David Chavez
Discrete Mathematics

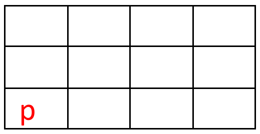
Chomp

Abstract: Chomp is mathematical game played on a rectangular grid of size MxN where M and N > 1. Chomp is a combinatorial game; therefore, it is played between two players where both players have perfect information and the game cannot end in a draw. For some basic cases like NxN boards, 2xN boards, and 3xN boards where N > 1 winning strategies exist, however, the winning strategy is unknown for larger chomp games. The purpose of this project is to make an AI that would play perfectly in a game of chomp. I will limit the board size to a very trivial board (2x2 board) but it should be very easy to scale to much larger games. I also plan to use minimax theorem to help me find the optimal route in the chomp game tree.

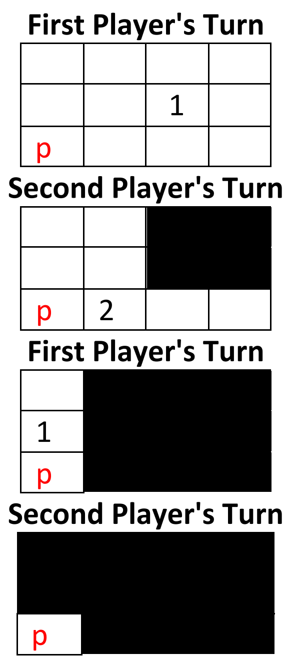
Introduction

To understand the game of chomp it might be easier to see an example of how it is played, therefore, I will begin by showing you a game. The game chomp is played on a rectangular board with n rows and m columns. Where m, n > 1 and the bottom left piece is poison. For example, consider the following board 3X4 board.

Initial Board



Each player takes turns making moves until only the poison piece is remaining. A move consists of selecting any square and removing all squares above it and to the right of it. Selecting the poison square is a loss for the player.



Second player would lose in this case.

For any rectangular board of chomp with N rows and M columns where N, M > 1 first player has a winning strategy.

Proof

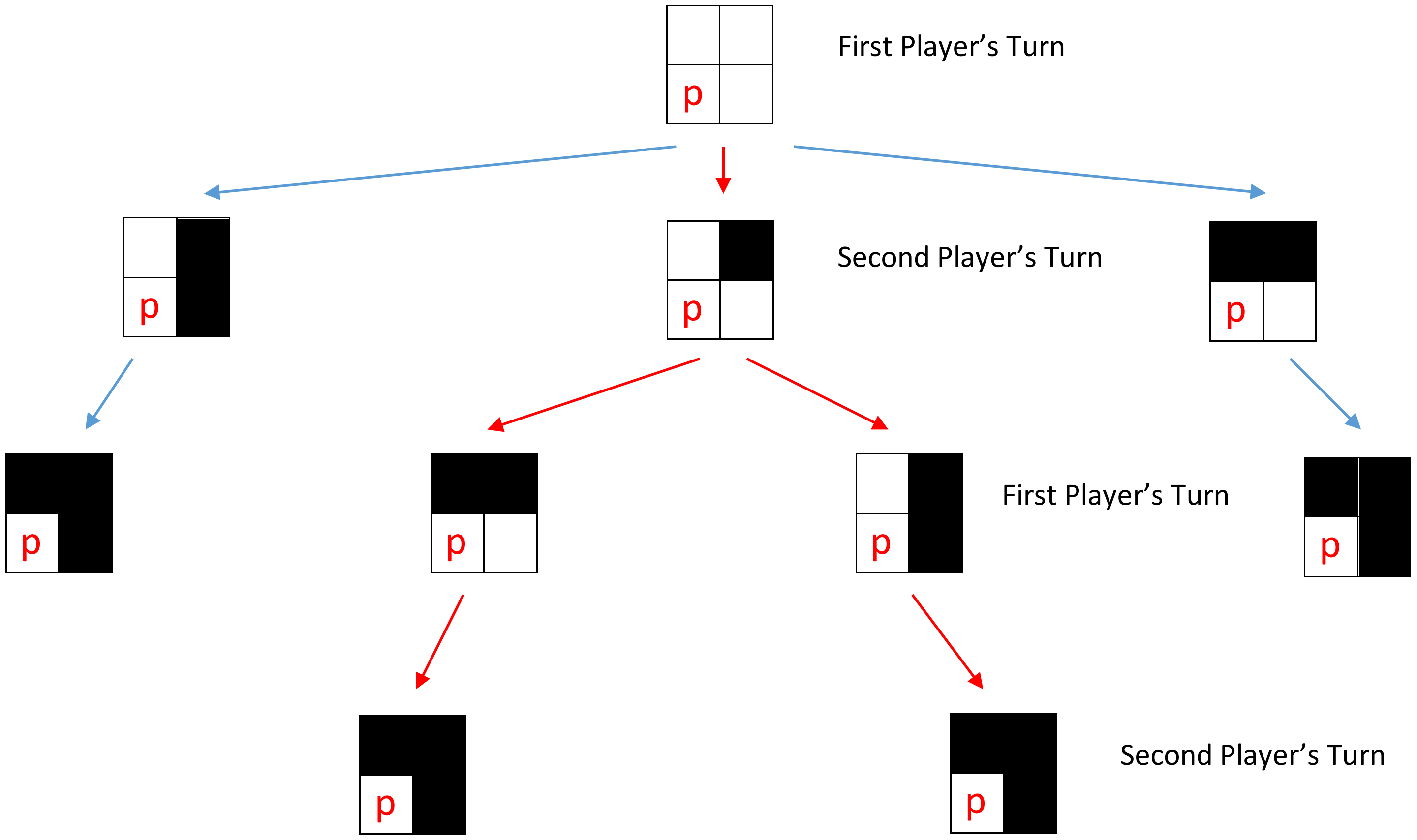
Suppose you have an M by N board where M,N >1. There are two cases to consider.

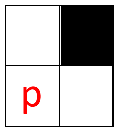
1. Player one chooses the top-right corner piece and it is the winning strategy.
2. Suppose the top-right corner piece is not the winning strategy and second player then chooses the winning strategy. Then first player could have started with the second players winning strategy.

Therefore first player has a winning strategy.

While it is quite easy to show that a winning strategy exist for player one finding the winning strategy can be very difficult for large games of chomp. One way of finding this winning strategy would be to list every possible move in a tree and choosing the route that will guarantee a win for first player. For example, consider the tree in Figure 1 that highlights the winning strategy in red.

Winning Strategy on a 2x2 Chomp board (Figure 1)



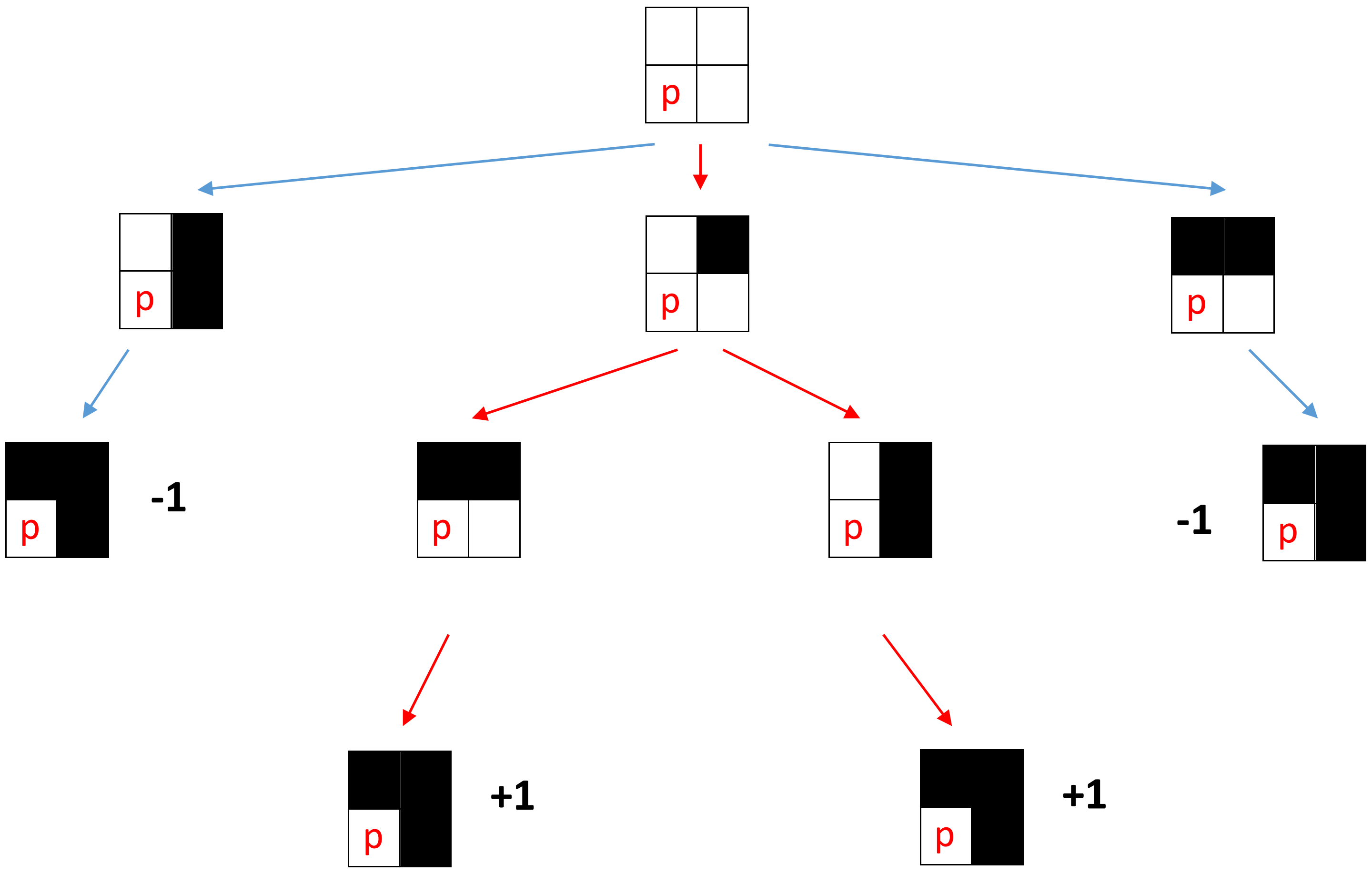
First players optimal move would be to choose .

Methods

Originally I wanted to write code and design a working model for my iPhone, however, I ran into a lot of issues since I am new to coding for IOS and the Swift language. Therefore, I will provide and explain what I tried to code using diagrams. (I provided my failed attempts at the end)

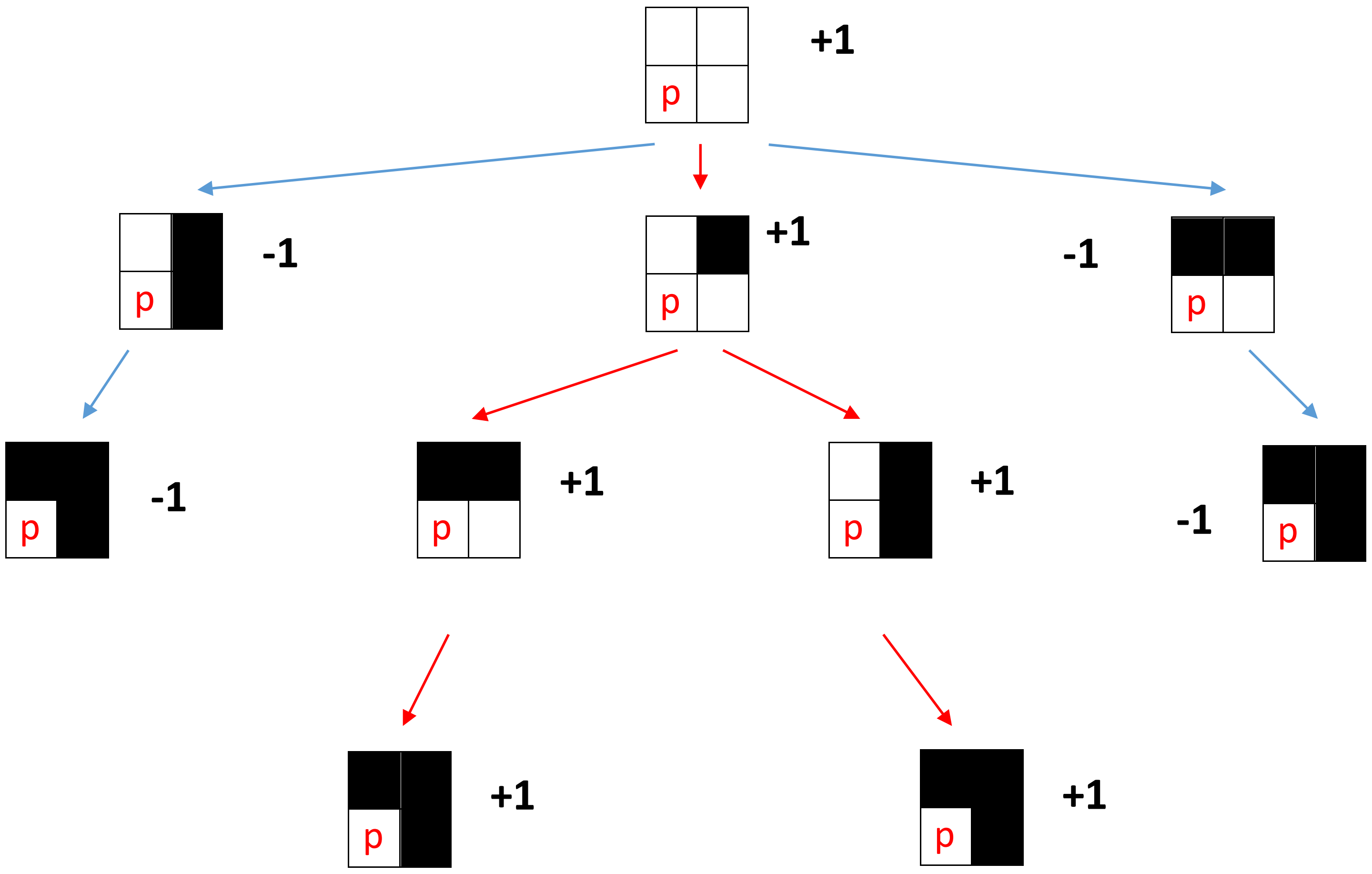
The tree in Figure 1 helps us see that the leaves height determines if it's a win for player one or player two. If the height of the leaf is divisible by 2 then player one loses and if it is not divisible by 2 then player 1 wins. We can now give a point value to each leaf. A +1 if it is a win for player one and a -1 if it is a loss for player one. Figure 2 demonstrates this.

Leaf Values on a 2x2 Chomp board (Figure 2)



Player one wants to maximize their value and player two wants to minimize player 1's value. This is called minimax and when applied to the tree we can get values for each node (Figure 3).

Node Values on a 2x2 Chomp board (Figure 3)



This proves that the path highlighted in red is the optimal path. We are almost ready to code but there is one more thing to note. If you notice you will see that some nodes are repeated in other parts of the tree. This is important when writing code since you do not want to have to do the same calculations repeated. In class we talked about the memoization technique and it will

be very useful for this problem.

Code

The code should do the following.

1. Create a Tree which shows every possible move in chomp using recursion and memoization.
2. Tell what the height is of each leaf.
3. Use minimax on the tree to figure out next players optimal move.

My code is able to create a tree, however, it is not optimal and it does not use memoization. It also is not able to tell the height of the leaf which is needed to apply minimax. Here is my code written in Swift.

```
import Foundation
struct GameModel{
    var matrix:[[Int]]
}
extension GameModel{

    init(model: String) {
        if(model == "Small"){
            let matrix = [[1,   2 ],
                          [3,   4 ],
                                        ]

            self.init(matrix: matrix)
        }else if(model == "Medium"){
            let matrix = [[1,   2,  3 , 4 , 5],
                          [6,   7,  8,  9, 10],
                          [11,  12, 13, 14, 15],
                          [16,  17, 18, 19, 20],
                          [21,  22, 23, 24, 25],
                          [26,  27, 28, 29, 30],
                          [31,  32, 33, 34, 35],
                          [36,  37, 38, 39, 40]]
                      self.init(matrix: matrix)
        }else{
            let matrix = [[1,   2,  3 , 4 , 5 ,6 ,7],
                          [8,  9, 10, 11, 12, 13 , 14],
                          [15 ,16 ,17 ,18 ,19 ,20 ,21] ,
                          [22 ,23 ,24 ,25 ,26 ,27 ,28] ,
                          [29 ,30 ,31 ,32 ,33 ,34 ,35] ,
                          [36 ,37 ,38 ,39 ,40 ,41 ,42] ,
                          [43 ,44 ,45 ,46 ,47 ,48 ,49] ,
                          [50 ,51 ,52 ,53 ,54 ,55 ,56] ,
                          [57 ,58 ,59 ,60 ,61 ,62 ,63] ,
                          [64 ,65 ,66 ,67 ,68 ,69 ,70] ,
                          [71 ,72 ,73 ,74 ,75 ,76 ,77]]
                        self.init(matrix: matrix)
        }
    }
}

class Game{
    var board: [[Int]]
    let ROWS: Int
    let COLS: Int
    init(model: String) {
        self.board = GameModel(model: model).matrix
        self.ROWS = board.count
        self.COLS = board[0].count
    }
    func selection(_ row:Int , col:Int){
        for i in 0...row{
            for j in col...(COLS-1){
                board[i][j] = 0
            }
        }
    }
    func printBoard(){
        for i in 0..<ROWS{
            print(board[i])
        }
        print("")
    }
}



public class TreeNode<T> {
    public var value: T

    public weak var parent: TreeNode?
    public var children = [TreeNode<T>]()

    public init(value: T) {
        self.value = value
    }

    public func addChild(_ node: TreeNode<T>) {
        children.append(node)
        node.parent = self
    }
}
```

```
extension TreeNode: CustomStringConvertible {
    public var description: String {
        var s = "\(value)"
        if !children.isEmpty {
            s += " {" + children.map { $0.description
}.joined(separator: ", ") + "}"
        }
        return s
    }
}

var game = Game(model: "Small")



//Original GameBoard
game.printBoard()
let tree = TreeNode<[[Int]]>(value: game.board)


func treeMaker(tree: TreeNode<[[Int]]>) -> Void{
    let board = tree.value
    for i in 0..<game.ROWS {
        for j in 0..<game.COLS {
            game.board = board
            if(game.board[i][game.COLS-j-1] != 0 ){
                game.selection(i, col: game.COLS-j-1)

                tree.addChild(TreeNode<[[Int]]>(value: game.board))
            }
        }
    }
    if(!tree.children.isEmpty){
        tree.children.popLast()
        for t in tree.children{
           treeMaker(tree: t)
        }
    }
}


treeMaker(tree: tree)

print(tree)
//[1, 2]
//[3, 4]
```

```
//[[1, 2], [3, 4]] {[[1, 0], [3, 4]] {[[0, 0], [3, 4]] {[[0, 0], [3,
//0]]}, [[1, 0], [3, 0]] {[[0, 0], [3, 0]]}}, [[0, 0], [3, 4]] {[[0,
//0], [3, 0]]}, [[1, 0], [3, 0]] {[[0, 0], [3, 0]]}}
```

As you can see I was able to print out all possible moves for a 2x2 game of chomp, however, my code is not optimal and is very slow for larger boards. I did find some resources online where a grad student did the same project in C++ but he did it over the course of a year. I plan to continue refining my code until I can easily compute trees for 10x10 game of chomp.