A report on

ARTIFICIAL INTELLIGENCE IN COMPETITIVE MULTIAGENT ENVIRONMENT

By

Gunjan Dokrimare
Shreyas Joshi
Sarvesh Kashelikar
Dheeraj Pande
Shail Rakhunde

Under the Guidance of

Prof. Dr. S. R. Sathe

Department of Computer Science and Engineering

Visvesvaraya National Institute of Technology, Nagpur, Maharashtra – 440010

Computer Science and Engineering
Visvesvaraya National Institute of Technology

Certificate

This is to certify that the project titled "**ARTIFICIAL INTELLIGENCE IN COMPETITIVE MULTIAGENT ENVIRONMENT**", being submitted by my students, in partial fulfilment of the requirement for the completion of the course for the degree of Bachelor of Technology in Computer Science and Engineering, is a record of the students' own work carried out by them under my supervision and guidance.

Prof. Dr. S. R. Sathe

(Project Guide)

Prof. Dr. P.S. Deshpande

(Head of Department)

Date: 1/5/2014

Abstract

Artificial intelligence (AI) is the human-like intelligence exhibited by machines or software. A multi-agent system is a computerized system composed of multiple interacting intelligent agents within an environment. The agents in a multi-agent system could equally well be robots or humans.

In a multi agent environment, any given agent will need to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce many possible contingencies. There could be competitive or cooperative environments. Competitive environments, in which the agent's goals are in conflict require adversarial search – these problems are called as games. Here we focus on one such game called 'Chain Reaction' and discuss various approaches of developing an AI agent.

We have used Eclipse (ADT – Android Developer Tools) for software development in Java. To optimize the decision making capabilities of the AI agent(s), we have used adversarial search techniques like Minimax trees, **α–β** (Alpha Beta) pruning to reduce the number of nodes to be evaluated in the tree. To avoid the possibility of memory explosion, the tree generation is cut off at a preset level, and evaluation function returns the utility value of the node, by computing the 'fitness' of the node, based on various factors. The AI so developed, theoretically "looks forward" into the preset number of moves (which is equal to the depth of the tree) and selects the best-fit choice from all the possible alternatives.

Table of Contents

Introduction

Chain Reaction is a strategy game for 2 or more players. The objective of Chain Reaction is to take control of the board by eliminating your opponents' orbs. Players take it in turns to place their orbs in a cell.

Once a cell has reached critical mass the orbs explode into the surrounding cells adding an extra orb and claiming the cell for the player. A player may only place their orbs in a blank cell or a cell that contains orbs of their own colour. As soon as a player loses all their orbs they are out of the game.

The environment of the game is

- **Multi Agent:**
  Various agents affect the environment, and each agent has to take into consideration the action of other agents.
- **Static:**
  The environment's state doesn't change when an agent is deliberating his move.
- **Discrete:**
  The states are discrete, as in the change from one state to another is a discrete set of percepts and actions.
- **Fully Observable:**
  The agent's sensors give it access to the complete state of the environment at each point in time.
- **Deterministic:**
  The next state of the environment is completely determined by the current state and the action executed by the agent.
- **Competitive:**
  The agents are directly in competition with each other.

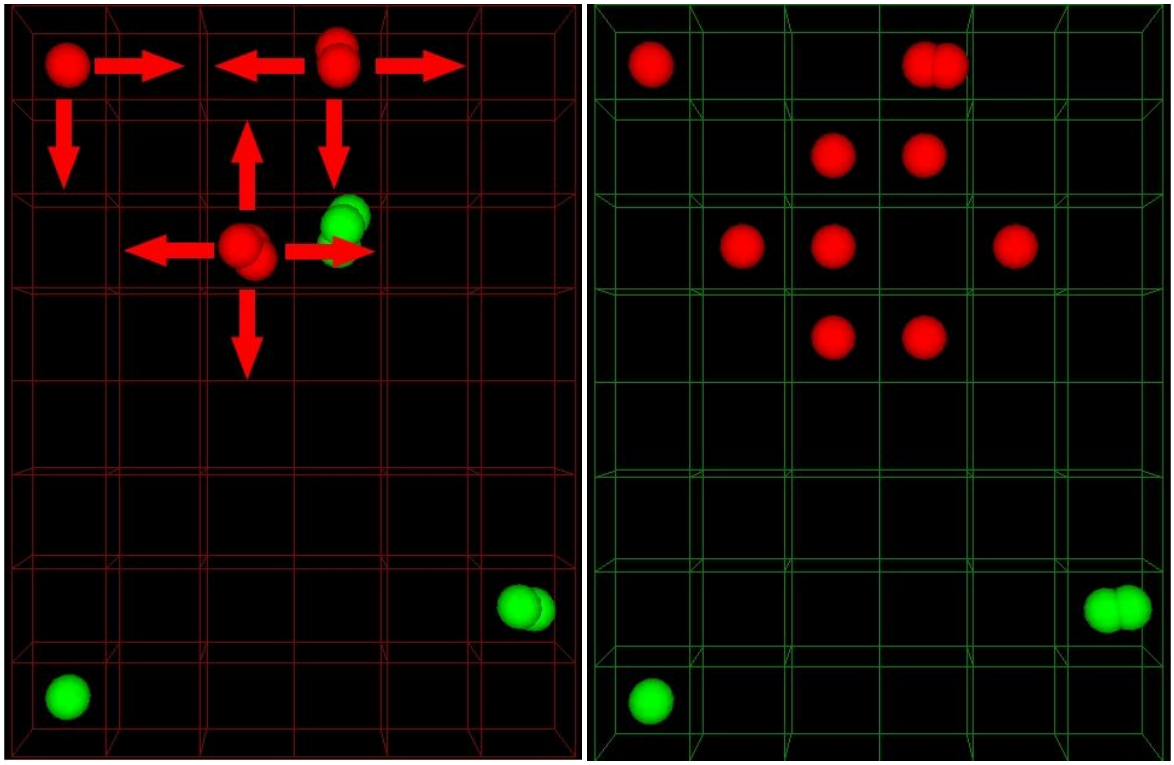

Fig 1-a **Fig 1-b**

Fig 1-a: Possible expansions for red orbs. (We select the one where red orbs expand in all four directions.)

Fig 1-b: Expanded red orbs claim surrounding green orbs.

Need for AI

Humankind has given itself the scientific name Homo Sapiens --man the wise-- because our mental capacities are so important to our everyday lives and our sense of self. The field of artificial intelligence, or AI, attempts to understand intelligent entities. Thus, one reason to study it is to learn more about ourselves. But unlike philosophy and psychology, which are also concerned with intelligence, AI strives to build intelligent entities as well as understand them. Another reason to study AI is that these constructed intelligent entities are interesting and useful in their own right. AI has produced many significant and impressive products even at this early stage in its development. Although no one can predict the future in detail, it is clear that computers with human-level intelligence (or better) would have a huge impact on our everyday lives and on the future course of civilization.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavour. In this sense, it is truly a universal field.

Adversarial search has specific applications when we try to plan ahead in a world where other agents are planning against us.

AI Development

Approach

We separated the functionalities of the 'Chain Reaction' game by defining the most basic unit as 'Cell'. A group of cells forms a 'Grid' which acts as the environment in which the basic computations are performed.

'Game' maintains a 'Grid' and takes input of the moves from 'Player'. 'Player' maintains a list of all the players, their types and acts as the structure which 'Game' refers to for any information about the agents.

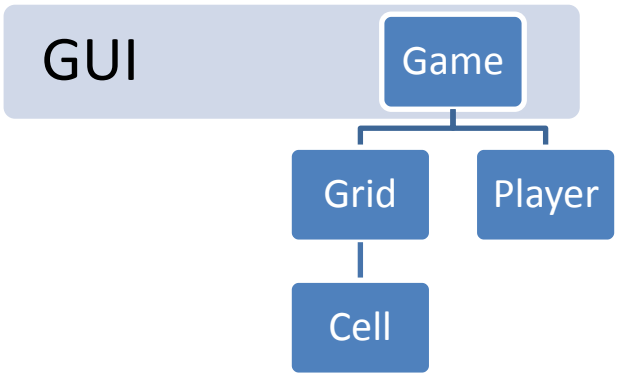

Fig 3-a

CELL Structure

```
class Cell {

        private int orbs_count;
        private int color;
        private int limit;

        public Cell();
        public Cell(int l);
        public int getCount();
        public int getColor();
        public int setColor(int c);
        public boolean isStable();
        public int raiseCount(int col);
        public void clearCell();
}
```

GRID Structure

```
class Grid {

        private int grid_rowsize;
        private int grid_colsize;
        private Cell grid[][];
        private int color;
        private int num_players;
        private int move_count;
        private int[] color_count;
        public static String[] colorcode = {"Red","Blue","Green","Yellow","Orange","Pink","Violet","Indigo"};

        public GridGUI();
        public Grid();
        public Grid(int numplay);
        public Grid(int numplay, int rows, int cols);
        public Grid(int rows, int cols);
        public int setColor(int c);
        public int getColor();
        public boolean validMove(Vector v);
        public boolean validCell(Vector v);
        public int getLimit(Vector v);
        public int getColorCount (int player_color);
        private void split(Vector v);
        public boolean addOrb(Vector v);
        public void move(Vector v);
        public void printGrid();
        public boolean isWinner();
        public static void main(String arg[]);

}
```

PLAYER Structure

```
class Player {

        private int color;
        private int type; // 0 - Human, 1 - AI, 2 - External Network
        private final int HUMAN = 0;

        public Player();
        public Player(int col, int ty);
        public int setColor(int c);
        public int getColor();
        public Vector move();
        public Vector requestHuman();
        public Vector requestAI();

}
```

GAME Structure

```
class Game {

        private int grid_rowsize;
        private int grid_colsize;
        private Grid grid;
        private GameGUI gui;
        private int no_players;
        private Player players[];

        public Game();
        public Game(int human_players);
        public Game(int human_players, int ai_players);
        public Game(int human_players, int rows, int cols);
        public Game(int human_players, int ai_players, int rows, int cols);
        public Player Play();
        public static void main (String args[]);

}
```

Minimax Tree

The minimax search is especially known for its usefulness in calculating the best move in two player games where all the information is available, such as chess or tic tac toe (Muller, 2001). It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players. It follows that this can only be used to make decisions in zero-sum games, where one player's loss is the other player's gain. Theoretically, this search algorithm is based von Neumann's minimax theorem which states that in these types of games there is always a set of strategies which leads to both players gaining the same value and that seeing as this is the best possible value one can expect to gain, one should employ this set of strategies (Kulenovic, 2008).

The first step is to give each node a value in terms of utility for player 1. These values are derived from the values of the terminal nodes, which each represent an end of the game. Each level of the tree alternates between player 1's move, who is trying to maximize her score, and player 2's move, who is trying to minimize player 1's score in order to undermine her success. If the level above the terminal nodes represents player 2's possible moves, then the value of the nodes will be the lowest value among those of their children, the minimum. On the other hand, if this level represents the possible moves for player 1, the value of its nodes will be the highest value among those of their children. Finally, once all the nodes have been assigned values, it is decision time for player 1 who just picks the move which will lead to the highest value (Russell & Norvig, 1995). Figure 1 illustrates this process.

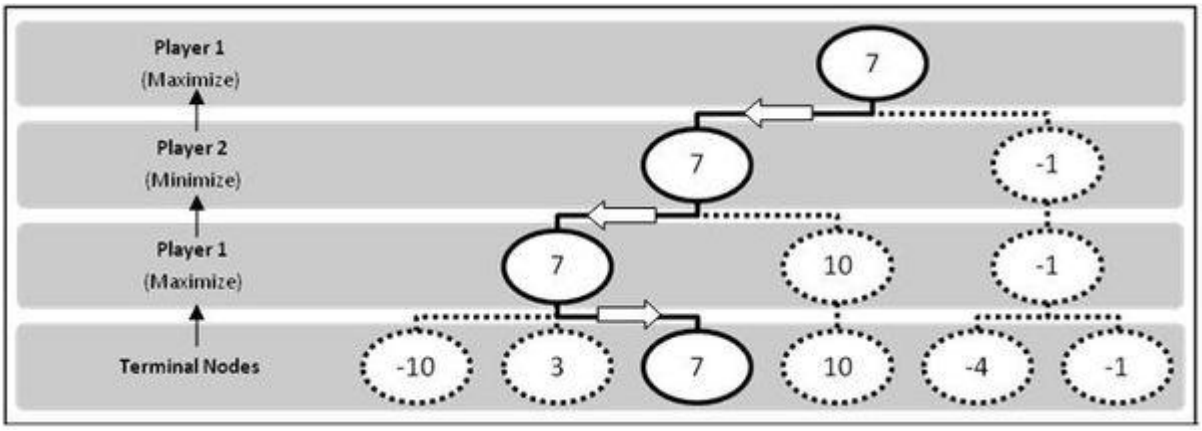

Fig 3-b: Assigning a value to the nodes and deciding the best move.

This pseudocode is adapted from a version presented by Russell and Norvig (1995).

```
function minimaxDecision (game)
        for each possible move in game
                moveValue = minimaxValue(state, game)

        return move with highest value

function minimaxValue (state, game)
        if terminal node
                return utility

        if player 1's turn
                return highest value of children

        if player 2's turn
                return lowest value of children
```

The AI agent creates a minimax tree where it always tries to choose the MAX value of children, while assuming that all the other agents will try to choose the MIN value. This is under the assumption that other players always play the optimum move.

Each depth represents all the possible states from which the agent chooses the one best suited according to its AI engine. However, the number of possible states increases exponentially as we move down the levels. Thus, to increase the performance and time efficiency of the algorithm, we use alpha beta pruning to reduce the number of nodes to be evaluated in the minimax tree.

Properties of Minimax:

• Complete?
– Yes (if tree is finite)

• Optimal?
– Yes (against an optimal opponent)
– No (does not exploit opponent weakness against suboptimal opponent)

• Time complexity?
– $O(b^m)$

• Space complexity?
– O(bm) (depth-first exploration)

Alpha beta pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). 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. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

Pseudocode

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
            α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
            if β ≤ α
                break (* β cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
            if β ≤ α
                break (* α cut-off *)
        return β
(* Initial call *)
alphabeta(origin, depth, -∞, +∞, TRUE)
```

ALPHABETA vs. MINIMAX

MINIMAX visits all tree nodes. With branching factor b and depth d, this amounts to b^d.
How many nodes ALPHABETA visits depends on the order in which moves are examined.
If the best move is always examined first, ALPHABETA visits $b^{d/2}$ nodes.
We can use domain-specific knowledge to achieve a reasonable move order.

Numbers in perspective

Let us say that we have a branching factor b = 20 and want to search to depth d = 8.
MINIMAX will search $b^d = 20^8 = 25{,}600{,}000{,}000$ nodes.
A "perfect" ALPHABETA would search $b^{d/2} = 20^4 = 160{,}000$ nodes.
A "reasonable" ALPHABETA would search $b^{3d/4} = 20^6 = 64{,}000{,}000$ nodes.

Cutting off the search

Rather than letting ALPHABETA search the whole game tree (which is unrealistic for, e.g., chess), we want to cut the search off and return a heuristic evaluation. For this, we have used Evaluation Function which return a Utility Value for the state, calculated on the basis of various heuristic measures.

Decision Criterion

Utility Value

We devise a basic set of heuristics, and assign them a weight, according to the perceived importance of each heuristic, which when combined, return a utility value between 0 to 1.

0 is the minimum utility value which indicates that the agent loses or will eventually lose the game if he chooses that state.

1 is the maximum utility value which indicates that the agent wins or will eventually win the game if that state is chosen.

Any AI agent, in its own move chooses the MAXIMUM utility valued state from the available options, with the assumption that the other agents will choose the MINIMUM utility valued state.

We calculate the Utility Value based on three heuristics, then normalize the value such that the final result always falls between 0 and 1.

1. Percentage of cells acquired by the agent. We calculate this as:
   (Number of cells acquired by the agent / Total number of cells acquired by all agents)
2. Percentage of orbs of the agent. We calculate this as:
   (Number of orbs of the agent / Total number of orbs of all agents)
3. The number of stable / safe cells for the agent
   (This is calculated by a function which checks for total cells which are recommended for the player. For example, if a cell has reached critical mass and is about to explode, the agent can consider adding an orb to that cell.)

We assign weights w_1, w_2, w_3 to the above heuristics h_1, h_2, h_3 and calculate the utility value as:

$U = w_1.h_1 + w_2.h_2 + w_3.h_3$

Then we normalize the Utility value by a normalizing function which returns N(U) between 0 and 1.

The utility value is assigned to each node and it is the decision criterion for selection of nodes in the Minimax tree, complemented with Alpha Beta pruning.

Evaluation Function

An evaluation function, also known as a heuristic evaluation function or static evaluation function, is a function used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms. The evaluation function is typically designed to prioritize speed over accuracy; the function looks only at the current position and does not explore possible moves (therefore static).

In the 'Chain Reaction' program, we call upon the Evaluation Function to return a utility value of the state, if the minimax tree has reached its maximum depth. Otherwise, the Evaluation Function simply checks what depth of the tree it is called at, if the state of the node is a position where one of the agents is winning (to cut off the tree at that depth, without generating any future nodes). If the agent who is winning here, is the AI agent itself, then the Evaluation Function returns Utility value 1, and the AI agent chooses that branch initially which leads to this state. If however, the agent winning in this state is not the AI agent generating the minimax tree, the Evaluation Function returns Utility value 0, consequently the AI agent immediately rejects the branch which leads to this state.

The Evaluation Function takes 'Grid' as input. It then calls upon various sub-functions, which calculate the value of the heuristics – For example, h_1 requires the computation of number of cells acquired by the AI agent, as well as total cells acquired by all the agents. So the Evaluation Function calls upon ComputeCellRatio(grid G), which will take the Grid as the input and iteratively scan through all the nodes returning the number of cells acquired by the AI agent (which is represented by current colour of the grid), and also the total number of cells.

Similarly, the other heuristics are computed. We can optimize the evaluation function by adding more heuristics such that a more optimal Utility Value is computed. An ideal evaluation function is the one which always returns a value of 0 or 1 by exploring the entire depth of the tree. However, due to memory constraints, we cannot always explore each branch from root to leaf, and that is why instead of getting utility value at the terminal state, we apply evaluation function at the preset cut-off depth.

To determine the quality of Evaluation Function, four broad categories are used to classify the results:

- **Optimal:**
  An OPTIMAL evaluation function is the one which returns the most accurate Utility Value. Additional heuristics have no effects on the optimality of the Evaluation Function.

- **Strong super-human:**
  Performs better than all humans. This Evaluation Function always has a depth of the tree so great that it always makes the BEST choice, and so AI agent always wins against any human agent, however intelligent the human agent might be.

- **Super-human:**
  Performs better than most humans. This Evaluation Function does not always return the most accurate Utility Value. It might be the case that the preset depth of cut off is insufficient in some cases and so, a very intelligent, expert human agent might just defeat the AI agent from time to time.

- **Sub-human:**
  Performs worse than most humans. This Evaluation Function always gets cut off too early compared to the number of moves, or it might be the case that the number of nodes generated is so large, that the depth has to be set very low in order to not let the memory explode with computations.

Conclusions and Future Scope

Thus, we conclude that intelligence is concerned mainly with rational action. Ideally, an intelligent agent takes the best possible action in a situation. Given a number of accurate heuristics, an ideal Evaluation Function can be generated. We need to develop a model of the given problem-statement which classifies the various factors affecting the state and these heuristics, if all-inclusive, can lead to an OPTIMAL Artificial Intelligence Agent which can never be defeated.

Future scope for this particular 'AI in Chain Reaction' project is to implement the AI engine on a cloud computational network, such that the network of the players is not restricted to those physically present at the same place, instead we can have multi-player games where each player connects to a game server, which takes up the matching of the players, and also inserts AI players. This gives the Human Player a chance to compete with Agents located anywhere across the globe, who are Online at the time.

Also, by enabling Bluetooth and / or LAN wire connectivity features, we can provide increased accessibility to Human Players to create their own restricted network for multiplayer gaming. We need to take care of additional features such as Security over network by implementing the proper cryptographic functions which ensure that the data is not modified by any third party, and that the user experience of the game remains unaffected. Other general features of high availability of the game server, low latency and high throughput also need to be taken care of, to ensure that the system doesn't falter under heavy loads.

Acknowledgements

We profusely thank Prof. Dr. S. R. Sathe for always finding time for us from his busy schedule. We could not have completed the project without your constant support and guidance.

We also thank the Department of Computer Science and Engineering for providing us resources and facilities to do research on various topics related to the project.

Last, but not the least, we thank Matt from Buddy-Matt entertainment for developing such a ingenious and wonderful game which gave us the idea to develop the AI for it, as the topic for our project.

References

1. Introduction -
https://play.google.com/store/apps/details?id=com.BuddyMattEnt.ChainReaction

2. Need for AI -
http://www.cs.berkeley.edu/~russell/intro.html

3. Minimax search -
http://en.wikibooks.org/wiki/Artificial_Intelligence/Search/Adversarial_search/Minimax_Search
https://courses.cs.washington.edu/courses/csep573/11wi/lectures/05-games.pdf

4. Alpha beta pruning -
http://www8.cs.umu.se/kurser/5DV122/HT13/material/02-web.pdf

Russell, Stuart J.; Norvig, Peter (2003), **Artificial Intelligence: A Modern Approach (2nd ed.)**, Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2