Santorini

Artificial Intelligence with Minimax algorithm in Santorini board game on Java

Short summary of implemented algorithms and their functions:

Minimax

```
public double minimax(Node node, int depth, boolean isMaximizingPlayer)
{
        if (node.children == null || depth == table.sant.diff) {


            return node.f_score;
        }
        if (isMaximizingPlayer) {
            node.f_score = Double.NEGATIVE_INFINITY;
            for (int i = 0; i < node.children_lenght(); i++) {

                Node n = node.children.get(i);
                double value = minimax(n, depth + 1, false );

                node.f_score = Math.max(node.f_score, value);

            }
            return node.f_score;
        } else {
            node.f_score = Double.POSITIVE_INFINITY;
            for (int i = 0; i < node.children_lenght(); i++) {

                Node n = node.children.get(i);
                double value = minimax(n, depth + 1, true);

                node.f_score = Math.min(node.f_score, value);

            }
            return node.f_score;
        }
    }
```

- Using Minimize algorithm to select the best move from all the moves that are presented as tree nodes.

```
depth == table.sant.diff
```

- Check if we reached depth we selected before game started.

```
node.f_score = Math.max(node.f_score, value);
```

```
node.f_score = Math.min(node.f_score, value);
```

- Taking minimum or maximum depending of which player made a move.

Minimax with Alpha-Beta pruning

```
public double minimax(Node node, int depth, boolean isMaxi
mizingPlayer, double alpha, double beta) {
        if (node.children == null || depth == table.sant.diff) {

            return node.f_score;
        }
        if (isMaximizingPlayer) {
            node.f_score = Double.NEGATIVE_INFINITY;
            for (int i = 0; i < node.children_lenght(); i++) {

                Node n = node.children.get(i);
                double value = minimax(n, depth + 1, false, alpha, beta);

                node.f_score = Math.max(node.f_score, valu e);

                alpha = Math.max(alpha, node.f_score);
                if (beta <= alpha) break;
            }
            return node.f_score;
        } else {
            node.f_score = Double.POSITIVE_INFINITY;
            for (int i = 0; i < node.children_lenght(); i++) {

                Node n = node.children.get(i);
                double value = minimax(n, depth + 1, true, alpha, beta);

                node.f_score = Math.min(node.f_score, valu e);
                beta = Math.min(beta, node.f_score);
                if (beta <= alpha) break;
            }
            return node.f_score;
        }
    }
```

- Minimax with Alpha-Beta pruning is improved version of Minimax where we cut parts of tree we know that cant contain better move than current one.

```
alpha = Math.max(alpha, node.f_score);
if (beta <= alpha) break;
```

```
beta = Math.min(beta, node.f_score);
if (beta <= alpha) break;
```

- We update values of alpha and beta throughout whole tree so we can compare them and if MIN <= alpha or MAX>= beta we break for loop and dont iterate rest of children.

Improving Alpha-Beta pruning

- One way to improve Alpha-Beta pruning is to sort first few children. If nodes with biggest function score are among first our algorithm will cut off bigger chunk of tree which will result in faster realization. In our case reserving first 3 spots of list for 3 nodes with biggest heuristic function led to increasing speed of algorithm by 40%.

```
if (depth == table.sant.diff-1 && temp.f_score == table.sa
nt.max) children.add(0, temp);
if (depth == table.sant.diff-1 && temp.f_score == table.sa
nt.max1 && children.size() > 1) children.add(1, temp);
if (depth == table.sant.diff-1 && temp.f_score == table.sa
nt.max2 && children.size() > 2) children.add(2, temp);
else children.add(temp);
```

- After we create all children for our node using recursion, after coming back we need to insert them in a list. Condition of depth == table.sant.diff-1 is neccessary so only our leafs will be sorted. If child is not in top 3 spots we add it to the end of the list.

```
if (f_score > table.sant.max) { // pomeranje maxova
    table.sant.max2 = table.sant.max1;
    table.sant.max1 = table.sant.max;
    table.sant.max = f_score;
}
else if (f_score > table.sant.max1) {
        table.sant.max2 = table.sant.max1;
        table.sant.max1 = f_score;
    } else if (f_score > table.sant.max2) {
            table.sant.max2 = f_score;
```

- With code above we assure that max, max1 and max2 will have biggest scores of all children in descending order.

Brief descriptions of all realized classes, with signatures and method descriptions.

Field

Class field represents one of 25 fields in game which can be built to maximum height of four. Four figurines in game move across all fields to reach field with height of three. Each of them has JButton so players can press fields they want to move or build and references to class Table and Figure.

```
public Field(Table table, int x, int y)
```

- Constructor with reference to its Table.

```
public void add_figure(Figure f)
```

- Links figure to field.

```
public void remove_figure()
```

- Removes figure from that field.

```
public boolean is_taken()
```

- Checks if field has any field on it and returns boolean.

```
public void build()
```

- Increments current height.

```
public void change_color_on_height(int i)
```

- Based of argument, change color of JButton.

```
public Field clone()
```

- Creates deep clone of object Field.

```
private class FieldAction implements ActionListener
```

- Implements series of conditions which will be resilient to almost any error player can commit.Private class triggers when player click certain Field. Depending on state of the game that click can:
    1. Select field (figure) we want to move.
    2. Move figure from one field to another.
    3. Build on certain field.
    4. Place figures on their starting postions.
- It also controls that in PvP mode players go one after the other.

Figure

Class Figure represent one of four figures active in game (two per player). It has references to their Table and Field as well as coordinates. Each figure has id which is id of player (1 or 2) and id2 which represent one of two figures per player.

```
public Figure(int x, int y, Table table, int id, int id2)
```

- Figure constructor with neccessary arguments.

```
public boolean isWinner()
```

- Checks if figure moved on field with height of three.

```
public boolean is_movable()
```

- Checks if figure is moveable on any possible field.

```
public boolean possible_to_move(int x, int y)
```

- Checks if figure is moveable on specific field.

```
public List<Field> possible_moves()
```

- Method that returns all possible moves as List of Fields.

```
public boolean move(int x, int y)
```

- Removes figure from old field and assign it to new field, as well

as updating values of current heights.
- Writes movement to text file.

```
public boolean possible_to_build(int x, int y)
```

- Checks if figure can build on specific field.

```
public List<Field> possible_builds()
```

- Method that returns all possible buils as List of Fields.

```
public boolean build(int x, int y)
```

- Checks if figure can build on certain field and if it can calls build function for that field.
- Writes building to text file.

```
private void add_field(Field cur)
```

- Setter for variable field.

```
private void remove_field()
```

- Nullify reference to field.

```
public Figure clone()
```

- Creates deep clone of class Figure.

Table

This class represents core of backend. It reference to every figure, player and field. Only table has reference to Santorini class. Has reference to JPanel which represent 25 fields that can be clicked.

```
public Table(Santorini s)
```

- Constructor that creates fields and players depending on game mode.

```
public String encrypt(int y, int x)
```

- Since coordinates of fields are matrix of 0-4, converts coords to prettier format.

```
public void decrypt(String s)
```

- Changes one format to other one.

```
public void create_players()
```

- Creates players depending on game mode.

```
public Figure find_figure(int id, int id2)
```

- Since depending on game mode player 1 or 2 might not exists, this method returns Player or AI based of id and id2.

```
public Table clone()
```

- Creates deep clone of class Table.

```
public void repaint(Table table)
```

- After finding Node that has optimal move, we need to replicate fields and figures from referenced Table to main Table.

Player

Class Player is representation of real player or parent class of AI player.

```
public void create_figures1(int x1, int y1)
public void create_figures2(int x2, int y2)
```

- Creates figures for player.

```
public boolean isLoser()
```

- Checks if either figures are movable. If not, returns true.

```
public Player clone()
```

- Creates deep clone of class Player.

SimpleAI

Class SimpleAI represents artificial intelligence that will be able to compete versus real player or other AI.

```
public void function()
```

- This method based on random number or where opponents figures are placed, creates x_func and y_func as coordinates where AI will place its figures depending on game mode.

```
private class FieldAction implements ActionListener
```

- Private class is triggered when User presses button “Next step” which can:
    1. Places figures
    2. Create tree of all possible moves
    3. Calls minimax algorithm
    4. Checks if someone won or lost.

```
public double minimax(Node node, int depth, boolean isMaximizingPlayer)
public double minimax(Node node, int depth, boolean isMaximizingPlayer, double alpha, double beta)
```

- Minimax with and without alpha-beta pruning.

Tree

Class that creates root as new Node which by recursion leads to creating of entire tree.

```
public Tree(Table table)
```

- Creates root.

```
public Node next_root()
```

- When minimax is complete, method finds Node with heuristic score that matches score that was dound optimal.

Node

Creates one possible “situation” which basically is positions and values of figures and fields on table. That table, then get evaluated by heuristic function which will represent how good that situation is.

```
public Node(Table table, int d, boolean max, Field temp_fi
eld_move, Field temp_field_build, int delta)
```

- Calls creation of new Nodes that will represent its children. It needs many arguments because more information we feed to heuristic function it will be more accurate.

```
public double calc_score(Field f, Field ff, int delta)
```

- Calculates heuristic static function based on position of figures on table. It has 3 special cases where function returns specific values. Case of Win means player can win in next move which returns maximum value, case of Lose means player will lose in next turn which returns minimal value and case of Deny means if opponent can win in next turn we move specifically to block him.

```
public void possibleChildren()
```

- Creates all possible moves and builds with three for loops (one for 2 figures, one for all moves and one for all builds). At the end it calls new Node and adds it to children list to create whole tree.

Santorini

Class that represent front end of our game. GUI class that has radio buttons, JLabels, JButtons, panels and frames.

```
public JPanel fill_panel()
```

- Creates first panel on which we can choose game mode, new or load game and difficulty of game if we play with AI.

```
private class GameMode implements ItemListener
private class SaveGame implements ItemListener
private class Difficulty implements ItemListener
private class StartGame implements ActionListener
```

- Private classes which are listeners who listen to radio buttons on first panel and set variables accordingly. StartGame creates Second panel with Table and three JLabels. Depending on game mode we will or wont have button “Next step”.

```
private class FinishGame implements ActionListener
```

- In AI vs AI game mode we have option to “Finish game” which will, when pressed run game until either AI wins.

```
public void load_game()
```

- If we choose load game, we have to recreate game that was saved. Method uses loop to reads all moves and builds from text file and starts game on point you exited last time.