ArsDigita University
Month 5: Algorithms - Professor Shai Simonson

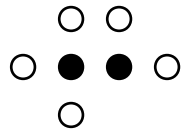
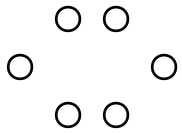
Problem Set 3 – Applications of Graph Algorithms

Legal Moves in the Game Go

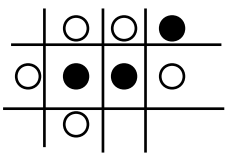
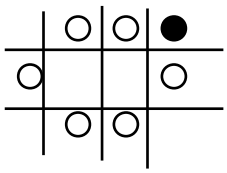
Background: The oriental game Go (http://www2.psy.uq.edu.au/~jay/go/CS-TR-339.html#4.3.1) is played on a 19 by 19 grid with black and white stones. There is currently a prize (the Ing Prize) of $1,500,000 outstanding for the first Go program to beat a bona-fide professional player in a sanctioned match. Go has resisted all attempts at elegant algorithms. It is part of a collection of games including Chess and Checkers which are Pspace-Complete. This is even worse that NP-Complete, and it implies that there is almost no chance the games can be solved efficiently by a computer program. The fact that there are world class Chess and Checkers programs, are a testament to clever engineering and the speed of today's computers. For boards of size 8 by 8, the intractability of the general problem can be tamed. However in Go, the intractability for even a baby board of 9 by 9 is currently insurmountable, hence it has become the darling of the AI community and the game programming hackers. To give you an idea of the current state of the art, each year at the World Computer Go tournament the winning programs plays an exhibition match against a clever apprentice 8 year old in Japan studying to be a Go professional (yes there are Go professionals). The 8 year old gives the program a 9 stone handicap, and then proceeds to beat it badly.

Project: You are going to write a program using depth first search that could be used as a module in a much larger Go program to help determine legal moves and strategy.

Rules of Go: The board starts empty and the players alternate placing their stones on the board. If a stone is placed on the board and it completes the surrounding of a collection of *connected* stones of the opposite color, then that collection of stones is removed from the board. (Diagonal connections do not count as *connections*). For example, figures 1 and 2 show before and after pictures of a *capture*.

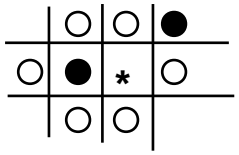
1  **2** 

or

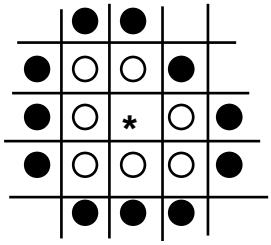
1  **2** 

A person is not allowed to commit suicide. That is, they may not place a stone anywhere where it results in their own collection of stones being surrounded. The exception to this rule is when the stone placed is capturing other stones. For example, in 1, black may not place a stone in the position marked "*". In 2, however, he may place a stone in the position marked "*" and the resulting shape is shown in 3, as all the white stones are captured.

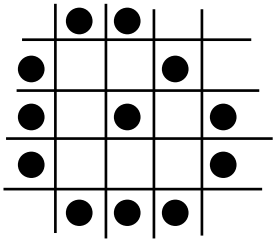
1



2



3



Problems

1. Legal Moves in the Game Go

You will be given an arbitrary *n* by *n* ($n \leq 19$) array filled with 0's (blank), B's (black stones) and W's (white stones), and a character (B or W) indicating whose move it is. Your program should output the array, with all the 0's replaced by either I (illegal) or L (legal) indicating which of the vacant locations are legal moves for the color whose turn it is to move.

Methodology: You should take the two dimensional game array and store it as a graph using an array of linked lists data structure, then use depth first search to help you traverse the graph and identify the legal and illegal moves.

2. (Optional) Further Work on Go

Get your program to work with a GUI, in order to allow two people to play a game of Go. The program should allow only legal moves and it should handle the scoring after three consecutive passes. Scoring is done as follows:

After three consecutive passes, you ask each player to remove his *dead* stones. This is done by allowing each player to click on connected strings of pieces of his color, and

then your program deletes the string of pieces leaving blanks behind. It is important practically to allow *undos* here. Don't be concerned with how you decide if a string of stones is dead, we leave that up to the players. Once all dead stones are removed from the board, the scoring for each color is done by adding the empty spaces surrounded by stones of that color and subtracting the captured and dead stones.

3. Extension of SameGame

Write a method to compute the sequence of moves that achieves the highest possible score from a given configuration of the board in the SameGame you programmed last month.

Methodology: You should generate a tree representing the (many) possible sequences of moves from the given configuration until the end of the game, and return the path from the initial node to the leaf with the highest score. The nodes on the tree represent configurations of the board, and there is an edge from one configuration to another when the first can be transformed into the other in one move. It is useful to also store the current score inside a node, that represents the score achieved so far to reach that configuration. The algorithm can traverse the tree either using DFS or BFS. Note the tree can be constructed on the fly by a simple recursive algorithm. That is, you do not need to first store it and then traverse it. Details will be discussed in recitation.

Note that this problem will almost surely be intractable for the size boards that your programs typically use. Try it on very small boards or it will run too long.

Another way to handle the intractability, is to use an *evaluation* function to measure the goodness of a position. The idea is that if you are doing well and have a chance for a high score then the evaluation function should be high. The function can depend the current score, size of the largest block etc. You should experiment and use your imagination. The point is that you can generate the tree up to some particular depth whether or not the game is over, and instead of returning an actual score, you just return the evaluation function. This way will not solve the problem, but it will give a good approximation to the solution.

4. (Optional - Beyond the Call of Duty) Even More Extensions to SameGame

Let's see how easily extendible Java really is :-).

You can go even further, by allowing new rules. For each variation below, modify your game, and traverse the tree in an appropriate way in order to decide on the best play for each player. Suggestions and hints will be given in recitation.

a. Use the minimum score instead of the maximum.
b. Two players alternate taking turns on one board, with one player's goal to maximize and one to minimize the score. This needs a technique used in Chess programs, invented by Claude Shannon in 1950's called the mini-max algorithm.

c. Two players alternate taking turns on one board, with each player's score kept separately. The goal of both players is to get more than the other player.
d. Two players alternate taking turns on one board, with each player's score kept separately. The goal of both players is to get less than the other player.