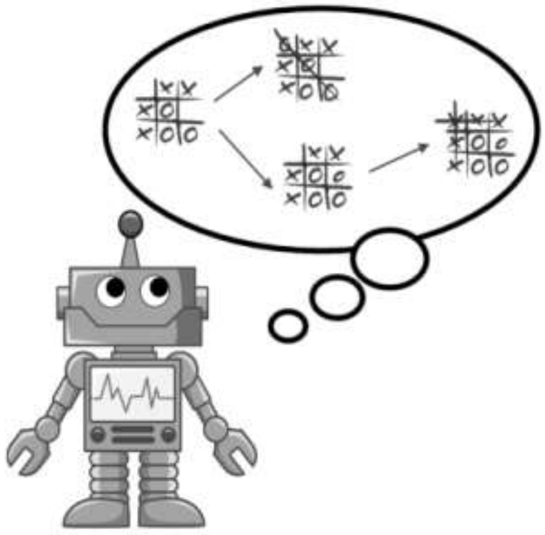


AI Tic Tac Toe

Mohammad Azim Ul Ekram

| Advanced Artificial Intelligence | 30th November, 2016

TEAMMATES

Elizabeth Phillips

Olaf Alexander

SELF-CONTRIBUTIONS:

- Developed searching algorithms based on provided materials
- Integration of searching algorithms to game
- Generation of board successors from specific state
- Work on 2 evaluation functions
- Reviewing and modularizing
- Automatic generation of graphs from games

MEMBERS CONTRIBUTIONS:

- Implementation of Board class
- Implementation of Ply class
- Implementation of Cell class
- Implementation of full game mechanics
- Partial implementation of search algorithms
- Implementation of user choices
- Partial integration of algorithms
- Implementation of Evaluation function

CONTENTS

1 INTRODUCTION

Tic-Tac-Toe is a type of paper-pencil game where two players alternately put **X**s and **O**s in cells of a pattern outlined by two vertical lines crossing two horizontal lines. Each player tries to get a row, a column or a consecutive diagonal of three **X**s or three **O**s before the opponent.

As tic-tac-toe is a simple game, it is often used as an interactive tool for teaching some concepts of artificial intelligence that deals with the searching of game trees. The goal of this project is to develop intelligent searching methods and heuristic functions which artificial players will use to play with another artificial player in a game of tic-tac-toe and evaluate the performance of each searching method and evaluation function.

1.1 PROBLEM ANALYSIS

Evaluation of the board and anticipation of the movement of the opponent is crucial. One might wonder of the best way to approach their next move when playing tic-tac-toe. It is possible to play defensively, offensively, preferred spot or playing offensive while prioritizing defense. Each move changes by the player and the opponent changes the board layout and thus changes the score of the board. Analysis based on some heuristics and search algorithm is necessary to make an optimized turn.

1.2 PROPOSAL

Straightforwardly, a computer program can be written to enumerate the 765 essentially different positions, or the 255,168 unique possible games up to rotations and reflections. Thus, analyzing every possibility would take a very long time. To shorten the amount of combinations, 5 different heuristic functions will be used. To find the best way to analyze each board and to further shorten the combinations, two search algorithms will be developed and integrated with the game play. Name of these algorithms are namely Minimax-AB and Alpha-Beta Search.

2 SEARCHING ALGORITHMS

Two types of algorithms are used in this game, namely, MinimaxAB and AlphaBeta search. The MiniMaxAB algorithm were used from Artificial Intelligence by Rich and Knight. The AlphaBeta search method were quoted from Artificial Intelligence: A Modern Approach 3rd Edition. Both algorithms were used in the game for each player at the same time and with different algorithms for each player.

2.1 MiniMaxAB Algorithm

The MiniMax algorithm is a depth-first, depth limited search procedure. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. This is used to find the path for the best move in a two-player game.

It is necessary to modify this search procedure slightly to handle both maximizing and minimizing players as well as to modify the branch-and-bound strategy to include two bounds, one for each of the players. This modified strategy is called alpha-beta pruning.

This requires that the maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned, alpha or in this case x, and another representing an upper bound on the value that a minimizing node may be assigned, called beta or in this case.

The MiniMax procedure as it stands does not need to treat maximizing and minimizing levels differently since it simply negates evaluations each time it changes levels. Instead of referring to alpha and beta, MiniMaxAB uses two values, USE-THRESH and PASS-THRESH. USE-THRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level as its USE-THRESH. Of course, USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to be negated each time they were passed across levels, so too must these thresholds be negated. This is necessary so that, regardless of the level of the search, a test for greater than will determine whether a threshold has been crossed. Now there need still be no difference between the code required at maximizing levels and that required at minimizing ones.

```
MINIMAX-A-B(CURRENT, 0,
                       PLAYER-ONE,
                       maximum value STATIC can compute,
                       minimum value STATIC can compute)
```

These initial values for *Use-Thresh* and *Pass-Thresh* represent the worst values that each side could achieve.

Algorithm: MINIMAX-A-B(Position, Depth, Player, Use-Thresh, Pass-Thresh)

```
1. If DEEP-ENOUGH(Position, Depth), then return the structure
       VALUE = STATlC (Position, Player);
       PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVEGEN(Position, Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.
   For each clement SUCC of SUCCESSORS:
    (a) Set RESULT-SUCC to
        MINIMAX-A-B(SUCC, Depth + 1, OPPOSITE (Player),
        - Pass-Thresh, - Use-Thresh).
    (b) Set NEW-VALUE to - VALUE(RESULT-SUCC).
    (c) If NEW-VALUE> Pass-Thresh, then we have found a successor that is better than any that
have been examined so far. Record this by doing the following.
        (i) Set Pass-Thresh to NEW-VALUE.
        (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate
path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to
        the result of attaching SUCC to the front of PATH(RESULT-SUCC).
 (d) If Pass-Thresh (reflecting the current best value) is not better than Use-Thresh, then we
should stop examining this branch. But both thresholds and values have been inverted. So if Pass-
Thresh >= Use-Thresh, then return immediately with the value
            VALUE = Pass-Thresh
            PATH = BEST-PATH
5. Return the structure
   VALUE = Pass-Thresh
   PATH = BEST-PATH
```

2.2 ALPHA-BETA SEARCH

Alpha–Beta search is a search algorithm that seeks to decrease the number of nodes that are evaluated by the Minimax algorithm in its search tree. 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. Minimax AB can go through the tree and get rid of unnecessary nodes, but Alpha-Beta search can get rid of full branches that are not needed. Even with this amount of reduction in searching, it can still find a better move. If a good move is already found, then even one refutation is enough to avoid it with further searches.

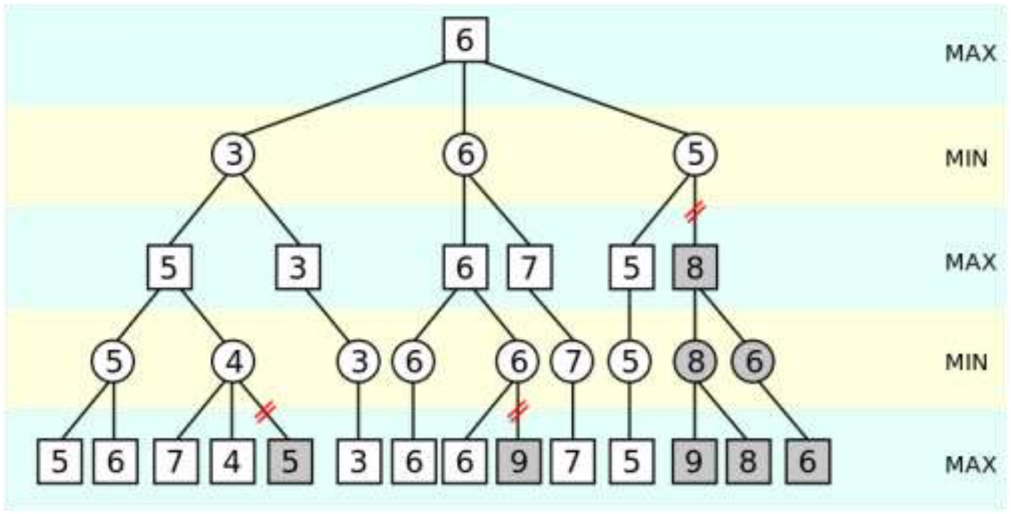


Figure 1: Alpha-Beta Search with pruning of several whole branches

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state,-∞,+∞)
return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state,α,β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a),α,β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state,α,β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) ,α,β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

3 DEVELOPMENT APPROACH

First, it was necessary to have a base structure for Tic-Tac-Toe board and standard values which will represent Xs and Os. For this reason, the Cell, cell_value, board and ply classes were designed first. After that the game mechanics were designed, so that a player would play based on suggested location on the board, which at that time was taken from user input. But it would be determined by the algorithms and evaluation function later.

Generating successors and saving the actual game state were a bit challenging where a lot of board were saved in a vector of shared_ptr, which later were released so that all possible children wouldn't be saved all the time. A generic dummy evaluation function was placed to test whether the ply class calls the evaluation functions correctly or not.

At the same time a generic search algorithm was being developed which was tested on various trees found in the book examples. After the search algorithms were verified and tested to be correct and accurate, next step was the integration of those into the game. After integrating the searching algorithm was another hard part which was being done as a teamwork. After that evaluation functions were integrated and tested vigorously.

The setup of the game allows each player to choose how they would like to evaluate each board and how they would like to minimize the amount of choices they would need to analyze before each move. The user was asked what evaluation functions have to be used for both X and O. Another choice was prompted to ask for the searching algorithm to be used with each of the X and O.

3.1 SHORT CLASS DIAGRAM

The class diagram briefly shows the association of classes with each other using standard UML 2.0 diagram. It also shows the main object responsible for running the whole game (*game* object of *ply* class). The class diagram is shown below:

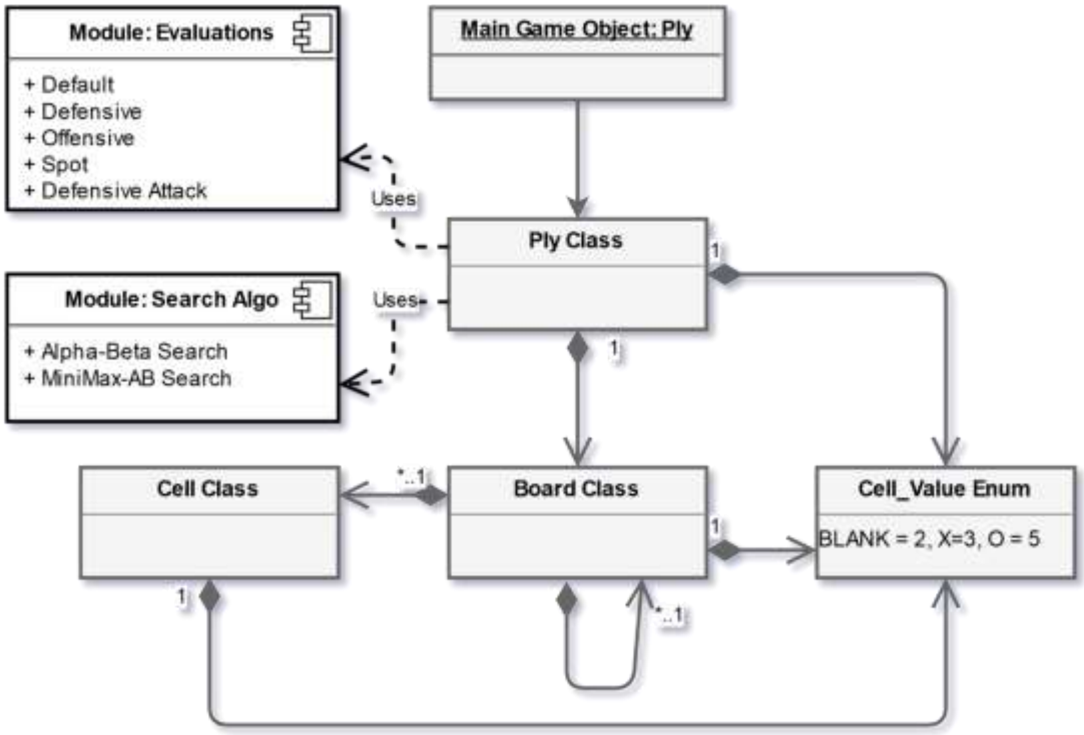


Figure 2: Short class diagram of the whole project including modules Evaluation and Search Algorithm

3.2 DECISION TREE

A decision tree represents every possibility of the game. But as there are a lot of possible nodes for each combination, it is not possible to depict the tree. Instead a stripped-down version of the tree is shown in figure 3. As mentioned before, with the searching algorithms, the number of branches that will be used and discovered in each game will be substantially reduced.

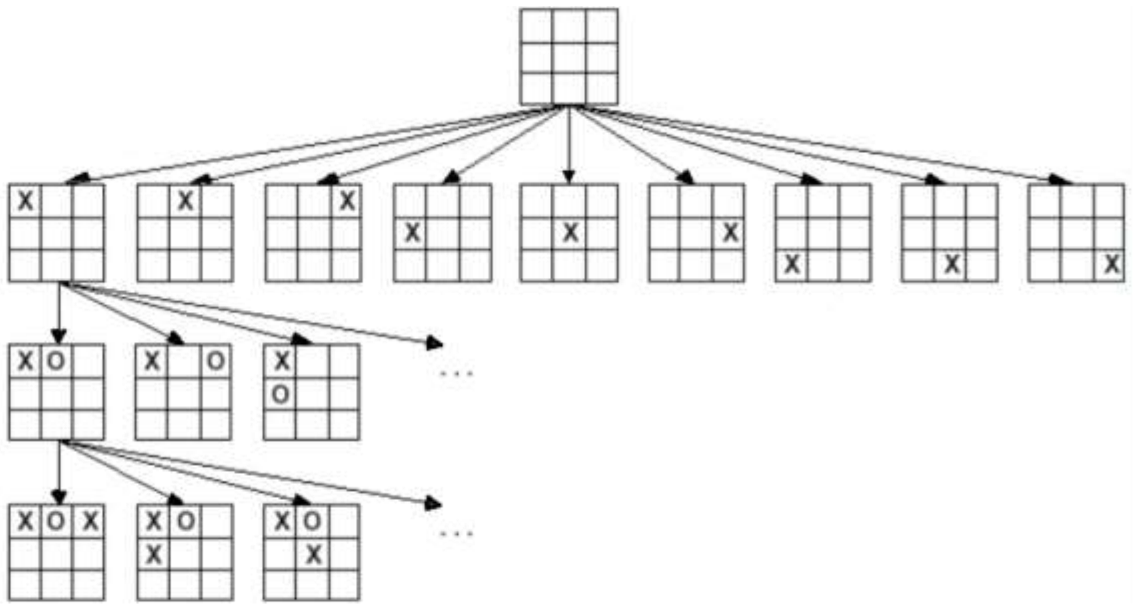
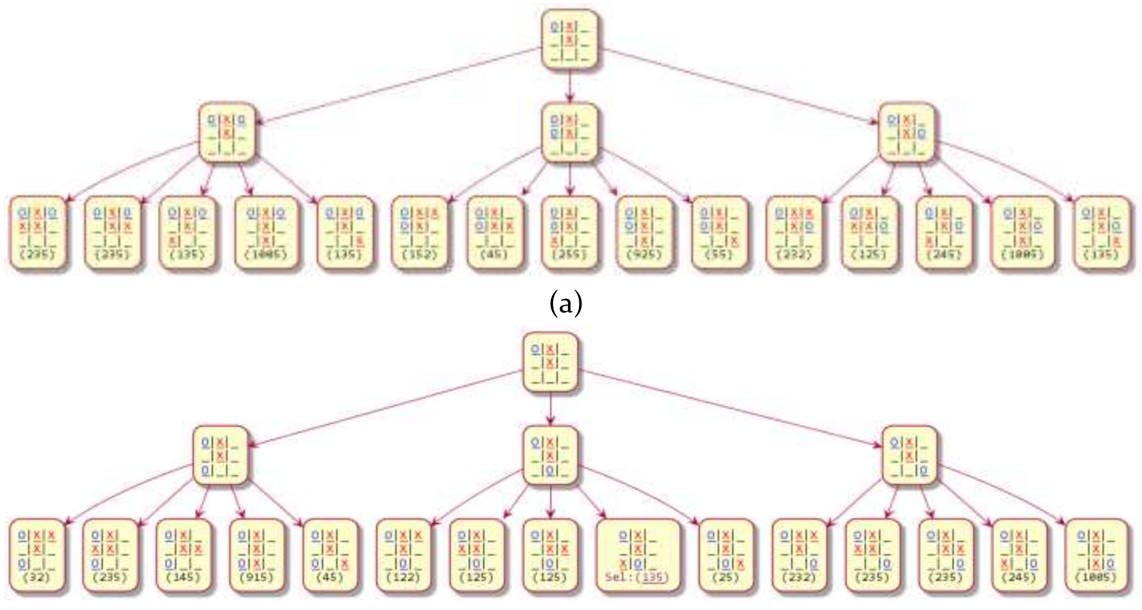


Figure 3: Portion of a full decision tree

Graphs along with their evaluation values are generated by the game at runtime. Graphs for all steps are generated by the game and saved as ‘text’ file. Then the text files are converted to PNG afterwards if necessary. Here only the fourth step of our tic-tac-toe game is shown as the graphs are quite large.


(a)

(b)

Figure 4: Full graph for the fourth turn for ‘O’. In (b) the selected node is highlighted in the Evaluation porition

3.3 CLASS DESCRIPTIONS

3.3.1 Cell Class

Cell class is an encapsulation of the possible values of tic-tac-toe. In a tic-tac-toe board, possible values are ‘X’, ‘O’ and no value that is ‘BLANK’. These values are represented by the cell_value enumeration, and is encapsulated by ‘Cell’ class. Cell class also contains the position of the cell in the tic-tac-toe board.

```
enum class cell_value
{
	BLANK = 2,
	X = 3, //MAX
	O = 5 //MIN
};
class cell
{
private:
	cell_value _value = cell_value::BLANK;
	int _position = -1;
	const char* cell_value_chars[6] = { "",""," ","X","","O" };
public:
	cell() {}
	//The default value of a cell is blank
	cell(int position, cell_value v) :_value(v), _position(position) {}

	//Accessors of instance variables
	const cell_value value() const
	{
		return _value;
	}
	void value(cell_value cv)
	{
		_value = cv;
	}

	//Returns if the cell is currently filled or blank
	bool is_blank() const
	{
		return _value == cell_value::BLANK;
	}

	//Prints out the value of the cell
	string print_value() {
		return string(cell_value_chars[(int)_value]);
	}

	//Compare cells
	bool equals(cell c) const
	{
		return c.value() == _value;
	}
};
```

3.3.2 Board Class

The board class is a placeholder for the actual tic-tac-toe board. It holds 9 cells as vector, represented by ‘cell’ class. The board also holds the depth it is currently in, which player does it belongs to i.e. current player. It also points to its parent board from which it was originated. It also saves the last position a player put its turn. For example, if the last turn was in the middle position, it would save 4 as its _last_position value;

```
class board
{
private:
#if DEBUG|_DEBUG|!NDEBUG
       // Board Positions
       // 0 | 1 | 2
       //============
       // 3 | 4 | 5
       //============
       // 6 | 7 | 8
       //
       //For Debugging purpose
       string _board_values;
#endif
       Player _current_player;
       board* _parent = nullptr;
       vector<cell> _values;
       vector<board> _successors;
       int _xposs_win = 0;
       int _oposs_win = 0;
       int _depth = 0;
       int _last_position = -1;

public:
       board();
       board(int depth);
       board(int depth, Player p);
       board(int depth, Player p, vector<cell> values);

       //Accessors
       vector<cell>& get_board();
       vector<board>& successors();
       int x_poss_win() { return _xposs_win; }
       int o_poss_win() { return _oposs_win; };

       //Sets and gets the depth of the board
       int depth() const { return _depth; }
       void depth(int depth) { _depth = depth; }
#if DEBUG || _DEBUG || !NDEBUG
       string tag() { test(); return _board_values; }
#else
       string tag() { return ""; }
#endif
       //Sets and gets the Parent
       board* parent() const { return _parent; }
       void parent(board* parent) { _parent = parent; }
```

```
	//Set player (MIN or MAX)
	void player(Player p) { _current_player = p; }
	Player player() const { return _current_player; }

	//Gets the last changed position
	int get_last_position() const { return _last_position; }

	cell get_cell(int position);
	vector<cell> values() const { return _values; }
	vector<int> get_available();

	//Prints the Board in a formatted fashion
	void print_board();

	//Checks each row combination to see if there is a possible win or not
	bool find_win();

	//Change cells
	void change_cell(int position, cell_value value);

	//Finds the product of each row
	int find_product(int c1, int c2, int c3) const;

	//Finds if the game is won by either player and calculates the score
	bool is_terminal(int product);

	//Finds if the game is won or tie by either player for this board
	bool is_game_over() const;
};
```

The board is responsible for several functionalities. One of them is, generating possible successors when called on. It also can check whether there is any winning chance for 'X' or 'O' or if the current board was a won-board, i.e. game-over board. This checking is performed by assigning specific numbers to 'X', 'O' and 'BLANK', which as discussed in the class. Stored numbers were 2 (indicating blank), 3 (indicating X), or 5 (indicating O). If the product of a row, diagonal or column is 18 (3 x 3 x 2), then 'X' has a chance to win. If the product is 50(5 x 5 x 2), then 'O' can win. If a winning combination is found, then the blank cell is determined and return the position of that cell.

```
bool board::find_win() {
	_xposs_win = 0;
	_oposs_win = 0;
	return is_terminal(find_product(0, 1, 2)) ||
		is_terminal(find_product(3, 4, 5)) ||
		is_terminal(find_product(6, 7, 8)) ||
		is_terminal(find_product(0, 4, 8)) ||
		is_terminal(find_product(2, 4, 6)) ||
		is_terminal(find_product(0, 3, 6)) ||
		is_terminal(find_product(1, 4, 7)) ||
		is_terminal(find_product(2, 5, 8));
}
```

```
int board::find_product(int c1, int c2, int c3) const {
       return (int)_values[c1].value() * (int)_values[c2].value() *
              (int)_values[c3].value();
}

bool board::is_terminal(int product) {
       switch (product) {
              //If x is in row with two blanks
       case 12:
              _xposs_win++;
              break;
              //If o is in row with two blanks
       case 20:
              _oposs_win++;
              break;
              //If two x's in row with blanks
       case 18:
              _xposs_win += 10;
              break;
              //If two o's in row with blanks
       case 50:
              _oposs_win += 10;
              break;
       case 27:
              //cout << "\nX wins!\n";
              return true;
       case 125:
              //cout << "\nO wins!\n";
              return true;
       }
       return false;
}

bool board::is_game_over() const
{
       int cell_count = 9;
       for (auto c : _values)
       {
              cell_count -= c.value() == cell_value::BLANK;
       }
       //indicates final move
       if (cell_count >= 9)
       {
              return true;
       }
       else if (_current_player == Player::X)
       {
              return find_product(0, 1, 2) == 27 ||
                     find_product(3, 4, 5) == 27 ||
                     find_product(6, 7, 8) == 27 ||
                     find_product(0, 4, 8) == 27 ||
                     find_product(2, 4, 6) == 27 ||
                     find_product(0, 3, 6) == 27 ||
                     find_product(1, 4, 7) == 27 ||
                     find_product(2, 5, 8) == 27;
       }
       else
       {
              return find_product(0, 1, 2) == 125 ||
                     find_product(3, 4, 5) == 125 ||
                     find_product(6, 7, 8) == 125 ||
                     find_product(0, 4, 8) == 125 ||
                     find_product(2, 4, 6) == 125 ||
                     find_product(0, 3, 6) == 125 ||
                     find_product(1, 4, 7) == 125 ||
                     find_product(2, 5, 8) == 125;
       }
}
```

3.3.3 Ply Class

The 'ply' class can be said to be the heart of the program. This class manages all the discrete classes and connects all the classes together to perform as a single mechanism. It also determines which algorithm to call, to what depth and which evaluation functions to call. In a way this class acts like a manager.

```
class ply
{
private:
       board _board;
       cell_value _curr_turn;
       //char node_id;
       int _cutoff_depth;
       int _eval_id_x;
       int _eval_id_o;
       int _node_count_x;
       int _node_count_o;
       bool _x_ab;
       bool _o_ab;

       struct search_result max_value_search(board* state, Player player,
                            int cutoff_depth, int alpha, int beta);
       struct search_result min_value_search(board* state, Player player,
                            int cutoff_depth, int alpha, int beta);
       struct search_result minimax_ab(board* pos, int depth, Player player,
                            int use_thres, int pass_thres);
public:
       struct search_result alpha_beta_search(board* root_state,
                     int cutoff_depth, Player firstPlayer, int alpha, int beta);
       board& get_borad() { return _board; }
       //Clone the current board, and generate ply of count 'depth',
       //not saving the boards
       shared_ptr<board> generate_ply_depth(int depth);
       search_result check_best_move(board* root);

       //Start the game by specifying the evaluation method desired:
       //----1: default
       //----2: defense
       //----3: offense
```

```
        //----4: Defensive Attack
        //----5: spot
        int evaluate(board& b, Player);
        ply(int eval_id_x, int eval_id_o, int depth, bool x_ab, bool o_ab);
        //~ply();
        //Changes the value of specified cell, if not already filled
        bool make_move(int index);
        //Print current player
        const cell_value& get_current_player();

        //Accessor for the board's successors
        //vector<board> get_successors(board board, cell_value player);
        //Switch to the next player
        cell_value switch_player(cell_value player);
        //Print out the successors
        //void print_successors();

        //Get node count
        int get_node_count_x();
        int get_node_count_o();

        void increase_node_count();
};
```

3.3.4 Search Module

Search module includes the search algorithm function. This module was separated for modular design and readability. This module contains the MinimaxAB and Alpha-Beta search functions and all of it's accessories.

```
struct search_result max_value_search(board* state, Player player,
                              int cutoff_depth, int alpha, int beta);

struct search_result min_value_search(board* state, Player player,
                              int cutoff_depth, int alpha, int beta);

struct search_result minimax_ab(board* pos, int depth, Player player,
                              int use_thres, int pass_thres);

bool cutoff_reached(const board* node, int cutoff_depth);

struct search_result alpha_beta_search(board* root_state, int cutoff_depth,
                              Player firstPlayer, int alpha, int beta);
```

3.3.5 Evaluation Module

Evaluation module contains the evaluation functions. As with the search module, it was also separated to promote modular and refactorable design. It helped diffuse some cluttering in ply class. This is not a class, this is just a header file which includes the evaluation functions only, completely modularized. Also all of the functions related to evaluation heuristics and accessories are in this module.

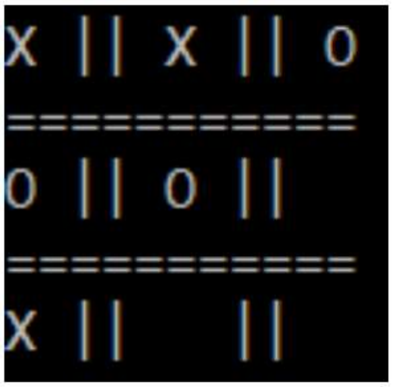
```
//Count Defense point
static int def_attk_eval_score(int product, Player p);
//Use defense first always at the same time look for an attack
static int defensive_attack_eval(board& position, cell_value player);
//evaluation that uses the magic square to take the current player's score
versus the opponent's
static int default_eval(board& position, cell_value player);
//evaluation that has the current player only think about defense (blocking the
other players moves)
static int eval_defense(board& position, cell_value player);
//evaluation that has the current player only think about offense (not caring
what the other player is doing)
static int eval_offense(board& position, cell_value player);
//Count up desired spots
static int desired_spots(board current, cell_value _value);
//Choose desired spots
static int eval_spot(board& position, cell_value player);
```

4 EVALUATION METHODS

4.1 DEFAULT EVALUATION METHOD

The default evaluation is the method that is shown in the Russell and Norvig textbook that was also demonstrated by the instructor in the class. In this evaluation, it is first checked to see the current players turn. The board is then checked to see if the current board is a win for either X or O. If so, the game is completed. The system checks to see how many different possible wins they would have with a move versus their opponent on the same board.

```
//evaluation that uses the magic square to take the current player's score
versus the opponent's
static int default_eval(board& position, cell_value player) {

       if (position.find_win()) {
              if (is_x(player)) {
                     return 27;
              }
              else if (is_o(player)) {
                     return 125;
              }
       }

       if (is_x(player)) {
              return position.x_poss_win() - position.o_poss_win();
       }
       else if (is_o(player)) {
              return position.o_poss_win() - position.x_poss_win();
       }
       return -1;
}
```



```
x || x || o
===========
o || o ||
===========
x ||    ||
```

For example, if the board was set up like above, X would try to make O's possibility of winning lower, as O is in a winning combination. In this case, X would need to go in the middle row on the right to minimize the number of possible wins for O. This is done with every successor of the current board and then compared to the other boards. The successor with the largest score is then returned as the best choice for the user to move to.

4.2 DEFENSIVE EVALUATION METHOD

Blocking the opponent is the only of defensive evaluation. The player will just worry about blocking the other user's play. The more that the user blocks the other player, the "lower" the board's score. Just like in the default evaluation, it is first checked to see the current players turn. The board is then checked to see if the current board is a win for either x or o. If so, the game is completed. The system checks to see how many different possible wins opponent would have on a specific board.

```
//evaluation that has the current player only think about defense (blocking the
other players moves)
static int eval_defense(board& position, cell_value player) {
       if (position.find_win()) {
              if (is_x(player)) {
                     return 27;
              }
              else if (is_o(player)) {
                     return 125;
              }
       }
       if (is_x(player)) {
              return position.o_poss_win();
       }
       else if (is_o(player)) {
              return position.x_poss_win();
       }
       return -1;
}
```

Every successor of the current board is checked and evaluated, the board with the smallest score is the ideal board in this evaluation. The lower score represents the lower number of possible wins for the opponent of the current player.

4.3 OFFENSIVE EVALUATION METHOD

The offensive evaluation is the exact opposite of the defensive evaluation method. The current player only worries about winning and not worrying about whether the opponent has an upper hand or not. In this evaluation, the board checked to see if the current board is a win for current player. If so, the game is completed. The system checks to see how many different possible wins they would have with a move completely disregarding the other player's positions.

```
//evaluation that has the current player only think about offense (not caring
what the other player is doing)
static int eval_offense(board& position, cell_value player) {
       if (position.find_win()) {
              if (is_x(player)) {
                     return 27;
              }
              else if (is_o(player)) {
                     return 125;
              }
       }
       if (is_x(player)) {
              return position.x_poss_win();
       }
       else if (is_o(player)) {
              return position.o_poss_win();
       }
       return -1;
}
```

Every successor of the current board is checked and evaluated, the board with the highest score is the ideal board in this evaluation. The higher score represents the higher number of possible wins for the current player.

4.4 SPOT EVALUATION METHOD

This evaluation method favors some positions beforehand, and plays those positions if possible. The spot evaluation method is the only evaluation method to not look at the possible wins of either player. The desired positions have score, using which the current player chooses which position to play. Shown below is the scores for each cell in a tic-tac-toe board.

5	1	5
1	10	1
5	1	5

The center of the board is the coveted spot, giving the user 10 points if they occupy it. The corner positions are next highest in points, a total of 5 per corner. The non-corner or noncentral position is just 1 point for the current player. This is evaluated in the desired_spots() method.

```
//Count up desired spots
static int desired_spots(board current, cell_value _value) {
       int score = 0;
       for (int i = 0; i < 9; i++) {
              if (current.get_cell(i).value() == _value) {
                     if (i == 4) {
                            score += 10;
                     }
                     else if (i == 0 || i == 2 || i == 6 || i == 8)
                     {
                            score += 5;
                     }
                     else {
                            score += 1;
                     }
              }
       }
       return score;
}

static int eval_spot(board& position, cell_value player) {
       if (position.find_win()) {
              if (is_x(player)) {
                     return 27;
              }
              else if (is_o(player)) {
                     return 125;
              }
       }

       return desired_spots(position, player);
}
```

4.5 DEFENSIVE ATTACK EVALUATION METHOD

Like default evaluation method each of the winning position is checked. But based on current player a positive or negative mark is assigned. All the marks for 8 possible combinations are then summed up. Also, there is extra marks for corner and center pieces like spot evaluation. For MIN cases the values are inverted. This is the most optimized version of the evaluation.

```
static int def_attk_eval_score(int product, Player p)
{
        switch (product) {
                //If x is in row with two blanks
        case 12:
                return is_x(p) ? 10 : -10;
                //If o is in row with two blanks
        case 20:
                return is_o(p) ? 10 : -10;
                break;
                //If two x's in row with blanks
        case 18:
                return is_x(p) ? 120 : -100;
                break;
                //If two o's in row with blanks
        case 50:
                return is_o(p) ? 120 : -100;
                break;
        case 27:
                return is_x(p) ? 1000 : -1000;
        case 125:
                return is_o(p) ? 1000 : -1000;
        }

        return 0;
}

static int defensive_attack_eval(board& position, cell_value player)
{
        int total_score = 0;
        total_score += def_attk_eval_score(position.find_product(0, 1, 2), player);
        total_score += def_attk_eval_score(position.find_product(3, 4, 5), player);
        total_score += def_attk_eval_score(position.find_product(6, 7, 8), player);
        total_score += def_attk_eval_score(position.find_product(0, 4, 8), player);
        total_score += def_attk_eval_score(position.find_product(2, 4, 6), player);
        total_score += def_attk_eval_score(position.find_product(0, 3, 6), player);
        total_score += def_attk_eval_score(position.find_product(1, 4, 7), player);
        total_score += def_attk_eval_score(position.find_product(2, 5, 8), player);

        //Extra mark for corner and middle position
        if (is_x(player))
        {
                total_score += (position.get_cell(0).value() == cell_value::X) ? 2 : 0 +
                        (position.get_cell(2).value() == cell_value::X) ? 2 : 0 +
                        (position.get_cell(4).value() == cell_value::X) ? 5 : 0 +
                        (position.get_cell(6).value() == cell_value::X) ? 2 : 0 +
                        (position.get_cell(8).value() == cell_value::X) ? 2 : 0;
        }
        else
        {
                total_score += (position.get_cell(0).value() == cell_value::O) ? 2 : 0 +
                        (position.get_cell(2).value() == cell_value::O) ? 2 : 0 +
                        (position.get_cell(4).value() == cell_value::O) ? 5 : 0 +
                        (position.get_cell(6).value() == cell_value::O) ? 2 : 0 +
                        (position.get_cell(8).value() == cell_value::O) ? 2 : 0;
        }

        //For MIN case the values have to be reverted
        if (is_o(player))
                total_score = -total_score;

        return total_score;
}
```

4.6 THE MAIN FUNCTION

The main function resides in main.cpp file. This is the starter function. This function is responsible for taking all required inputs from the user. The search algorithm, evaluation function choice and the depth value input are all gathered here.

```
int eval_choice(string name) {
       int choice;
       cout << "Enter evaluation desired for " + name + ": \n";
       cout << "----1. Default Evaluation\n";
       cout << "----2. Defense Evaluation\n";
       cout << "----3. Offense Evaluation\n";
       cout << "----4. Defensive-Attack Evaluation\n";
       cout << "----5. Spot Evaluation\n";
       cout << "Enter choice: ";
       cin >> choice;
       return choice;
}

bool search_choice(string name) {
       int choice;
       cout << "Enter algorithm desired for " + name + ": \n";
       cout << "----1. Alpha Beta Searching\n";
       cout << "----2. MinMaxAB Searching\n";
       cout << "Enter choice: ";
       cin >> choice;
       return choice == 1;
}

int depth_choice() {
       int depth;
       cout << "Depth of searching: ";
       cin >> depth;
       return depth;
}
```

First player one is then asked what type of evaluation should it use. Then it is asked for type of search algorithm the player would like to use. After that Player 2 is asked for the desired evaluation method and the search algorithm. The desired depth for the game is asked to the user. The depth will determine how far down the graph search algorithms will expand before applying evaluation.

4.7 Comparison of Algorithms with fixed Evaluation Functions

The following will show the test results and comparison charts for each of the evaluation function and search algorithms.

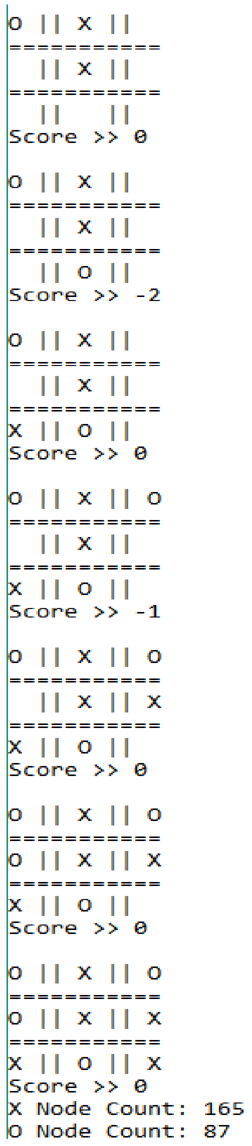
4.7.1 Depth 2

4.7.1.1 Comparison of MiniMaxAB and Alpha-Beta Searching at Depth 2

4.7.1.1.1 Default Evaluation

X	MiniMaxAb Searching	Default Evaluation Function
O	AlphaBeta Searching	

Output:



```
o || x ||
===========
  || x ||
===========
  ||   ||
Score >> 0

o || x ||
===========
  || x ||
===========
  || o ||
Score >> -2

o || x ||
===========
  || x ||
===========
x || o ||
Score >> 0

o || x || o
===========
  || x ||
===========
x || o ||
Score >> -1

o || x || o
===========
  || x || x
===========
x || o ||
Score >> 0

o || x || o
===========
o || x || x
===========
x || o ||
Score >> 0

o || x || o
===========
o || x || x
===========
x || o || x
Score >> 0
X Node Count: 165
O Node Count: 87
```

4.7.1.1.2 Defensive Evaluation

X	MiniMaxAb Searching	Defensive Evaluation Function
O	AlphaBeta Searching	

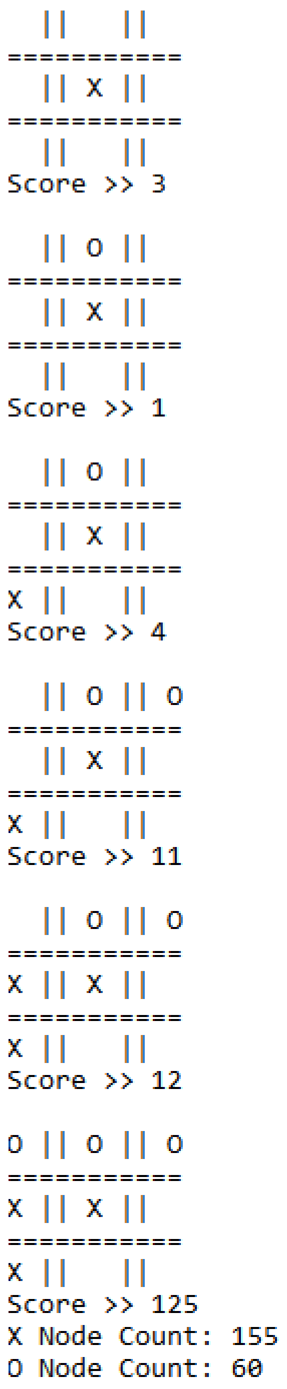
Output:

```
X ||   ||
===========
  ||   ||
===========
  ||   ||
Score >> 1

X ||   ||
===========
  || O ||
===========
  ||   ||
Score >> 12

X ||   ||
===========
  || O ||
===========
  ||   || X
Score >> 12

X ||   || O
===========
  || O ||
===========
  ||   || X
Score >> 20

X || X || O
===========
  || O ||
===========
  ||   || X
Score >> 11

X || X || O
===========
O || O ||
===========
  ||   || X
Score >> 10

X || X || O
===========
O || O ||
===========
  || X || X
Score >> 27

X || X || O
===========
O || O || O
===========
  || X || X
Score >> 125
X Node Count: 164
O Node Count: 68
```

4.7.1.1.3 Offensive Evaluation

X	MiniMaxAb Searching	Offensive Evaluation Function
O	AlphaBeta Searching	

Output

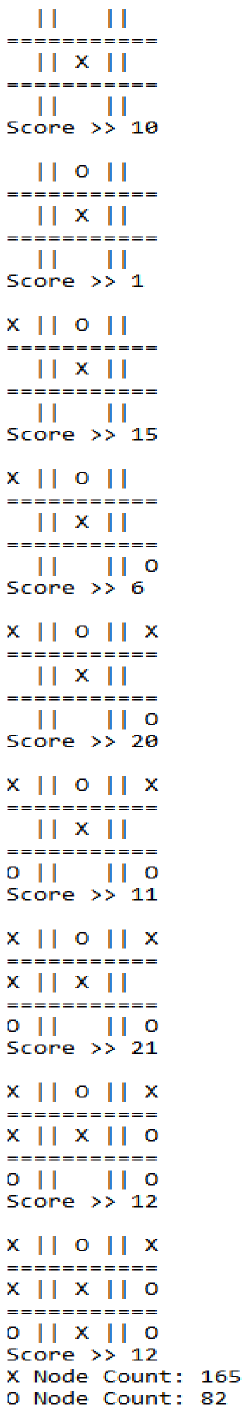


```
  ||   ||
===========
  || X ||
===========
  ||   ||
Score >> 3

  || O ||
===========
  || X ||
===========
  ||   ||
Score >> 1

  || O ||
===========
  || X ||
===========
X ||   ||
Score >> 4

  || O || O
===========
  || X ||
===========
X ||   ||
Score >> 11

  || O || O
===========
X || X ||
===========
X ||   ||
Score >> 12

O || O || O
===========
X || X ||
===========
X ||   ||
Score >> 125
X Node Count: 155
O Node Count: 60
```

4.7.1.1.4 Spot Evaluation Function

X	MiniMaxAb Searching	Spot Evaluation Function
O	AlphaBeta Searching	

Output

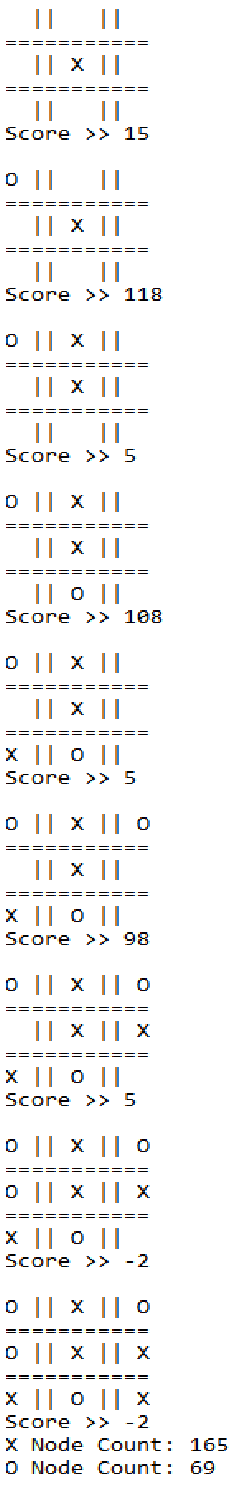


```
  ||   ||
===========
  || x ||
===========
  ||   ||
Score >> 10

  || o ||
===========
  || x ||
===========
  ||   ||
Score >> 1

x || o ||
===========
  || x ||
===========
  ||   ||
Score >> 15

x || o ||
===========
  || x ||
===========
  ||   || o
Score >> 6

x || o || x
===========
  || x ||
===========
  ||   || o
Score >> 20

x || o || x
===========
  || x ||
===========
o ||   || o
Score >> 11

x || o || x
===========
x || x ||
===========
o ||   || o
Score >> 21

x || o || x
===========
x || x || o
===========
o ||   || o
Score >> 12

x || o || x
===========
x || x || o
===========
o || x || o
Score >> 12
X Node Count: 165
O Node Count: 82
```

4.7.1.1.5 Defensive Attack Evaluation Function

X	MiniMaxAb Searching	Defensive Attack Evaluation Function
O	AlphaBeta Searching	

Output



```
   ||   ||
===========
   || X ||
===========
   ||   ||
Score >> 15

O  ||   ||
===========
   || X ||
===========
   ||   ||
Score >> 118

O  || X ||
===========
   || X ||
===========
   ||   ||
Score >> 5

O  || X ||
===========
   || X ||
===========
   || O ||
Score >> 108

O  || X ||
===========
   || X ||
===========
X  || O ||
Score >> 5

O  || X || O
===========
   || X ||
===========
X  || O ||
Score >> 98

O  || X || O
===========
   || X || X
===========
X  || O ||
Score >> 5

O  || X || O
===========
O  || X || X
===========
X  || O ||
Score >> -2

O  || X || O
===========
O  || X || X
===========
X  || O || X
Score >> -2
X Node Count: 165
O Node Count: 69
```

4.7.1.2 Overview of Comparison and Analysis

Here it is apparent that MiniMaxAB expands more nodes than Alpha-Beta. It is conclusive that MinimaxAB is less performant although the first player X was assigned to use MiniMaxAB, the first player always must expand more nodes than its opponent O.

	X - MiniMaxAB (# of Nodes Visited)	O - Alpha-Beta (# of Nodes Visited)	**Game Status**
Default	165	87	Tie
Defensive	164	68	O Won
Offensive	155	60	O Won
Spot	165	82	Tie
Defensive Attack*	**165**	**69**	**Tie**

About the evaluation function, it is apparent that the fifth one, Defensive Attack is performing better, as it ties the match, so each player is getting same output as each other from the evaluation, also the node visited count on O's end are smallest.

4.7.2 Depth 4

4.7.2.1 Comparison of MiniMaxAB and Alpha-Beta Searching at Depth 4

4.7.2.1.1 Default Evaluation

X	MiniMaxAb Searching	Default Evaluation Function
O	AlphaBeta Searching	

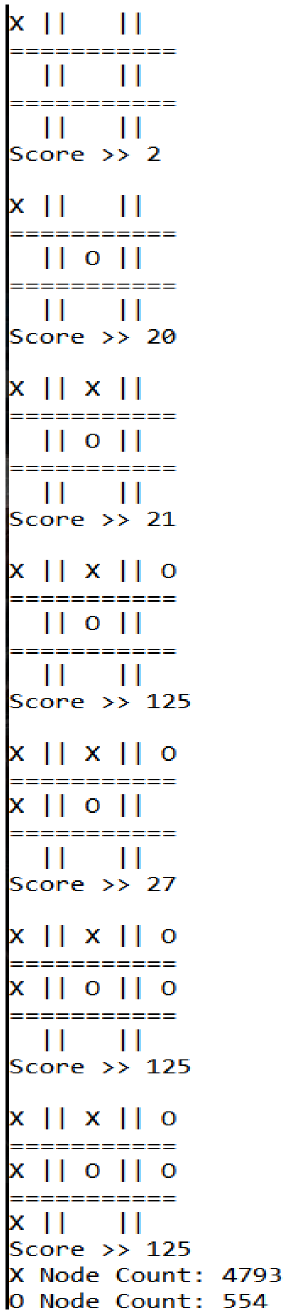
Output

```
  ||   ||
===========
  || X ||
===========
  ||   ||
Score >> 0

  || O ||
===========
  || X ||
===========
  ||   ||
Score >> -2

  || O ||
===========
  || X ||
===========
X ||   ||
Score >> 2

O || O ||
===========
  || X ||
===========
X ||   ||
Score >> 125

O || O || X
===========
  || X ||
===========
X ||   ||
Score >> 125
X Node Count: 4677
O Node Count: 493
```

4.7.2.1.2 Defensive Evaluation

X	MiniMaxAb Searching	Defensive Evaluation Function
O	AlphaBeta Searching	

Output



```
X ||   ||
===========
  ||   ||
===========
  ||   ||
Score >> 2

X ||   ||
===========
  || O ||
===========
  ||   ||
Score >> 20

X || X ||
===========
  || O ||
===========
  ||   ||
Score >> 21

X || X || O
===========
  || O ||
===========
  ||   ||
Score >> 125

X || X || O
===========
X || O ||
===========
  ||   ||
Score >> 27

X || X || O
===========
X || O || O
===========
  ||   ||
Score >> 125

X || X || O
===========
X || O || O
===========
X ||   ||
Score >> 125
X Node Count: 4793
O Node Count: 554
```

4.7.2.1.3 Offensive Evaluation

X	MiniMaxAb Searching	Offensive Evaluation Function
O	AlphaBeta Searching	

Output

```
   ||   ||
===========
   || x ||
===========
   ||   ||
Score >> 3

o  ||   ||
===========
   || x ||
===========
   ||   ||
Score >> 11

o  ||   || x
===========
   || x ||
===========
   ||   ||
Score >> 11

o  || o || x
===========
   || x ||
===========
   ||   ||
Score >> 125

o  || o || x
===========
x  || x ||
===========
   ||   ||
Score >> 125

o  || o || x
===========
x  || x || o
===========
   ||   ||
Score >> 125

o  || o || x
===========
x  || x || o
===========
x  ||   ||
Score >> 125
X Node Count: 4716
O Node Count: 626
```

4.7.2.1.4 Spot evaluation

X	MiniMaxAb Searching	Spot Evaluation Function
O	AlphaBeta Searching	

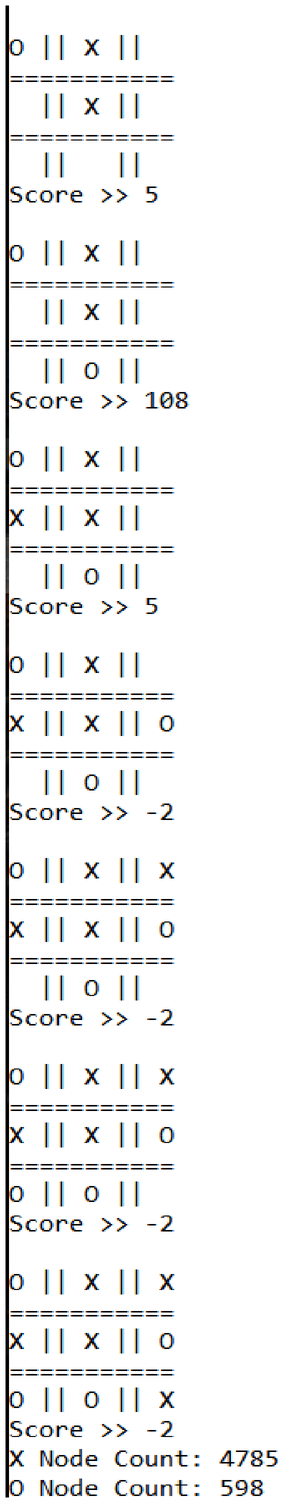
Output

```
  ||   ||
===========
  || x ||
===========
  ||   ||
Score >> 15

  || o ||
===========
  || x ||
===========
  ||   ||
Score >> 6

x || o ||
===========
  || x ||
===========
  ||   ||
Score >> 20

x || o || o
===========
  || x ||
===========
  ||   ||
Score >> 125

x || o || o
===========
x || x ||
===========
  ||   ||
Score >> 125

x || o || o
===========
x || x || o
===========
  ||   ||
Score >> 125

x || o || o
===========
x || x || o
===========
x ||   ||
Score >> 125
X Node Count: 4711
O Node Count: 487
```

4.7.2.1.5 Defensive Attack Evaluation

X	MiniMaxAb Searching	Defensive Attack Evaluation Function
O	AlphaBeta Searching	

Output



```
o || x ||
===========
  || x ||
===========
  ||   ||
Score >> 5

o || x ||
===========
  || x ||
===========
  || o ||
Score >> 108

o || x ||
===========
x || x ||
===========
  || o ||
Score >> 5

o || x ||
===========
x || x || o
===========
  || o ||
Score >> -2

o || x || x
===========
x || x || o
===========
  || o ||
Score >> -2

o || x || x
===========
x || x || o
===========
o || o ||
Score >> -2

o || x || x
===========
x || x || o
===========
o || o || x
Score >> -2
X Node Count: 4785
O Node Count: 598
```

4.7.2.2 Overview of Comparison and Analysis

Judging the data of depth 4, we can determine MiniMaxAB expands more nodes than Alpha-Beta.

	X - MiniMaxAB (# of Nodes Visited)	O - Alpha-Beta (# of Nodes Visited)	**Game Status**
Default	4677	493	X
Defensive	4793	554	X
Offensive	4716	626	X
Spot	4711	487	X
Defensive Attack*	**4785**	**598**	**Tie**

Also, all the evaluation functions didn't perform that good when dealing with depth 4. They let one player win all the games. From the previous conclusion and from this results it is apparent that the fifth one, Defensive Attack is performing better, as it ties the match, while expanding reasonably lower nodes.

4.7.3 Overall Comparison

This is the overall comparative picture where both depths are present.

	Minimax AB (#Nodes)		AlphaBeta (#Nodes)		Game Status	
	Depth=2	Depth=4	Depth=2	Depth=4	Depth=2	Depth=4
Default	165	4677	87	493	Tie	X
Defensive	164	4793	68	554	O Won	X
Offensive	155	4716	60	626	O Won	X
Spot	165	4711	82	487	Tie	X
Defensive Attack*	**165**	**4785**	**69**	**598**	**Tie**	**Tie**

4.8 COMPARISON OF EVALUATION FUNCTIONS WITH FIXED ALGORITHM

The output has been executed but not presented in this report. Just the analysis results are presented

4.8.1 Depth 2 and MinimaxAB

# of Nodes Expanded	# of nodes Expanded
Spot Evaluation vs **Defensive Evaluation**	
165	116
Defensive Evaluation vs **Offensive Evaluation**	
165	120
Spot Evaluation vs **Offensive Evaluation**	
165	119
Defensive Vs Defensive Attack Evaluation	
164	115
Offensive vs Defensive-Attack Evaluation	
155	113
Spot vs Defensive - Attack Evaluation	
165	116

From the output, it is apparent that the spot evaluation always expands the most nodes, but at the same time defensive attack always uses less nodes than all of them.

4.8.2 Depth 4 and MinimaxAB

# of Nodes Expanded	# of nodes Expanded
Spot Evaluation vs **Defensive Evaluation**	
4729	2525
Defensive Evaluation vs **Offensive Evaluation**	
4751	2524
Spot Evaluation vs **Offensive Evaluation**	
4729	2525
Defensive Vs Defensive Attack Evaluation	
4785	2503
Offensive vs Defensive-Attack Evaluation	
4729	2525
Spot vs Defensive - Attack Evaluation	
4729	2525

From the output, it is apparent that our previous hypothesis were correct that defensive attack always uses less nodes than all of them.

4.8.3 Depth 2 and AlphaBeta

# of Nodes Expanded	# of nodes Expanded
Spot Evaluation vs **Defensive Evaluation**	
80	62
Defensive Evaluation vs **Offensive Evaluation**	
165	120
Spot Evaluation vs **Offensive Evaluation**	
72	81
Defensive Vs Defensive Attack Evaluation	
96	71
Offensive vs Defensive-Attack Evaluation	
79	60
Spot vs Defensive - Attack Evaluation	
72	60

When defensive and offensive evaluations went against each other, they had to expand a lot of nodes to see find the best path. Any time an evaluation went against spot evaluation, they expanded a low number of nodes. Spot evaluation ended up being better than offensive, but worse than defensive. But all of them are trumped by defensive-attack evaluation, in terms of node number and winning chance.

4.8.4 Depth 4 and AlphaBeta

# of Nodes Expanded	# of nodes Expanded
Spot Evaluation vs **Defensive Evaluation**	
750	375
Defensive Evaluation vs **Offensive Evaluation**	
1125	534
Spot Evaluation vs **Offensive Evaluation**	
750	626
Defensive Vs Defensive-Attack Evaluation	
1001	563
Offensive vs Defensive-Attack Evaluation	
809	589
Spot vs Defensive – Attack Evaluation	
789	589

Same result is for depth 4 as depth 2 in terms of defensive-attack evaluation.

5 Conclusion

This project was one of the toughest and exciting project. This project, while providing a broad view about the implementation of AI techniques with heuristics, we also did get a deeper understanding of how to implement and optimize our own heuristics for different types of application. We found a new paradigm to a mere simple game like tic-tac-toe.

We could find the best method after several different comparisons to see what is the best evaluation method combined with the best searching algorithm is. We conducted several analysis with different permutations of search algorithms and evaluation functions. In my opinion, based on that empirical evidence, alpha-beta in combination with defensive-attack evaluation function would be best for most of the scenario of tic-tac-toe.

References

1. Russell S.J., Norvig P. - Artificial Intelligence - A Modern Approach 3rd edition (Prentice Hall) – 2010
2. Elaine Rich, Kevin Knight, Artificial Intelligence 3rd Edition, (Tata McGrew Hill) - 2010

APPENDIX

SOURCE CODE

Board.h

```cpp
#pragma once

#include "stdafx.h"

using namespace std;
enum class cell_value
{
	BLANK = 2,
	X = 3, //MAX
	O = 5 //MIN
};

using Player = cell_value;

//SWAP PLAYER MACRO
#define SWAP_PLAYER(_X_) (_X_ == cell_value::X)?cell_value::O:cell_value::X
//Find if player is x
bool is_x(cell_value player);
//Find if player is o
bool is_o(cell_value player);

class cell
{
private:
	cell_value _value = cell_value::BLANK;
	int _position = -1;
	const char* cell_value_chars[6] = { "",""," ","X","","O" };
public:
	cell() {}
	//The default value of a cell is blank
	cell(int position, cell_value v) :_value(v), _position(position) {}

	//Accessors of instance variables
	const cell_value value() const
	{
		return _value;
	}
	void value(cell_value cv)
	{
		_value = cv;
	}

	//Returns if the cell is currently filled or blank
	bool is_blank() const
	{
		return _value == cell_value::BLANK;
	}

	//Prints out the value of the cell
	string print_value() {
		return string(cell_value_chars[(int)_value]);
	}

	//Compare cells
	bool equals(cell c) const
	{
		return c.value() == _value;
	}
};

class board
{
private:
#if DEBUG|_DEBUG|!NDEBUG
	// 0 | 1 | 2
	//===========
	// 3 | 4 | 5
	//===========
	// 6 | 7 | 8
	//
	//For Debugging purpose
	string _board_values;
#endif
	Player _current_player;
	board* _parent = nullptr;
	vector<cell> _values;
	vector<board> _successors;
	int _xposs_win = 0;
	int _oposs_win = 0;
	int _depth = 0;
	int _last_position = -1;

public:
#if _DEBUG || DEBUG || !NDEBUG
	string test();
	string test2();
#endif
	board();
	board(int depth);
	board(int depth, Player p);
	board(int depth, Player p, vector<cell> values);

	//Accessors
	vector<cell>& get_board();
	vector<board>& successors();
	int x_poss_win() { return _xposs_win; }
	int o_poss_win() { return _oposs_win; };

	//Sets and gets the depth of the board
	int depth() const { return _depth; }
	void depth(int depth) { _depth = depth; }
#if DEBUG || _DEBUG || !NDEBUG
```

```
        string tag() { test(); return _board_values; }
#else
        string tag() { return ""; }
#endif
        //Sets and gets the Parent
        board* parent() const { return _parent; }
        void parent(board* parent) { _parent = parent; }

        //Set player (MIN or MAX)
        void player(Player p) { _current_player = p; }
        Player player() const { return _current_player; }

        //Gets the last changed position
        int get_last_position() const { return _last_position; }

        cell get_cell(int position);
        vector<cell> values() const { return _values; }
        vector<int> get_available();

        //Prints the Board in a formatted fashion
        void print_board();

        //Checks each row combination to see if there is a possible win or not
        bool find_win();

        //Change cells
        void change_cell(int position, cell_value value);

        //Finds the product of each row
        int find_product(int c1, int c2, int c3) const;

        //Finds if the game is won by either player and calculates the score
        bool is_terminal(int product);

        //Finds if the game is won or tie by either player for this board
        bool is_game_over() const;
};
```

Board.cpp

```
#include "board.h"

bool is_x(cell_value player) {
        return player == cell_value::X;
}

//Find if player is o
bool is_o(cell_value player) {
        return (player == cell_value::O);
}

/*       Board Methods       */
board::board() :board(0) {
}
```

```
board::board(int depth) : _depth(depth), _values(9) {}

board::board(int depth, Player p) : _current_player(p), _depth(depth),
_values(9) {}

board::board(int depth, Player p, vector<cell> values) : _current_player(p),
_values(values), _depth(depth) {}

vector<cell>& board::get_board() {
	return _values;
}

vector<int> board::get_available() {
	vector<int> currently_available;
	for (int i = 0; i < 9; i++) {
		if (get_cell(i).value() == cell_value::BLANK) {
			currently_available.push_back(i);
		}
	}
	return currently_available;
}

cell board::get_cell(int index) {
	return _values[index];
}

#if _DEBUG || DEBUG || !NDEBUG
string board::test()
{
	string ss;
	for (int i = 0; i < 9; i++) {
		if (i % 3 == 0 && i != 0) {
			ss += "*";
		}
		switch (this->get_cell(i).value())
		{
		case cell_value::BLANK:
			ss += " |";
			break;
		case cell_value::X:
			ss += "X|";
			break;
		case cell_value::O:
			ss += "O|";
			break;
		}
	}
	_board_values = ss;
	return ss;
}
string board::test2()
{
	string ss;
	for (int i = 0; i < 9; i++) {
```

```
                switch (this->get_cell(i).value())
                {
                case cell_value::BLANK:
                        ss += " ";
                        break;
                case cell_value::X:
                        ss += "X";
                        break;
                case cell_value::O:
                        ss += "O";
                        break;
                }
        }
        return ss;
}
#endif

void board::change_cell(int position, cell_value value) {
        _values[position].value(value);
        _last_position = position;
#if _DEBUG || DEBUG || !NDEBUG
        //TEST, for debug PURPOSE
        _board_values = test();
#endif
}

vector<board>& board::successors()
{
        if (_successors.size() > 0 || this->is_game_over())
        {
                return _successors;
        }

        Player swapped_player = SWAP_PLAYER(_current_player);
        for (int i = 0; i < 9; i++)
        {
                board b(_depth + 1, swapped_player, _values);
                if (!b.get_cell(i).is_blank()) continue;
                b.change_cell(i, swapped_player);
                b.parent(this);
                _successors.push_back(b);
        }

        return _successors;
}

void board::print_board() {
        cout << endl;
        for (int i = 0; i < 9; i++) {
                if (i % 3 == 0 && i != 0) {
                        cout << "\n===========\n";
                }
                cout << get_cell(i).print_value();
                if (i == 0 || i == 1 || i == 3 || i == 4 || i == 6 || i == 7) {
                        cout << " || ";
                }
```

```
	}
	cout << endl;
}

bool board::find_win() {
	_xposs_win = 0;
	_oposs_win = 0;
	return is_terminal(find_product(0, 1, 2)) ||
		is_terminal(find_product(3, 4, 5)) ||
		is_terminal(find_product(6, 7, 8)) ||
		is_terminal(find_product(0, 4, 8)) ||
		is_terminal(find_product(2, 4, 6)) ||
		is_terminal(find_product(0, 3, 6)) ||
		is_terminal(find_product(1, 4, 7)) ||
		is_terminal(find_product(2, 5, 8));
}



int board::find_product(int c1, int c2, int c3) const {
	return (int)_values[c1].value() * (int)_values[c2].value() *
(int)_values[c3].value();
}

bool board::is_terminal(int product) {
	switch (product) {
		//If x is in row with two blanks
	case 12:
		_xposs_win++;
		break;
		//If o is in row with two blanks
	case 20:
		_oposs_win++;
		break;
		//If two x's in row with blanks
	case 18:
		_xposs_win += 10;
		break;
		//If two o's in row with blanks
	case 50:
		_oposs_win += 10;
		break;
	case 27:
		//cout << "\nX wins!\n";
		return true;
	case 125:
		//cout << "\nO wins!\n";
		return true;
	}
	return false;
}

bool board::is_game_over() const
{
	int cell_count = 9;
	for (auto c : _values)
```

```
        {
                cell_count -= c.value() == cell_value::BLANK;
        }
        //indicates final move
        if (cell_count >= 9)
        {
                return true;
        }
        else if (_current_player == Player::X)
        {
                return  find_product(0, 1, 2) == 27 ||
                        find_product(3, 4, 5) == 27 ||
                        find_product(6, 7, 8) == 27 ||
                        find_product(0, 4, 8) == 27 ||
                        find_product(2, 4, 6) == 27 ||
                        find_product(0, 3, 6) == 27 ||
                        find_product(1, 4, 7) == 27 ||
                        find_product(2, 5, 8) == 27;
        }
        else
        {
                return  find_product(0, 1, 2) == 125 ||
                        find_product(3, 4, 5) == 125 ||
                        find_product(6, 7, 8) == 125 ||
                        find_product(0, 4, 8) == 125 ||
                        find_product(2, 4, 6) == 125 ||
                        find_product(0, 3, 6) == 125 ||
                        find_product(1, 4, 7) == 125 ||
                        find_product(2, 5, 8) == 125;
        }
}
```

Ply.h

```
#ifndef PLY_H
#define PLY_H
#include <cstdlib>
#include <limits>
#include "board.h"

#if !defined(INT_MIN)
#define INT_MIN std::numeric_limits<int>::min()
#endif
#if !defined(INT_MAX)
#define INT_MAX std::numeric_limits<int>::max()
#endif

#define POS_INF INT_MAX
#define NEG_INF INT_MIN

struct search_result {
        int value;
        int position;
        board* node = nullptr;
```

```
	search_result() {}
	search_result(int v, int p) : value(v), position(p) {}
};

class ply
{
private:
	board _board;
	cell_value _curr_turn;
	//char node_id;
	int _cutoff_depth;
	int _eval_id_x;
	int _eval_id_o;
	int _node_count_x;
	int _node_count_o;
	bool _x_ab;
	bool _o_ab;

	//board* _current_board;

	//search_result _min_max_search(board b, int depth, bool maximizing);
	//search_result _alpha_beta_search(board b, int alpha, int beta, int
depth, bool maximizing);
	struct search_result max_value_search(board* state, Player player, int
cutoff_depth, int alpha, int beta);
	struct search_result min_value_search(board* state, Player player, int
cutoff_depth, int alpha, int beta);
	struct search_result minimax_ab(board* pos, int depth, Player player, int
use_thres, int pass_thres);
public:
	struct search_result alpha_beta_search(board* root_state, int
cutoff_depth, Player firstPlayer, int alpha, int beta);
	board& get_borad() { return _board; }
	//Clone the current board, and generate ply of count 'depth', not saving
the boards
	shared_ptr<board> generate_ply_depth(int depth);
	search_result check_best_move(board* root);

	//Start the game by specifying the evaluation method desired:
	//----1: default
	//----2: defense
	//----3: offense
	//----4: Defensive Attack
	//----5: spot
	int evaluate(board& b, Player);
	ply(int eval_id_x, int eval_id_o, int depth, bool x_ab, bool o_ab);
	//~ply();
	//Changes the value of specified cell, if not already filled
	bool make_move(int index);
	//Print current player
	const cell_value& get_current_player();

	//Accessor for the board's successors
	//vector<board> get_successors(board board, cell_value player);
	//Switch to the next player
	cell_value switch_player(cell_value player);
	//Print out the successors
	//void print_successors();

	//Get node count
	int get_node_count_x();
	int get_node_count_o();

	void increase_node_count()
	{
		if (is_x(_curr_turn))
			_node_count_x++;
		else
			_node_count_o++;
	}
};

#endif // PLY_H
```

Ply.cpp

```
#include "ply.h"
#include "evaluations.h"

ply::ply(int eval_id_x, int eval_id_o, int depth, bool x_ab, bool o_ab)
{
	_cutoff_depth = depth;

	_curr_turn = cell_value::X;
	_board = board(0, SWAP_PLAYER(_curr_turn));
	//Kushal
	//_current_board = &_board;

	_node_count_x = 0;
	_node_count_o = 0;
	_eval_id_x = eval_id_x;
	_eval_id_o = eval_id_o;
	_x_ab = x_ab;
	_o_ab = o_ab;
}

const cell_value& ply::get_current_player() {
	return _curr_turn;
}

bool ply::make_move(int index) {
	//Checks to see if game is over
	if (_board.is_game_over()) {
		return false;
	}
	//Check to see if spot is filled
	else if (!_board.get_cell(index).is_blank()) {
		return false;
	}
```

```
        else {
                //cout << endl << (int)_curr_turn;

                _board.depth(_board.depth() + 1);
                _board.change_cell(index, _curr_turn);
                //_board.print_board();
                //print_score(_board, _curr_turn);
                switch_player(_curr_turn);
                _board.player(SWAP_PLAYER(_curr_turn));
                return true;
        }
}


shared_ptr<board> ply::generate_ply_depth(int depth)
{
        shared_ptr<board> gen_b(new board(_board));
        int targetDepth = _board.depth() + depth;
        queue<board*> all_boards;
        all_boards.push(gen_b.get());
        for (; all_boards.size() != 0; )
        {
                board* b = all_boards.front();
                all_boards.pop();
                vector<board>& succ = b->successors();
                if (b->depth() >= targetDepth - 1)
                        continue;
                for (board& s : b->successors())
                {
                        //if(s.depth() < targetDepth)
                        all_boards.push(&s);
                }
        }

        return gen_b;
}

struct search_result ply::check_best_move(board* root)
{
        int alpha_pass_thres = NEG_INF, beta_user_thres = POS_INF;
        int cutOffActualDepth = root->depth() + _cutoff_depth;

        if (is_x(_curr_turn)&_x_ab || is_o(_curr_turn)&_o_ab)
        {
                return alpha_beta_search(root, cutOffActualDepth, _curr_turn,
alpha_pass_thres, beta_user_thres);
        }
        return minimax_ab(root, cutOffActualDepth, _curr_turn, beta_user_thres,
alpha_pass_thres);
        /*if(is_x(_curr_turn))
                return _x_ab?alpha_beta_search(root, cutOffActualDepth,
_curr_turn, alpha_pass_thres, beta_user_thres):minimax_ab(root,
cutOffActualDepth, _curr_turn, alpha_pass_thres, beta_user_thres);
        else
```

```
              return _o_ab?alpha_beta_search(root, cutOffActualDepth,
_curr_turn, alpha_pass_thres, beta_user_thres):minimax_ab(root,
cutOffActualDepth, _curr_turn, alpha_pass_thres, beta_user_thres);*/
}

int ply::evaluate(board& curr, Player player)
{
       int score = 0;
       int eval_id = is_x(player) ? _eval_id_x : _eval_id_o;
       switch (eval_id) {
       case 1:
              score = default_eval(curr, player);
              break;
       case 2:
              score = eval_defense(curr, player);
              break;
       case 3:
              score = eval_offense(curr, player);
              break;
       case 4:
              score = defensive_attack_eval(curr, player);
              break;
       default:
              score = eval_spot(curr, player);
              break;
       }
       return score;
}

cell_value ply::switch_player(cell_value player) {
       _curr_turn = is_x(_curr_turn) ? cell_value::O : cell_value::X;
       return _curr_turn;
}

int ply::get_node_count_x() {
       return _node_count_x;
}

int ply::get_node_count_o() {
       return _node_count_o;
}
```

Search_Algo.cpp

```
#include "stdafx.h"
#include "ply.h"

bool cutoff_reached(const board* node, int cutoff_depth)
{
       //Terminal Node: Run utility function or in this case return its value
       return node->is_game_over() || node->depth() == cutoff_depth;
}
```

```
struct search_result ply::alpha_beta_search(board* root_state, int cutoff_depth,
Player firstPlayer, int alpha, int beta)
{
	LOG("Starting Alpha-Beta Search with Alpha:" << alpha << ", Beta:" <<
beta);
	return (firstPlayer == Player::X) ? max_value_search(root_state,
firstPlayer, cutoff_depth, alpha, beta) : min_value_search(root_state,
firstPlayer, cutoff_depth, alpha, beta);
}

struct search_result ply::max_value_search(board* state, Player player, int
cutoff_depth, int alpha, int beta)
{
	LOG_D(state->depth(), state->tag() << ", MAX-VALUE function, Alpha:" <<
alpha << ", Beta:" << beta);
	struct search_result ret;
	if (cutoff_reached(state, cutoff_depth)) //Cuttoff Test, could be
terminal or depth-limited
	{
		ret.node = state;
		ret.position = state->get_last_position();
		//ret.value = state->evaluate(player);
		ret.value = evaluate(*state, player);
		LOG_D(state->depth(), "Node:" << state->tag() << ", Eval:" <<
ret.value);
		return ret;
	}

	int value = NEG_INF;
	ret.value = value;
	for (board& n : state->successors())
	{
		increase_node_count();
		LOG_D(state->depth(), state->tag() << ", Succ:" << n.tag() << "-
>MIN-VALUE");
		struct search_result ret2 = min_value_search(&n, player,
cutoff_depth, alpha, beta);
		if (value < ret2.value) //MAX(v, MIN-
VALUE(RESULT(s,a),alpha,beta))
		{
			LOG_D(state->depth(), state->tag() << ",CVal: " << value <<
",Min res: " << ret2.value);
			value = ret2.value;
			ret = ret2;
		}
		if (value >= beta)
		{
			//LOG_D(state->depth(), "Returning Node: "<< state->tag()
<< ", as Value("<< value <<") >= Beta" );
			return ret;
		}
		alpha = std::max(alpha, value);
		LOG_D(state->depth(), state->tag() << ", Alpha: " << alpha);
	}
	return ret;
}
```

```
struct search_result ply::min_value_search(board* state, Player player, int
cutoff_depth, int alpha, int beta)
{
	LOG_D(state->depth(), state->tag() << ", MIN-VALUE function, Alpha:" <<
alpha << ", Beta:" << beta);
	struct search_result ret;
	if (cutoff_reached(state, cutoff_depth)) //Cuttoff Test, could be
terminal or depth-limited
	{
		ret.node = state;
		ret.position = state->get_last_position();
		//ret.value = state->evaluate(player);
		ret.value = evaluate(*state, player);
		LOG_D(state->depth(), state->tag() << "Eval:" << ret.value);
		return ret;
	}

	int value = POS_INF;
	ret.value = value;
	for (board& n : state->successors())
	{
		increase_node_count();
		LOG_D(state->depth(), state->tag() << ", Succ:" << n.tag() << ",
calling: MAX-VALUE");
		struct search_result ret2 = max_value_search(&n, player,
cutoff_depth, alpha, beta);
		if (value > ret2.value) //MIN(v, MAX-VALUE(RESULT(s,a),
alpha,beta))
		{
			LOG_D(state->depth(), state->tag() << ",CVal: " << value <<
",Max res: " << ret2.value);
			value = ret2.value;
			ret = ret2;
		}
		if (value <= alpha)
		{
			//LOG_D(state->depth(), "Returning Node: " << state->tag()
<< ", as Value(" << value << ") <= Alpha");
			return ret;
		}
		beta = std::min(beta, value);
		LOG_D(state->depth(), state->tag() << ", Beta: " << beta);
	}
	return ret;
}

struct search_result ply::minimax_ab(board* pos, int depth, Player player, int
use_thres, int pass_thres)
{
	LOG_D(pos->depth(), "MinimaxAB()->Node:" << pos->tag() << ", depth:" <<
depth << ", player:" << PLAYER_NAMES[player == cell_value::X ? 0 : 1] << ", UT:"
<< use_thres << ", PT:" << pass_thres);
	struct search_result rs;
	if (cutoff_reached(pos, depth))
```

```
        {
                //This is absolutely necessary as MiniMaxAB won't work if the MIN
nodes are not flipped negative
                rs.value = (is_x(pos->player()) ? (-1) : 1)*evaluate(*pos,
player);
                //rs.value = evaluate(*pos, player);
                LOG_D(depth, "Return-Terminal Node:" << pos->tag() << ",value:" <<
rs.value);
                rs.node = pos;
                rs.position = pos->get_last_position();
                return rs;
        }/*
        else
        {
                LOG_D(depth, "Deep-Enough false, node: " << pos->tag());
        }*/

        vector<board>& succ_list = pos->successors(); //MOVE_GEN
        LOG_D(pos->depth(), "Node:" << pos->tag() << ", Succ Count:" <<
succ_list.size());
        //Currently we don't need to check for if the succ is empty or not coz we
are defining deep_enough using zero succ
        //But if we are to implement imperfect real-world deep_enough the we have
to add extra checking

        //So Succ isn't empty
        int new_value;
        for (board& succ : succ_list)
        {
                increase_node_count();
                Player swapped_player = SWAP_PLAYER(player);
                struct search_result rs2 = minimax_ab(&succ, depth,
swapped_player, -pass_thres, -use_thres);
                new_value = -rs2.value;
                LOG_D(pos->depth(), pos->tag() << ", Succ:" << succ.tag() << ",
Value:" << rs2.value << ", New-Value:" << new_value);
                if (new_value > pass_thres) // found a succ which is better
                {
                        pass_thres = new_value;
                        rs.node = rs2.node;
                        //rs.position = rs2.node->get_last_position();
                        LOG_D(pos->depth(), pos->tag() << ", Succ:" << succ.tag()
<< ", NV>PT, PT:" << pass_thres << /*", BP:" << rs2.node->tag()*/);
                }

                //if (pass_thres < use_thres) //stop examining this branch
                //{
                //      //std::swap(pass_thres, use_thres);
                //      //??? We are supposed to Swap here but the example didn't
swapped
                //      LOG_D(depth, "Node:"<<pos->tag() << ", Succ:" << succ-
>tag() << ",PT-UT, PT:" << pass_thres << ",UT:" << use_thres);
                //}
                if (pass_thres >= use_thres)
                {
                        rs.value = pass_thres;
```

```
                    LOG_D(pos->depth(), pos->tag() << ", Succ:" << succ.tag()
<< ", PT>=UT, Value:" << pass_thres);
                    //rs.best_path_node is already set

                }
        }

        rs.value = pass_thres;
        //rs.best_path_node is already set
        LOG_D(pos->depth(), "Ret:" << pos->tag() << ", BP:" << rs.node->tag() <<
", Value:" << rs.value);
        return rs;
}
```

Evaluations.h

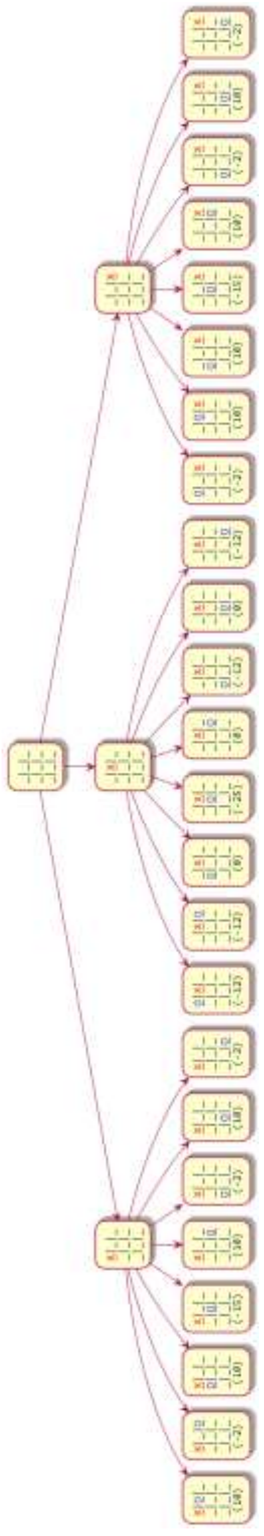
```
#pragma once
#include "board.h"

static int def_attk_eval_score(int product, Player p)
{
        switch (product) {
                //If x is in row with two blanks
        case 12:
                return is_x(p) ? 10 : -10;
                //If o is in row with two blanks
        case 20:
                return is_o(p) ? 10 : -10;
                break;
                //If two x's in row with blanks
        case 18:
                return is_x(p) ? 120 : -100;
                break;
                //If two o's in row with blanks
        case 50:
                return is_o(p) ? 120 : -100;
                break;
        case 27:
                return is_x(p) ? 1000 : -1000;
        case 125:
                return is_o(p) ? 1000 : -1000;
        }

        return 0;
}

static int defensive_attack_eval(board& position, cell_value player)
{
        int total_score = 0;
        total_score += def_attk_eval_score(position.find_product(0, 1, 2),
player);
        total_score += def_attk_eval_score(position.find_product(3, 4, 5),
player);
        total_score += def_attk_eval_score(position.find_product(6, 7, 8),
player);
```

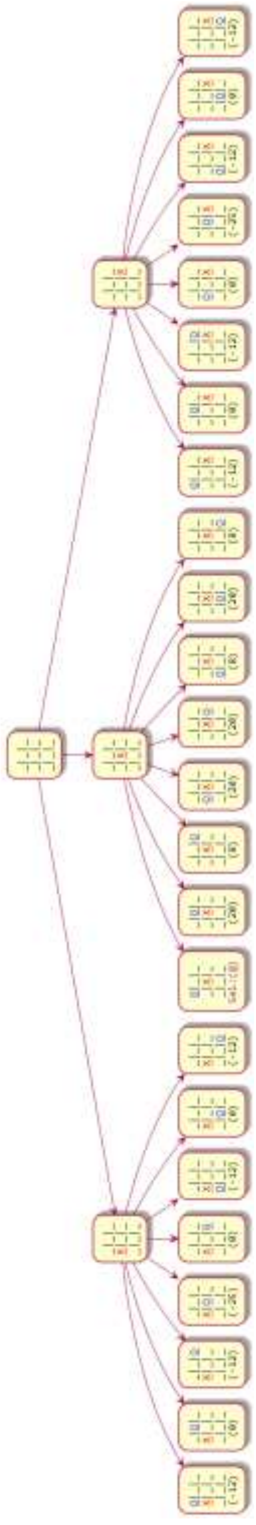
```
	total_score += def_attk_eval_score(position.find_product(0, 4, 8),
player);
	total_score += def_attk_eval_score(position.find_product(2, 4, 6),
player);
	total_score += def_attk_eval_score(position.find_product(0, 3, 6),
player);
	total_score += def_attk_eval_score(position.find_product(1, 4, 7),
player);
	total_score += def_attk_eval_score(position.find_product(2, 5, 8),
player);

	//Extra mark for corner and middle position
	if (is_x(player))
	{
		total_score += (position.get_cell(0).value() == cell_value::X) ? 2
: 0 +
			(position.get_cell(2).value() == cell_value::X) ? 2 : 0 +
			(position.get_cell(4).value() == cell_value::X) ? 5 : 0 +
			(position.get_cell(6).value() == cell_value::X) ? 2 : 0 +
			(position.get_cell(8).value() == cell_value::X) ? 2 : 0;
	}
	else
	{
		total_score += (position.get_cell(0).value() == cell_value::O) ? 2
: 0 +
			(position.get_cell(2).value() == cell_value::O) ? 2 : 0 +
			(position.get_cell(4).value() == cell_value::O) ? 5 : 0 +
			(position.get_cell(6).value() == cell_value::O) ? 2 : 0 +
			(position.get_cell(8).value() == cell_value::O) ? 2 : 0;
	}

	//For MIN case the values have to be reverted
	if (is_o(player))
		total_score = -total_score;

	return total_score;
}

//evaluation that uses the magic square to take the current player's score
versus the opponent's
static int default_eval(board& position, cell_value player) {

	if (position.find_win()) {
		if (is_x(player)) {
			return 27;
		}
		else if (is_o(player)) {
			return 125;
		}
	}

	if (is_x(player)) {
		return position.x_poss_win() - position.o_poss_win();
	}
	else if (is_o(player)) {
		return position.o_poss_win() - position.x_poss_win();
	}
	return -1;
}

//evaluation that has the current player only think about defense (blocking the
other players moves)
static int eval_defense(board& position, cell_value player) {
	if (position.find_win()) {
		if (is_x(player)) {
			return 27;
		}
		else if (is_o(player)) {
			return 125;
		}
	}
	if (is_x(player)) {
		return position.o_poss_win();
	}
	else if (is_o(player)) {
		return position.x_poss_win();
	}
	return -1;
}

//evaluation that has the current player only think about offense (not caring
what the other player is doing)
static int eval_offense(board& position, cell_value player) {
	if (position.find_win()) {
		if (is_x(player)) {
			return 27;
		}
		else if (is_o(player)) {
			return 125;
		}
	}
	if (is_x(player)) {
		return position.x_poss_win();
	}
	else if (is_o(player)) {
		return position.o_poss_win();
	}
	return -1;
}

//Count up desired spots
static int desired_spots(board current, cell_value _value) {
	int score = 0;
	for (int i = 0; i < 9; i++) {
		if (current.get_cell(i).value() == _value) {
			if (i == 4) {
				score += 10;
			}
			else if (i == 0 || i == 2 || i == 6 || i == 8)
			{
				score += 5;
			}
```

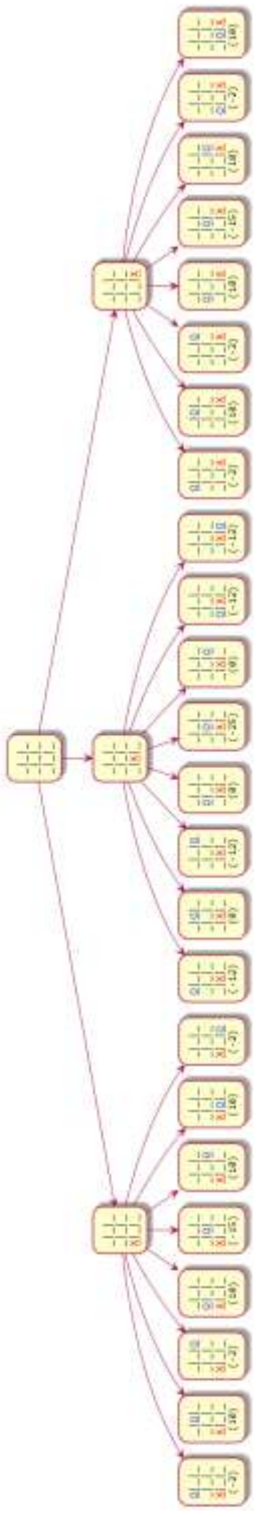
```
                    else {
                            score += 1;
                    }
                }
            }
            return score;
}

static int eval_spot(board& position, cell_value player) {
            if (position.find_win()) {
                if (is_x(player)) {
                    return 27;
                }
                else if (is_o(player)) {
                    return 125;
                }
            }

            return desired_spots(position, player);
}
```

Graphs for Depth 4, Defensive Evaluation

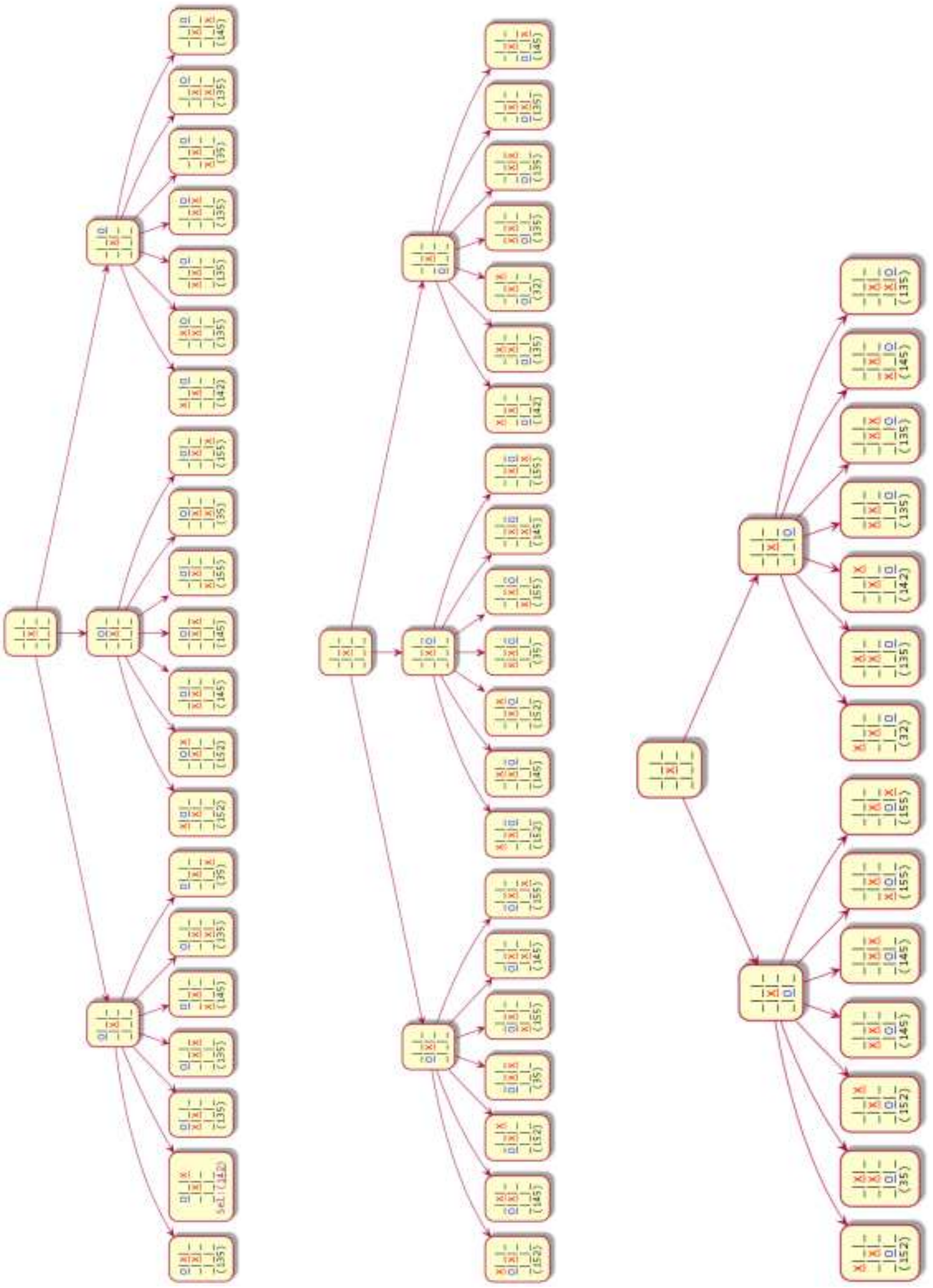
First Step

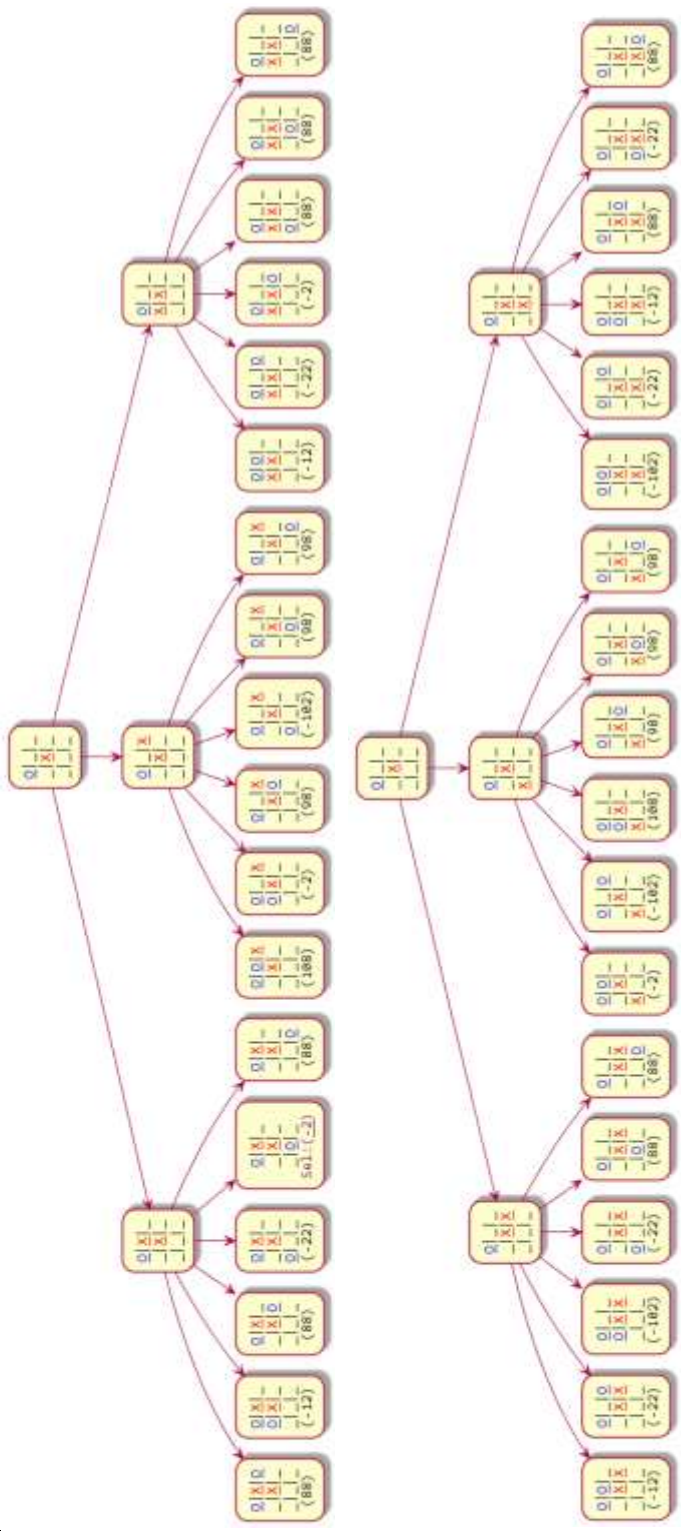




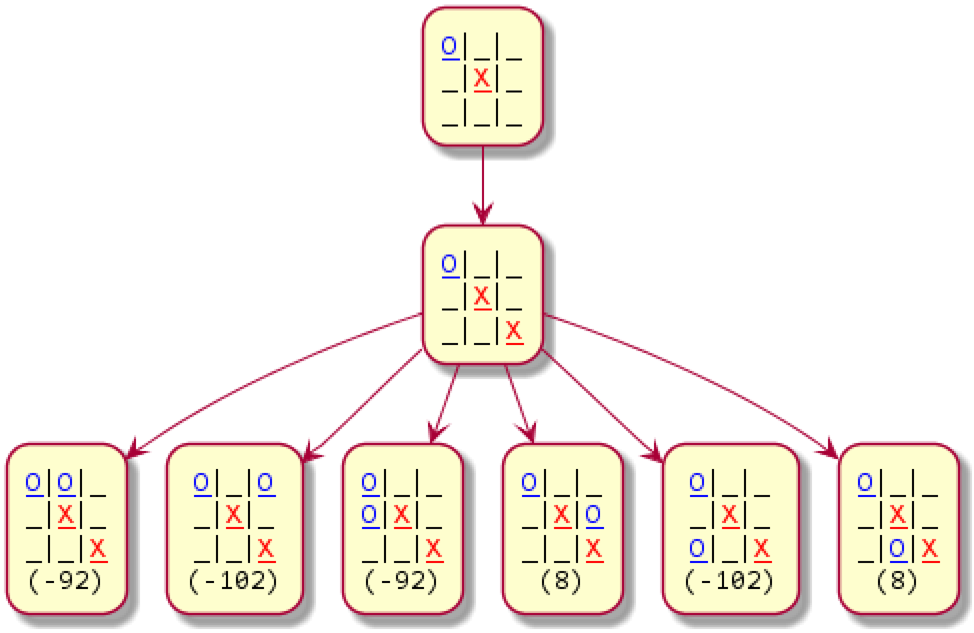


<u>Second Step</u>





Third Step


O|_|_
_|X|_
_|_|_
O|_|_
_|X|_
_|_|X
O|O|_
_|X|_
_|_|X
(-92)
O|_|O
_|X|_
_|_|X
(-102)
O|_|_
O|X|_
_|_|X
(-92)
O|_|_
_|X|O
_|_|X
(8)
O|_|_
_|X|_
O|_|X
(-102)
O|_|_
_|X|_
_|O|X
(8)

Fourth Step