Chain Reaction

Job Jacob, Paul Antony

Federal Institute of Science & Technology

September 18, 2018

Overview

1. Introduction
2. Problem Statement
3. Chain Reaction - Game Rules
4. Literature Survey
    - MiniMax
    - α-β Pruning
    - Q-Learning
5. Proposed Solution
6. Results Expected
7. References

Introduction

- Chain Reaction is a deterministic combinatorial game of perfect information for 2 - 8 players.
- Coloured orbs are placed by players in turns one after the other.
- One must gain access and try to eliminate their opponent's orbs.
- When a player loses all his orbs, he is out of the game. The player who stands till the end is the winner.

Problem Statement

Developing an efficient AI for playing Chain Reaction is a rigorous task. Only a handful of people have developed an AI for playing this game. So we aim to develop an AI that is efficient enough to beat the so called best players of chain reaction.

Chain Reaction - Game Rules

- The gameplay takes place in an $m * n$ board. The most commonly used size of the board is $9 * 6$.

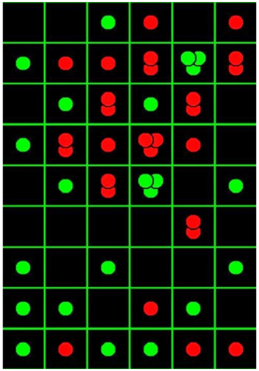


Figure: Chain Reaction Board

- For each cell in the board, we define a critical mass. The critical mass is equal to the number of orthogonally adjacent cells. That would be 4 for usual cells, 3 for cells in the edge and 2 for cells in the corner.

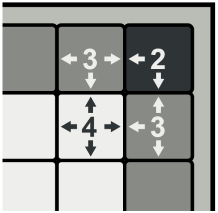


Figure: Figure explaining the critical mass of cells

- All cells are initially empty. The Red and the Green player take turns to place "orbs" of their corresponding colors. The Red player can only place an (red) orb in an empty cell or a cell which already contains one or more red orbs. When two or more orbs are placed in the same cell, they stack up.

- When a cell is loaded with a number of orbs equal to its critical mass, the stack immediately explodes. As a result of the explosion, to each of the orthogonally adjacent cells, an orb is added and the initial cell looses as many orbs as its critical mass. The explosions might result in overloading of an adjacent cell and the chain reaction of explosion continues until every cell is stable.
- When a red cell explodes and there are green cells around, the green cells are converted to red and the other rules of explosions still follow. The same rule is applicable for other colors.

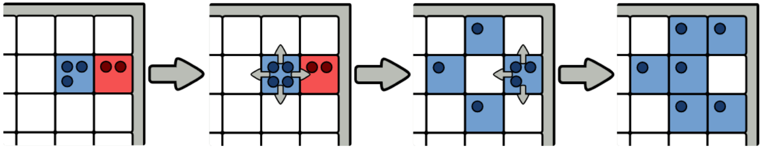


Figure: Figure demonstrating an explosion

- The winner is the one who eliminates every other player's orbs.

MiniMax - Introduction

- Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.
- It is widely used in two player turn based games such as Tic-Tac-Toe, Backgamon, Mancala, Chess, etc.
- A MiniMax tree mimics natural human thinking. It considers your move, your opponent's possible responses, and your subsequent responses. The loop goes on till we find a terminal game state, in which case we return the terminal node's value.
- Every possible board position maps to a value. The value is directly proportional to how much the AI favors the position. If the AI wins, the position has a + value. For a loss, it evaluates to - value.

MiniMax - Example

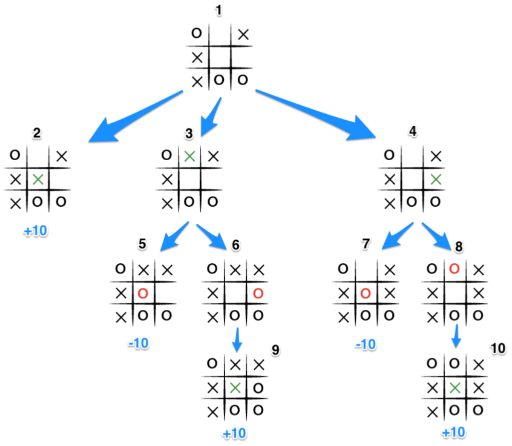


Figure: Example for Tic-Tac-Toe in MiniMax

MiniMax Algorithm - Pseudocode

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := $-\infty$
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := $+\infty$
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
(* Initial call *)
minimax(origin, depth, TRUE)
```

MiniMax Algorithm - Example

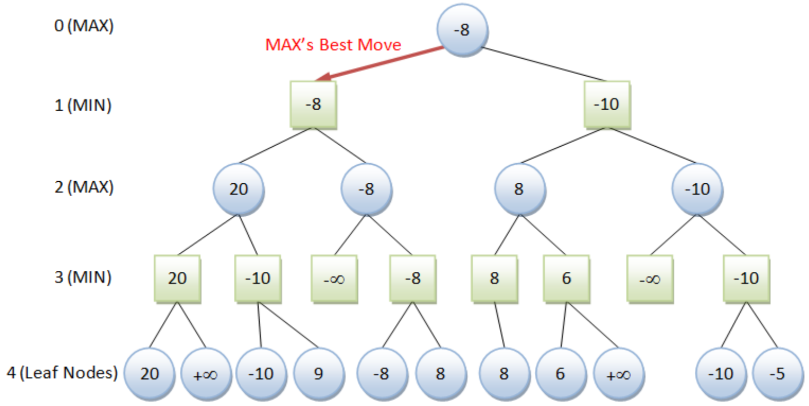


Figure: Example to explain MiniMax algorithm

MiniMax - Pros and Cons

Time Complexity: O(b^d)

Pros:

- Decision is made considering future moves.

Cons:

- Game tree with a large branching factor (b) and depth (d) will take longer time to compute.

α-β Pruning - Introduction

- Alpha-Beta pruning is an optimization technique for minimax algorithm.
- It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move
- It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.
- **Alpha** is the best value that the **maximizer** currently can guarantee at that level or above.
- **Beta** is the best value that the **minimizer** currently can guarantee at that level or above
- It stops evaluating a move when the condition $\alpha >= \beta$ is true

α-β Pruning - Example

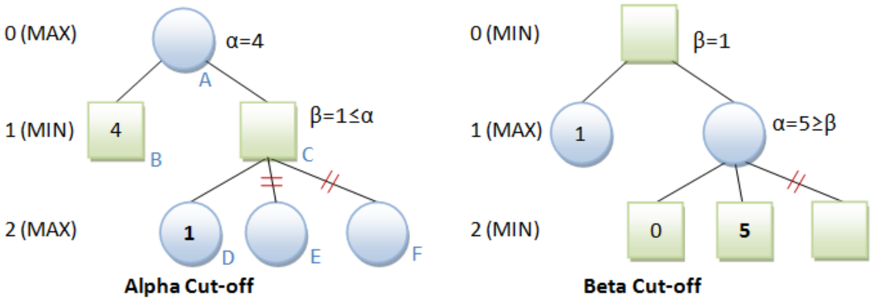


Figure: Example for α-β Pruning

α-β Pruning Algorithm - Pseudocode

```
function alphabeta(node, depth, $\alpha$, $\beta$, maximizingPlayer)
  if depth $=$ 0 or node is a terminal node then
    return the heuristic value of node
  if maximizingPlayer then
    value := -$\infty$
    for each child of node do
      value := max(value, alphabeta(child, depth - 1, $\alpha$, $\beta$, FALSE))
      $\alpha$ := max($\alpha$, value)
      if $\alpha >= \beta$ then break (* $\beta$ cut-off *)
      return value
  else
    value := $+\infty$
    for each child of node do
      value := min(value, alphabeta(child, depth - 1, $\alpha$, $\beta$, TRUE))
      $\beta$ := min($\beta$, value)
      if $\alpha >= \beta$ then break (* $\alpha$ cut-off *)
      return value
(* Initial call *)
alphabeta(origin, depth, -$\infty$, $+\infty$, TRUE)
```

α-β Pruning Algorithm - Example

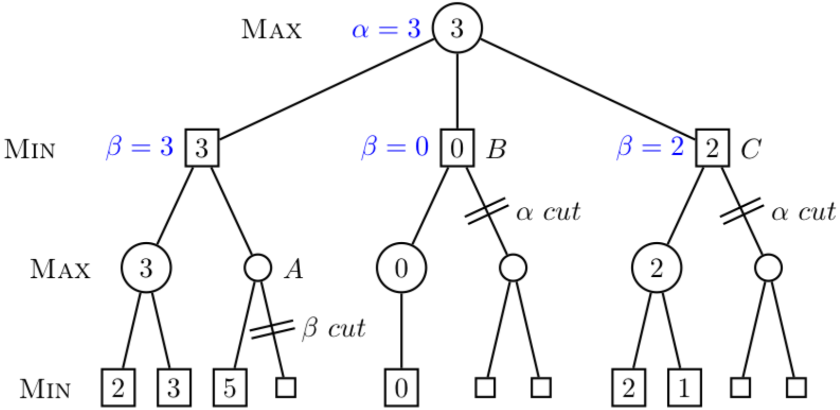


Figure: Example to explain α-β Pruning algorithm

α-β Pruning - Pros and Cons

Time Complexity: O(b$^{(d/2)}$)

Pros:

- It reduces the number of nodes evaluated.

Cons:

- Order of the nodes determine pruning.

Q-Learning

- Q-learning is a reinforcement learning technique used in machine learning.
- The goal of Q-Learning is to learn a policy, which tells an agent what action to take under what circumstances.
- It does not require a model of the environment and can handle problems with stochastic transitions and rewards, without requiring adaptations.

Q-Learning Process

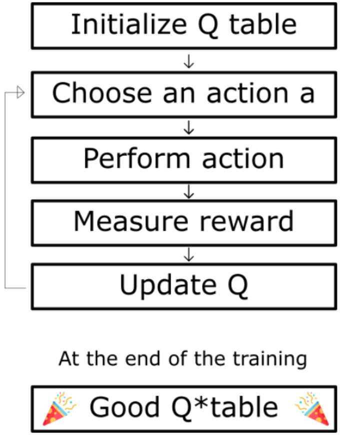


Figure: The Q-Learning algorithm Process

Q-Learning Algorithm - Pseudocode

```
Initialize Q-values (Q(s,a)) arbitrarily for all state-action pairs
For life or until learning is stopped
   a. Choose an action (a) in the current world state (s) based on current
      Q-value estimates (Q(s,.))
   b. Take the action (a) and observe the outcome state (s') and reward (r)
   c. Update Q(s,a) := Q(s,a) + $\alpha$[r + $\gamma$*maxQ(s',a') - Q(s,a)]
```

Q-Learning Algorithm - Example

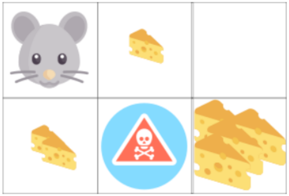
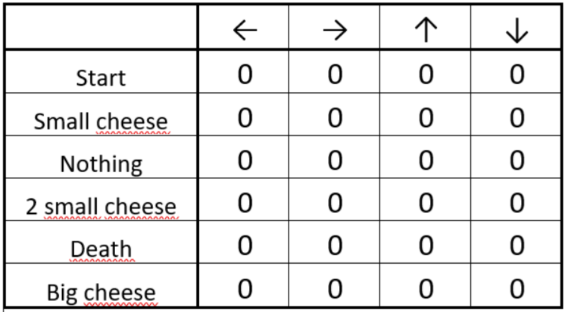


Figure: Cheese and rat game

- One cheese = +1
- Two cheese = +2
- Big pile of cheese = +10 (end of the episode)
- If you eat rat poison = -10 (end of the episode)

Step 1: We init our Q-table



	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Figure: The initialized Q-table

Step 2: Choose an action

From the starting position, you can choose between going right or down. Because we have a big epsilon rate (since we don't know anything about the environment yet), we choose randomly. For example, let's move right.

Figure: The rat moves to the right

We found a piece of cheese (+1), and we can now update the Q-value of being at start and going right. We do this by using the Bellman equation.

Step 3: Update the Q-function

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \, max \, Q'(s',a') - Q(s,a)]$$

Figure: The Bellman Equation

s=state | a=action | α=learning rate | γ=discount factor (a value between 0 and 1) | maxQ(s',a')=maximum future reward | r=reward for action a in state s

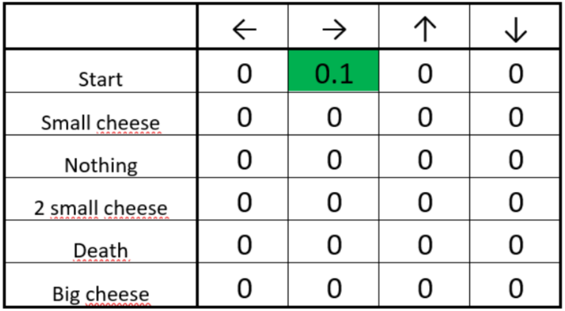
NewQ(start,right) = Q(start,right) + $\alpha[\delta$Q(start,right)]

δQ(start,right) = R(start,right) + γ*maxQ'(1cheese,a') - Q(start,right)

δQ(start,right) = 1 + 0.9*max(Q'(1cheese,left),Q'(1cheese,right),Q'(1cheese,down)) - Q(start,right)

δQ(start,right) = 1 + 0.9*0 - 0 = 1

NewQ(start,right) = 0 + 0.1*1 = 0.1



	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Figure: The updated Q-table

Step 4: Repeat the above steps

Repeat the above steps again and again until the learning in stopped.

Pros and Cons

Pros:

- Decision making improves over time.

Cons:

- Cannot take actions for states it has not yet observed.

Proposed Solution

- First develop an AI using MiniMax strategy along with α-β Pruning.
- Use this AI to train a Q-Learning Model.
- This trained Q-Learning Model is the final AI efficient enough to beat humans. It gets better and better as it plays on.
- Develop the user interface for this game using Phaser.

Results Expected

We expect to develop:

- an AI that learns as it plays on and end up becoming the best AI for playing Chain Reaction.
- a good user interface for the players to enjoy their game to the fullest.

References

1. Minimax | Wikipedia
https://en.wikipedia.org/wiki/Minimax
2. Alpha-beta pruning | Wikipedia
https://en.wikipedia.org/wiki/Alpha-beta_pruning
3. Q-Learning | Wikipedia
https://en.wikipedia.org/wiki/Q-learning

Image Citations
1. https://hope.scce.info/chainreaction/
2. https://www3.ntu.edu.sg/home/ehchua/programming/java/javagame_tictactoe_ai.html
3. https://github.com/LewisMatos/MiniMaxTicTacToe
4. https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe