & & \end{bmatrix}$$

For the sake of simplicity, consider a small chunk of the matrix. Furthermore, let 1 denote a black stone, and let 2 denote a white stone. A small chunk would be

$$\begin{bmatrix} 1 & 1 & 1 & 2 & 2 \\ 1 & \textcolor{red}{0} & \textcolor{red}{0} & 1 & 1 \\ 1 & 1 & \textcolor{red}{0} & \textcolor{red}{0} & 1 \\ 2 & 2 & 1 & \textcolor{red}{0} & 1 \\ 0 & 2 & 1 & 1 & 1 \end{bmatrix}$$

Clearly, all the red zeroes marked are completely enclosed by 1, which represents black. Thus, black would have 5 units of territory. Now, we present how to do a depth first search. Traverse through all the red zeroes, and then, when its ended and touches a number that is not zero, which is either 1 or 2, in order to ensure that a chunk of crosses is enclosed by a single color, we will use Hashset.

Hashset does not contain any duplicate of values. For example, $\{1, 2, 3, 5, 8\}$ is allowed, but $\{1, 3, 3, 5, 8\}$ will not be a valid hashset. The idea behind the algorithm is that, once we reached a number that is non-zero, store the number into the set and end the search tree. IN the example above, the hashset at the end of the graph traversal will be $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$, which extracts to only $\{1\}$. Now, what if there is another color in the area? Then, it would essentially be

$$\begin{bmatrix} 1 & 1 & 1 & 2 & 2 \\ 1 & \textcolor{red}{0} & \textcolor{red}{0} & 1 & 1 \\ 1 & 1 & \textcolor{red}{0} & \textcolor{red}{0} & 1 \\ 2 & 2 & 1 & \textcolor{red}{0} & 1 \\ 0 & 2 & 1 & \textcolor{blue}{2} & 1 \end{bmatrix}$$

Which in this case, the hash set results in $\{1, 2\}$, which means that the red zeroes cannot be counted as black's territory, unless the white stone marked blue is a dead stone, which would be counted as black territory.

Using this terminology, we will be able to determine the territories without any dead stones present in one's territory. This can be done by recursively calling the searching function. Consider checking the point cellArr[a][b]

$$\begin{cases} \text{If cellArr[a][b] is visited or there is a stone on cellArr[a][b], then end the searcher.} \\ \text{Else, continuously searching all the surrounding crosses by calling the function itself.} \end{cases}$$

And this ideology is essentially Floodfill, a branch of Depth First Search that is used to find the connected Components.

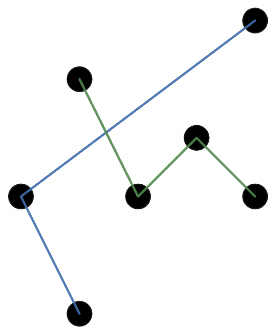
6 With Dead Stones

However, things doesn't always work in our favor... Consider the large Go board in section 2. There are two white stones in black territory, and there is one black stone in white territory. They are *Dead Stones.*

Reconsidering our original searching algorithm, the searcher always terminate when it encounters a point that is already visited or if there is already a stone at that location. However, dead stones are considered as a part of the territory. Therefore, we found a contradiction in the old algorithm, and we need to optimize it in a way that will be able to consider those dead stones.

With this mindset, we created the mechanism that the users will click on the dead stones, and when they click a stone, all the stones adjacent to it will be clicked as a dead group.

In order for this to happen, we have to discard the previous algorithm of a visited two dimensional array. Instead, we will use component DFS.



In this example, the dots connected with a specific color is connected with each other. Notice that this graph is not a ***Strongly Connected*** graph, meaning that it is impossible to reach all other points from any point.

For this program, the user interface is the following: User clicks a stone, and the searcher search through all the stones that are within the connected component of the stone clicked, and marked those stones with the color opposite to the color of the stone. This is because the stone marked are *Dead Stones*, which belongs to opponent's territory.

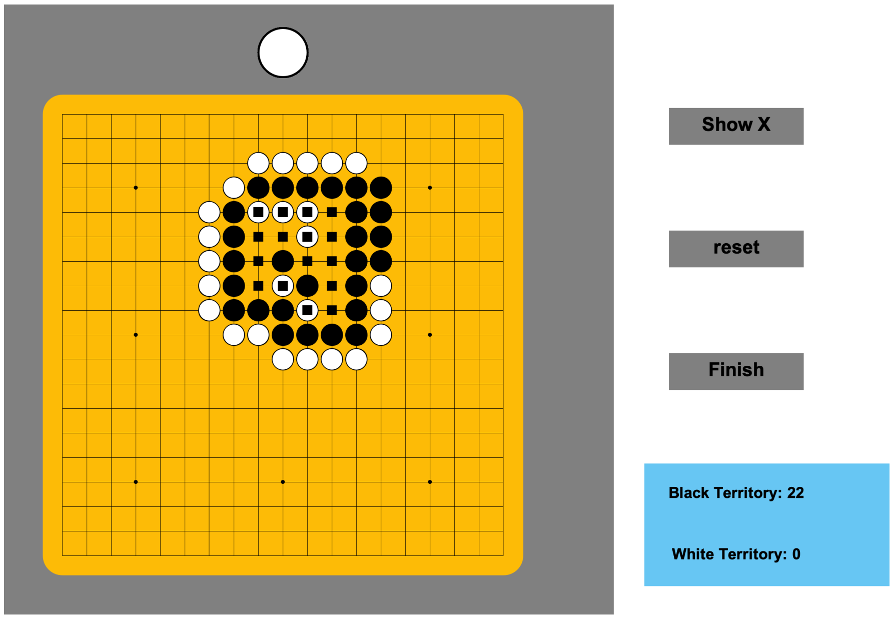
The general idea is first, create a two dimensional 19×19 arrayl that stores every element to be 0. In a more mathematical way, create a 19×19 matrix filled with 0. Now, we use a double for loop that searches through all crosspoints on the Go board. My program was created such that each 361 crosspoints contains a state: Black, White and blank, which blank means nothing is on there. After we setted the three states, we increment the component by 1, and we go through the entire 19×19 array to check all the connected strings, which is also done by Depth First Search. Then, we check for the location that user clicked on the board, and if it is within a certain distance from a stone, then we record the component number of that stone. Now, we continue to do a linear search among the two dimensional array to find all stones that are connected to that position.

$\star$ *A way to optimize this from an $O(N^2)$ algorithm is to use floodfill again by taking advantage of the fact that since we already categorized each connected Go String by a number, then, we can just continuously using floodfill to search the adjacent stones of the current stone. This would optimize into an $O(N)$ algorithm.*

From here, another question rises: How do we incorporate dead stones into territories? Recall the endgame rules of Go,

$$\text{Total Territory} = \textit{Number of crosses excluding the dead stones} + 2(\textit{Number of dead stones}$$

Within a closed area.



Black Territory: 22

White Territory: 0

In this scenario, all the white stones with a black mark on it are marked as dead stones, and all the black mark on an empty cross are the points that only counts for 1 point. There are 10 such points, and there are 6 dead white stones. This gives $10 + 2 \cdot 6 = 22$ points for this enclosed area. However, the territory can also be counted in the following way:

$$\textit{Total \# of crosses enclosed(excluding stones of the same color)} + \textit{Number of Dead Stones}$$

Which if we use this calculation instead, we would get $16 + 6 = 22$, the same result! This is because $\boxed{\textit{Number of empty crosses} + \textit{Number of Dead Stones} = \textit{Total number of crosses}}$

6.1 Implementation

In order to implement Depth First Search for that, we essentially have to carry out the following steps:

1. As we do DFS for the connected components when the user clicks for the dead stones, we create a counter that increments by 1 as we reached a dead stone that is within the same connected component. This can be done through checking the ”state” of a cross point, which basically indicates whether a stone is placed on the cross point, and stores the color as a string variable.

   (a) Initialize a 19×19 array with each marked 0, which $array[i][j]$ denotes the component it belongs to.

   (b) Start by making a counter with initial value 1.

   (c) Loop through the entire two dimensional array, and if $array[i][j] = 0$, meaning it is unvisited, the pointer will recursively search the adjacent crosses.

   (d) This is useful given the fact that each cell must be within a group, and exactly one group, so it is a bijection. This is trivial to proof through contradiction.

2. Count the dead stones as an empty point when doing DFS for the territory counting.

   (a) Check through conditional statements whether the stone with the opposite color is marked.

   (b) If it is, increment the territory counter and continue searching the surrounding cells, analogous to an empty cell.

Specific code:

```
public void countTerritory(){
        ArrayList<Cell> terri = new ArrayList<Cell>();
        for(int i = 0; i < 19; i++){
            for(int j = 0; j < 19; j++){
                if(!visited[i][j]){
                    //reset the variable to be 0.
                    this.currterri = 0;
                    this.colors.clear();
                    this.searcher(i, j, terri);
                    if(this.colors.size() == 1){
                        if(this.colors.contains("black")){
                            this.blackterri += this.currterri;
                            this.show_territory("black", terri);

                        }
                        else if(this.colors.contains("white")){
                            this.whiteterri += this.currterri;
                            this.show_territory("white", terri);
                        }
                    }
                }
                terri.clear();
            }
        }
}
public void searcher(int x, int y, ArrayList<Cell> arr){
        if(!this.cellArr[x][y].getState() && !this.cellArr[x][y].getMark()){
            this.colors.add(cellArr[x][y].getcolor());
            return;
        }
        if(this.visited[x][y]){
            return;
        }
        if(this.cellArr[x][y].getMark()){
            this.currterri++;
        }
        this.visited[x][y] = true;

        arr.add(this.cellArr[x][y]);

        this.currterri++;

        if(x + 1 <= 18){
            this.searcher(x+1, y, arr);
        }
        if(x - 1 >= 0){
            this.searcher(x-1, y, arr);
        }
        if(y + 1 <= 18){
            this.searcher(x, y+1, arr);
        }
        if(y - 1 >= 0){
            this.searcher(x, y-1, arr);
        }
    }
```

7 Further Research Topics(Unresolved)

7.1 Dead Stone Detection(Graph theory)

1. Auto-eye detection for stones, which would be used during board estimation throughout the game that gives a rough estimate of the territory of both side. The clear hardship is that it's nearly impossible to solve this problem using an algorithmic approach. Consider the following scenarios:

   (diagram insertion)

   During an in-person game, players estimates their scores by creating a rough boundaries, and they detect the dead stones in a more holistic way based on whether the group has two eyes. However, computer would not be possible to carry out such procedure. Furthermore, dead stones are often in different shapes, and their eyes are structured differently. Therefore, it is nearly impossible to create a systematic approach to assess each group of stones as a function with an input and an output–an input being the group of stones and an output representing its living status, which directly leads to whether it should be counted as a part of territory for the opponent.

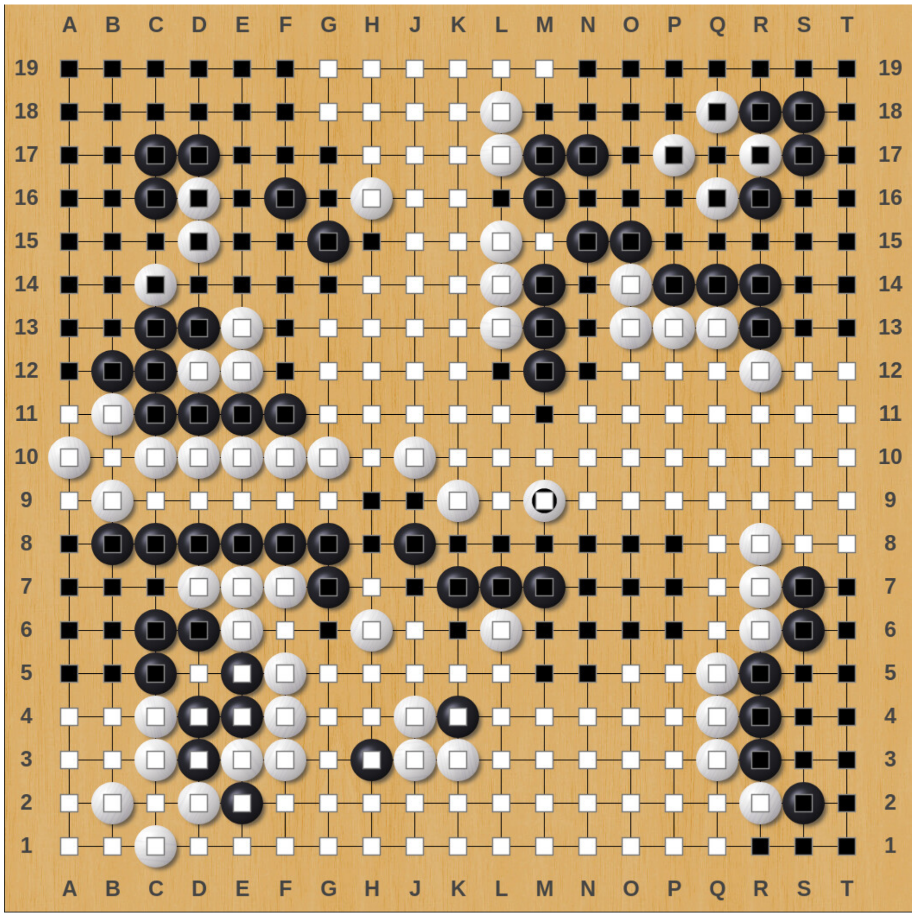
   One of the approach might be machine learning by creating a large database of Go games for the computer to recognize the pattern in eye recognition, and moreover, give the best estimate of whether a group is dead by using a weighted-machine learning formula. In a typical machine learning scenario, the computer would give an estimate with multiple factors into play. For example:

   Let $p_1, p_2, ..., p_n$ be the factors of a decision Q, then

   $$Q = p_1x_1 + p_2x_2 + ... + p_nx_n = Q$$

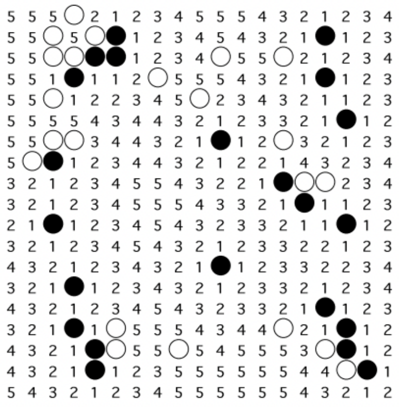
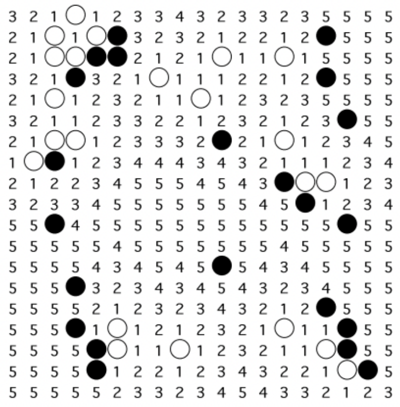
   which $x_1, ..., x_n$ are the "weights" of each factor. In other words, this estimation is basically a weighted average, and using a large database would reinforce such accuracy.

8 Score Estimation



Clearly in this case, difficulty drastically arises because boundaries aren't being considered. In this case, it's impossible to implement the DFS approach described in the earlier part of this paper. For territory counting, we relies on using a Hashset to detect all the possible stones that a *search branch* from a starting stone can reach to. One of the possibility to resolve that is to use a density evaluation, which there will be limitation about the accuracy since it assigns a value to all the unoccupied crosses on the board. This approach is based on the concept of uncertainty for each stones. To evaluate the side of each point, we find the color of stone that the crosspoint is closest to based on Taxicab distance. The definition of the taxicab distance between two points (x_1, y_1) and (x_2, y_2) is, instead of $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, is $|x_2 - x_1| + |y_2 - y_1|$. Then, it suffices to mark the distance from each stones to the closest stone, but one of the limitation is that the process of searching cannot touch the opponent's stone.

From *Counting the Score: Position Evaluation in Computer Go*[1]

This illustrates the distance calculation for black and white, respectively. The maximum is setted to be a stationary value, which in this case, a 5. Then, each point is marked by a number that represents the distance to the closest stone of a color without touching the stone of the opponent's color. WLOG let's assume that we are considering the distance to the black stones, then if the shortest route to reach the closest blackstone without touching any of the white stones is larger than the presetted maximum, then the starting point will be labeled with the maximum value. We denote the maximum value to be m.

If we store the ordered pair of (black distance, white distance), denote as $Q(m)$, to each of the cross point, then it's easy to verify that the two elements of the ordered pair cannot both be 5 unless if it will not touch any stones within the:

$$\begin{cases} (2m)^2 \text{ boundary for all points } \{(x,y) \mid m \le x \le 18-m, m \le y \le 18-m\} \\ (2(m-k))^2 \text{ boundary for all points } \{(x,y) \mid m-k \le x,y \le m \text{ or } m+k \le x,y \le 18-m-k\} \end{cases}$$

Which we will have to detect all such possible squares for each point, and time complexity wise, all these squares are considered as m^2, which the time complexity will be $O(n^2m^2) = O(361m^2)$, which is basically square time complexity and acceptable. Now, we continue with analysis of the numbers. WLOG we just use 5 as the maximum threshold. Analyzing the distance distribution of the black stones(figure one), then we can see evidently that att he lower bottom left corner, all the points seems to have a value of either $1, 2, 3$ or 4. Analogously, the top right corner occupied by black is also shown. However, at the top left corner which is occupied by the white, all the points are marked 5. Looking at the rules above, we are able to make the following conjecture:

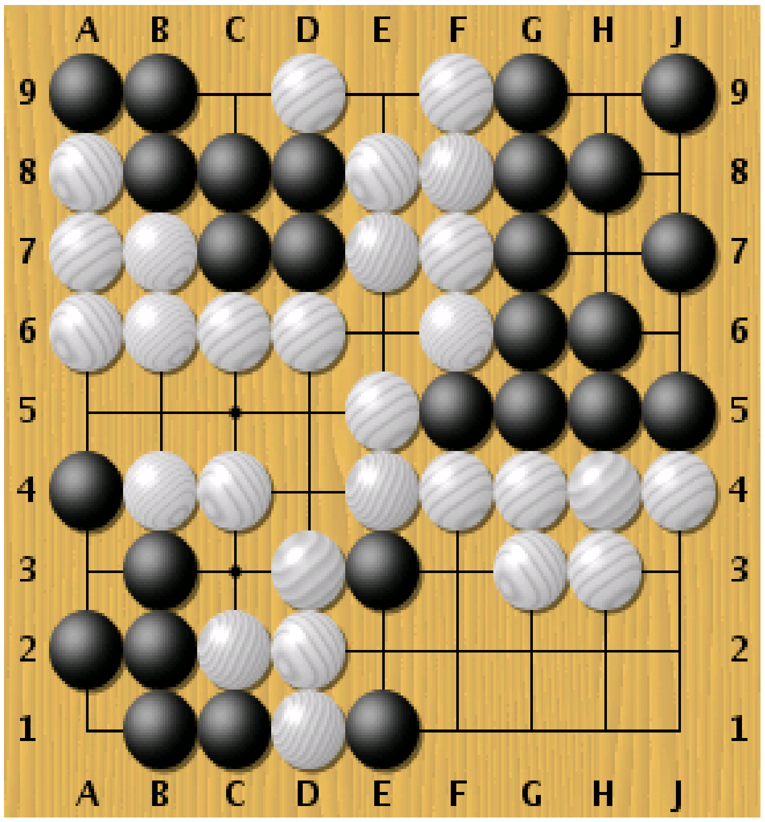
Conjecture: All points marked $1, 2, 3$ or 4 in one of the diagram will be calculated as the territory of that color, and any points marked as 5 does not belong to any side, which is regarded as the "swing" space. If we generalize to all the possible m, consider a point (x, y), then if $Q((x, y)) = (a_1, a_2)$, then if $a_1, a_2 = m$ or if $a_1, a_2 \neq m$ then the point is a "swing" space.

Clearly, there are some exceptions. For example, in the diagram above, the point in the lower left corner has value 5, while it is consider black's territory, which explains why in major Go softwares often points at the corner get neglected in score estimation.

Now, an optimized solution in contrary to setting up a specific maximum value would be to find the shortest distance and assign it to each point, but without having the threshold of m. After that, we can analyze the ordered pair. After we assign each point to an ordered pair (a_1, a_2), which is defined in the same way as the previous method, then we can add an uncertainty value α such that $\frac{\min(a_1,a_2)}{\max(a_1,a_2)} = \alpha$. For instance, if $a_1 = 1, a_2 = 16$, meaning it's only one away from the black stone, and the closest distance to a white stone is 16, then the uncertainty value is $\frac{1}{16}$, which is pretty minimal. This algorithm can be confirmed as at the end game, the uncertainty value will always be 0.

Time complexity wise, this new method would result in $O(n^4)$, which is acceptable for $n = 19$ which would give $O(130121)$. Now, the question arises: What will be the threshold of value α to assign the cross-point to be a territory point of one side, or if we assign $k(1-\alpha)$ territory points to the side that has a smaller value of distance, which will be a continuous estimation?

Clearly, the latter is not the way of evaluation implemented by the major Go companies, such as Tygem, Fox and OGS, since all the outcomes are written as $a\frac{b}{4}$, while the denominator of $k(1-\alpha)$ can be any positive integer ranging from 1 to 361. While this becomes more efficient,



In this example, take a look at the bottom row in white's territory, that is, all the points marked

$(1, E), (1, F), (1, G), (1, H), (1, J)$. Then, all the uncertainty values are $0, \frac{1}{3}, 1, \frac{2}{3}, \frac{3}{4}$ despite this being an endgame, which all the black stones below will be marked as dead. However, by using the first method–making a threshold for α–, it can be extremely tricky to create a maximum uncertainity level α_1 while still maintaining the accuracy of estimation. Therefore, we will mostly research into the second method, which define $t(p, k)$ to be a function of that takes a point p with a constant k as inputs, and output the estimated certain territory for that point, which ranges between 0 and 1, and define $d_1(p)$ and $d_2(p)$ to be the minimum distance to reach a black stone and a white stone, respectively. Furthermore, as predefined, we let α_p be the uncertainity value of point p, and we define confidence level to be $1 - \alpha_p$, that is, one minus the uncertainty level. Moreover, we also want to detect whether $t(p, k)$ is

$$t(p, k) = k\left(1 - \frac{\min(d_1(p), d_2(p))}{\max(d_1(p), d_2(p))}\right) = k(1 - \alpha_p)$$

Now, we denote B to be the set of points p such that $d_1(p) > d_2(p)$, and W otherwise. Notice that we don't consider $d_1(p) = d_2(p)$ as $f(p, k) = 0$ in that case.

Furthermore, we define t_b to be the expected territory estimation from human professionals, then we consider the Mean Square Error to be

$$L(k) = \frac{1}{2}\left((t_b - \sum_{p\in B} k(1 - \alpha_p))^2 + (t_w - \sum_{q\in W} k(1 - \alpha_q))^2\right)$$

Now, we take the gradient to give

$$\frac{\partial L}{\partial k} = \frac{1}{2}\left[-2\left(t_b - \sum_{p\in B} k(1 - \alpha_p)\right)\left(\sum_{p\in B} 1 - \alpha_p\right) - 2\left(t_w - \sum_{q\in W} k(1 - \alpha_q)\right)\left(\sum_{q\in W} 1 - \alpha_q\right)\right]$$

$$= -\left(t_b - k\sum_{p\in B}(1 - \alpha_p)\right)\left(\sum_{p\in B} 1 - \alpha_p\right) - \left(t_w - k\sum_{q\in W}(1 - \alpha_q)\right)\left(\sum_{q\in W} 1 - \alpha_q\right)$$

Which when analyzing these different components, notice that $1 - \alpha_p$ is always positive, and the sum of $k(1 - \alpha_p)$ will always be less than t_b for some small value k. This works the same with white's territory. Now, we define x to be the total confidence level across black territory, and y be the total confidence level across white territory, then we can rewrite further as

$$\frac{\partial L}{\partial k} = -(t_b - kx)x - (t_w - ky)(y) = k(x^2 + y^2) - xt_b - yt_w$$

Now notice that $\frac{\partial^2 L}{\partial k^2} = x^2 + y^2$ which is always positive, and by the definition of machine learning, we want to minimize the value of $L(k)$, so we basically just want to get $\frac{\partial L}{\partial k} = k(x^2 + y^2) - xt_b - yt_w = 0$, which gives the minimum of $L(k)$ at $k = \frac{xt_b + yt_w}{x^2 + y^2}$. From here, we have to following conjecture:

Conjecture: Assume that at a certain point when black's territory is evaluated to be t_b and white's territory to be t_w, and define x to be the total confidence level of all points in B, and define y to be the total confidence level of all points in W, then the k that will provide the best estimation is

$$\boxed{\frac{xt_b + yt_w}{x^2 + y^2}}$$

Furthermore, this will create a MSE function of

$$\frac{1}{2}\left[\left(t_b - \frac{x^2 t_b + xyt_w}{x^2 + y^2}\right)^2 + \left(t_w - \frac{xyt_b + y^2 t_w}{x^2 + y^2}\right)^2\right]$$

That is granted as the minimum.

An easy check to this is that when the game is ended, then clearly $t_b = x$ and $t_w = y$ since the size of B and W are t_b and t_w, respectively, and at that endgame, every space in the board is either completely enclosed by black or by white, so either one of $d_1(p)$ or $d_2(p)$ will be 0 since there will be no way to reach the stone of that color without touching the stone of the opposite color. Therefore, k will just be $\frac{x^2+y^2}{x^2+y^2} = 1$, which is as desired.

This, however, is a scenario with incomplete information because of dead stones. In actual games, human players estimate the dead stones based on the life/death status of the stones(*check the criterias of the dead stones in the previous chapters*). Because of the vast difference between human's mind and computers, it's impossible for computers to adapt to that intuition. This is a potential topic of my research in the future. For now, assume that no dead stones are present.

8.1 Implementation:

The first step is to compute the 19×19 matrix that contains all the uncertainty level(Maybe to simplify our calculation, we will compute the confidence level instead). One of the most natural way is to use BFS(Breadth First Search) which effectively detects the existence of "obstacles" that represents the stones of the opponent while maintaining the distance. During the traversal, there are a total of three cases: If the traversal branch encounters a stone of the opposite color, then immediately end that branch(return void); else if the branch encounters a stone of the same color, then we update the minimum distance accordingly; finally if we reached the border of the board, then we return void. Doing this twice to find all the "black" distances and "white distances". Then, we will categorize them into B and W, which is just a matter of conditional statements–we create a new 19×19 array, and assign each coordinate with 0 or 1 from the conditional statement that compares $d_1(p)$ and $d_2(p)$. From here, we are enough to achieve our goal by answering the following questions:

1. What are the accuracies of the theoretical value of k from conjecture 2 for different boards?
2. How does this accuracy correlates with the number of moves played?

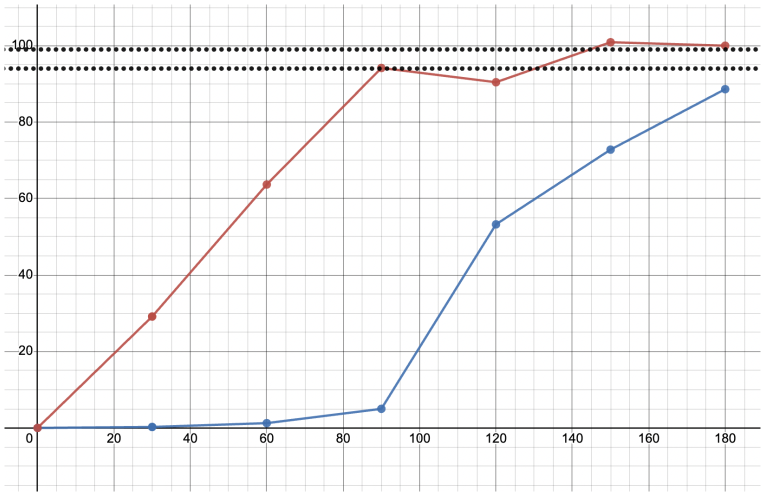
```
public void theoretical(int tb, int tw, double k){

        double[][] conf = distMatrix();

        double estiBlack = 0;
        double estiWhite = 0;

        for(int i = 0; i < 19; i++){
            for(int j = 0; j < 19; j++){
                if(category[i][j] == 1){
                    estiBlack += k * conf[i][j];
                }
                else if(category[i][j] == 2){
                    estiWhite += k * conf[i][j];
                }
            }
        }

        System.out.println(estiBlack);
        System.out.println(estiWhite);


    }

    public double[][] distMatrix(){
        double[][] confidence = new double[19][19];

        for(int i = 0; i < 19; i++){
            for(int j = 0; j < 19; j++){
                if(cellArr[i][j].getcolor().equals("null")){

                    distSearch("black", i, j);
                    distSearch("white", i, j);

                    confidence[i][j] = (1.0 -  (double) Math.min(blackdist[i][j], whitedist[i][j]) /
                    double w = Math.round(confidence[i][j] * 1000);
                    confidence[i][j] = w / 1000;
                }

            }
        }
        return confidence;
    }

    public void distSearch(String color, int startX, int startY){

        this.v = new boolean[19][19];

        Queue<pair> pq = new LinkedList<>();

        pq.add(new pair(0, cellArr[startX][startY]));

        while(!pq.isEmpty()){
            pair cur = pq.poll();

            v[cur.getCell().getX()][cur.getCell().getY()] = true;



            if(cur.getCell().getcolor().equals(color)){
                if(color.equals("black")){
                    blackdist[startX][startY] = cur.getSteps();
                    category[startX][startY] = 1;
                }
                else{
                    whitedist[startX][startY] = cur.getSteps();
                    category[startX][startY] = 2;

                }
                return;
            }
            else if(!(cur.getCell().getcolor().equals("null") || cur.getCell().getcolor().equals(colo
                continue;
            }

            if(cur.getCell().getX() + 1 <= 18 && checkadj(cur.getCell().getX()+1, cur.getCell().getY(
               pq.add(new pair(cur.getSteps()+1, cellArr[cur.getCell().getX()+1][cur.getCell().getY()
            }
            if(cur.getCell().getX() - 1 >= 0  && checkadj(cur.getCell().getX()-1, cur.getCell().getY(
               pq.add(new pair(cur.getSteps()+1, cellArr[cur.getCell().getX()-1][cur.getCell().getY()
            }
            if(cur.getCell().getY() + 1 <= 18 && checkadj(cur.getCell().getX(), cur.getCell().getY()+
               pq.add(new pair(cur.getSteps()+1, cellArr[cur.getCell().getX()][cur.getCell().getY()+1
            }
            if(cur.getCell().getY() - 1 >= 0  && checkadj(cur.getCell().getX(), cur.getCell().getY()-
               pq.add(new pair(cur.getSteps()+1, cellArr[cur.getCell().getX()][cur.getCell().getY()-1
            }







        }

        return;




    }
```

Continue to optimize the theoretical value of k, let m be the number of moves played so far, then this arrives to the next conjecture:

Conjecture 2: The value of k should also have a direct proportion with the number of moves that are played so far in the game. Let m be the number of moves played, and T_m be the total number of moves in a game.

$$k = \frac{mxt_b + myt_w}{T_m(x^2 + y^2)}$$

9 Data Analysis and final discussions



1. The dashed-lines are the targeted territory values, which are 94 and 99. I tested with a consistent interval of 30 up until move 180. The theoretical value of k produces the following set of ordered pairs(red represents white territory, blue represents black territory).

2. This data clearly shows a decrease in uncertainty levels of territory as moves increase. More data are needed to feed in to create a precise mathematical model of how number of move is related to the error bounds of these estimations. The explanation of this direct proportion is that boundaries are more formed during middle game, which the side of control(black/white) at a certain crosspoint becomes more clear than early game. This also creates a justification of why the estimation are off in early game.

3. Further research topics