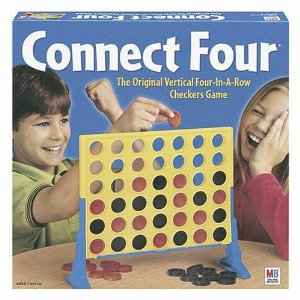
Tutorial – Game Trees – MinMax

Game Tree Theory is an important part of many game genres.

Techniques that apply to Game Tree Search can apply to many different problem domains including A.I. planning for strategy and tactical games, not just to solving board games and puzzles.

In this exercise you are tasked with implementing the MinMax (aka MiniMax) algorithm for an A.I. opponent that can play Connect Four ( <http://en.wikipedia.org/wiki/Connect_Four> ) or Tic-Tac-Toe ( <https://en.wikipedia.org/wiki/Tic-tac-toe> ).

If created in an abstract way your A.I. could be able to play many different types of games without needing to change the underlying code.

The Game Tree Base Code:

On Portal you will find a ZIP file that contains a GameTrees project that is to be added to your Bootstrap solution. It contains a base AI player class and a base Game class, along with a derived **ConnectFour** game and **TicTacToe** game.

Below we will briefly look at each of the provided classes.

The Game class is pure abstract and acts as an interface for a zero-sum two-player game that uses integers to represent actions. The **ConnectFour** game implements a game that uses the integer actions to represent which column to place a piece and also implements how to draw the game. For **TicTacToe** the integer action represents one of the locations in the 3x3 game board.

The **AIPlayer** base class has a method that makes a decision based off a current game state. Also provided is a derived class that makes purely random decisions, **RandomAI**.

The base **Game**, **AIPlayer** and **RandomAI** classes are all contained within a **GameTree** namespace.

The following is the base **Game** class that **ConnectFour** and **TicTacToe** derive from. It implements some base methods that could be used by other game types. As mentioned, Games simply take an integer as an action and make the relevant game-state changes when **performAction()** is called. The state of the game, meaning if the game has been won or if it is a draw, is represented by the enum **ePlayState** which is also used to track active players.

// current state of a zero-sum game

enum ePlayState {

UNKNOWN,

PLAYER\_ONE,

PLAYER\_TWO,

DRAW,

};

// base class to define a zero-sum game with two opponents.

// actions are defined as integers

class Game {

public:

Game() :

m\_currentPlayer(PLAYER\_ONE),

m\_currentOpponent(PLAYER\_TWO) {

}

virtual ~Game() {}

// returns wether someone is winning or not

virtual ePlayState getCurrentPlayState() const = 0;

// these switch whenever an action is made

ePlayState getCurrentPlayer() const { return m\_currentPlayer;}

ePlayState getCurrentOpponent() const {return m\_currentOpponent;}

// get a list of valid actions for current player

virtual bool isActionValid(int action) const = 0;

virtual void getValidActions(std::vector<int>& actions) const =0;

// performs an action for the current player

// and switches current player

virtual void performAction(int action) = 0;

virtual void reverseAction(int action) = 0;

// draw the game

virtual void draw(aie::Renderer2D\* renderer) const = 0;

// clones the current game state

virtual Game\* clone() const = 0;

protected:

ePlayState m\_currentPlayer;

ePlayState m\_currentOpponent;

};

**getValidActions()** will return a list of available actions for the current player.

The following is the A.I. base class that is responsible for making a decision based off of the current game state.

The **makeDecision()** method returns the action that the A.I. should perform.

// base class for an A.I. opponent

class AIPlayer {

public:

AIPlayer() {}

virtual ~AIPlayer() {}

// makes a decision for the current game

virtual int makeDecision(const Game& game) = 0;

};

As an example the following derived A.I. class is provided that makes purely random decisions based off available actions:

// an A.I. opponent that chooses purely random actions to perform

class RandomAI : public AIPlayer {

public:

RandomAI() {}

virtual ~RandomAI() {}

// randomly choose an action to perform

virtual int makeDecision(const Game& game) {

*std*::*vector*<int> actions;

game.getValidActions(actions);

if (actions.*size*() > 0)

return actions[*rand*() % actions.*size*()];

else

return -1;

}

};

The **ConnectFour** class simple implements a game of Connect Four and implements the abstract methods for drawing, cloning and performing actions. The same with **TicTacToe**.

The **clone()** method for **Game** needs to be able to create a brand new copy of the current **Game**, including any derived properties, so in the case of **ConnectFour** and **TicTacToe** it creates a copy of the piece locations as well as which player’s turn it is.

Spend some time looking over the base classes to understand their purpose and use.

The GameTreesApp application class then creates and stores the game along with an AI opponent.

The class declaration contains the following within GameTreesApp.h:

enum {

TICTACTOE,

CONNECTFOUR,

};

int m\_gameType = TICTACTOE;

GameTree::Game\* m\_game;

GameTree::AIPlayer\* m\_ai;

Within the startup of the application we would then initialise the objects, using the **ConnectFour** or **TicTacToe** game and the **RandomAI** for now:

switch (m\_gameType) {

case TICTACTOE: m\_game = new TicTacToeGame(); break;

case CONNECTFOUR:m\_game = new ConnectFourGame(); break;

}

m\_ai = new GameTree::RandomAI();

Within the drawing of your application it calls **draw()** on the game using a **Renderer2D** instance:

// draw the game board

m\_game->draw(m\_2dRenderer);

// display if anyone has won

m\_2dRenderer->setRenderColour(1, 1, 0);

auto result = m\_game->getCurrentPlayState();

if (result == GameTree::ePlayState::PLAYER\_ONE)

m\_2dRenderer->drawText(m\_font, "Player One is the winner!",

0, 0);

else if (result == GameTree::ePlayState::PLAYER\_TWO)

m\_2dRenderer->drawText(m\_font, "Player Two is the winner!",

0, 0);

else if (result == GameTree::ePlayState::DRAW)

m\_2dRenderer->drawText(m\_font, "It's a draw!", 0, 0);

During our Application’s update it updates the game by either waiting for player input or having the AI opponent make a decision.

It does this by tracking the current player’s turn and then converting a mouse-click location to an action for the player, or simply calling **makeDecision()** on the AI:

// keep track of if the mouse has been clicked

static bool moveMade = false;

// if the game isn't over...

if (m\_game->getCurrentPlayState() ==

GameTree::ePlayState::UNKNOWN) {

// if it is the user's turn (player 1)

if (m\_game->getCurrentPlayer() ==

GameTree::ePlayState::PLAYER\_ONE) {

// if the mouse button is down...

if (input->wasMouseButtonPressed(0) ) {

// get the mouse position within the game grid

int x = 0, y = 0;

input->getMouseXY(&x, &y);

int action = -1;

// for connect four

if (m\_gameType == CONNECTFOUR) {

action = (x - 100) / 100;

}

// for tic-tac-toe

else if (m\_gameType == TICTACTOE) {

if ((x - 100) / 100 < 3 &&

(y - 100) / 100 < 3)

action = ((y - 100) / 100) \* 3 +

(x - 100) / 100;

}

if (m\_game->isActionValid(action))

m\_game->performAction(action);

}

}

else {

// it is the opponent's turn (player 2)

// use the A.I. to make a decision

m\_game->performAction(m\_ai->makeDecision(\*m\_game));

}

}

You should be able to run and play the game against the random AI. You can switch which game is played simply by changing the value of **m\_gameType**.

A MinMax Opponent:

We will look at some of the setup required for our new A.I. opponent but we will not cover everything in detail; you will need to set up the class yourself.

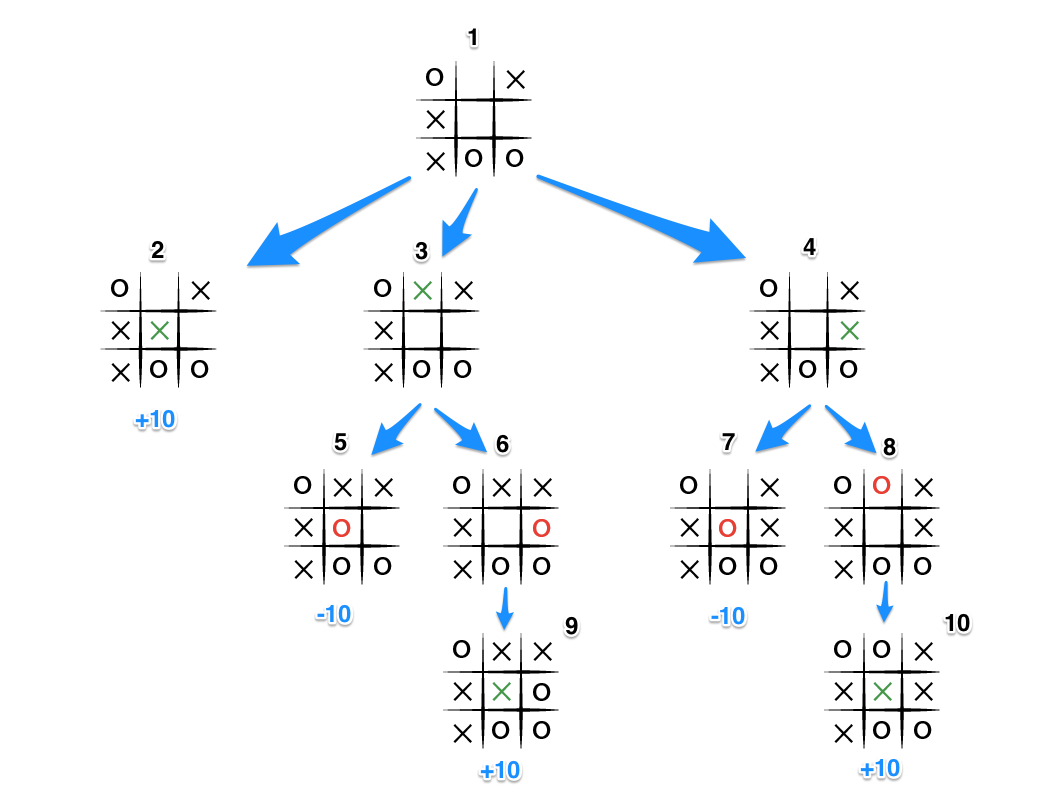
We begin by creating a derived A.I. class. MinMax works by gathering a list of potential actions for the current opponent, creating new game states for each action, then for those states gathering a list of potential actions for the next opponent and creates states for each, so on so forth, until each branch of states has concluded with a finished game.

The states that result in a game being a win for the original opponent whose turn it was receive a positive score, while those that result in a loss for the original opponent receive a negative score. Draws receive a result of 0.

In states that don’t result in a game over the score is calculated differently:

* If the turn of the current branching state is for the original opponent then its result is the highest value that a child state calculated.
* If the turn of the current branching state is not the original opponent’s future turn then its result is that of the lowest scoring child state.

To display that in image form for Tic-Tac-Toe, if it was X’s turn in the following image then it has 3 potential actions, 1 which results in a game-over and 2 which create further branches:



For state 1 it is X’s turn, but for states 3 and 4 if is O’s turn. States 2, 5, and 7 result in game-over, but 6 and 8 branch again (X’s turn).

State 9 and 10 both return a positive result (in this example +10) because it was X’s turn and they’re the opponent we’re trying to find the best move for. But when it was O’s turn they want to reduce our score. O’s moves win in states 5 and 7 so they score -10. Since states 3 and 4 was O’s turn it selects the lowest scoring child scores. State 6 and 8 only have 1 child each so they receive their children’s scores of +10.

So state 5 and 7 are both scored -10, but 6 and 8 are scored +10. Since 3 and 4 was O’s turn they both select their lowest scoring child so 3 and 4’s scores are both -10.

This results in State 1, X’s turn, selecting from its highest scoring child state. State 2 is +10 while state 3 and 4 is -10, so it chooses the action required to create state 2 and thus wins the game.

This all results in a recursive algorithm that we can implement in our derived MinMaxAI class. We can create a recursive method that takes in the current game state and a potential action. It creates a copy of the current game state and performs the action on the copy. If this results in a game-over with the original player winning it returns +10, if they lose it returns -10, and if it is a draw it returns 0.

If it was not a game over then we get a list of potential actions on the copy of the game state, call the recursive algorithm again on the copy of the game-state, passing in the new potential action. Once we have returned a score for each potential action we then:

* If the copied game state’s current player is the same as the original player’s turn then we get the maximum score from the child states.
* If the copied game state’s current player is not the same as the original player’s turn then we get the minimum score from the child states.

def scoreMove(gameState, move)

copyState = gameState

copyState.performAction( move )

if copyState is gameOver

if copyState is draw return 0

if copyState.currentPlayer is currentPlayer return 10

else return -10

actions = copyState.getValidActions

scores : list

for each action in actions

scores.push\_back( scoreMove( copyState, action ) )

if copyState.currentPlayer is currentPlayer return max scores

else return min scores

We can start by setting up our **MinMaxAI** class and implementing the **makeDecision()** method to find potential actions on the current game state, track the player whose turn it is, then tracking the best score from the **scoreMove()** method. This best score will be our AI’s preferred action:

class MinMaxAI : public GameTree::AIPlayer {

public:

MinMaxAI() {}

virtual ~MinMaxAI() {}

GameTree::ePlayState m\_currentPlayer;

virtual int makeDecision(const GameTree::Game& game) {

m\_currentPlayer = game.getCurrentPlayer();

std::vector<int> actions;

game.getValidActions(actions);

float bestScore = -999999;

int bestMove = -1;

// get child move with highest result

for (int action : actions) {

float score = **scoreMove**(game, action);

if (score > bestScore) {

bestScore = score;

bestMove = action;

}

}

return bestMove;

}

};

It is now up to you to implement the **scoreMove()** method, following the preceding pseudo-code. It is recommended you try to set it up to solve the **TicTacToe** game initially as the AI may take a while to make a decision for the **ConnectFour** game as it is a lot more potential moves and a larger game tree.

To use your **MinMaxAI** class simply replace the **RandomAI** opponent within the **GameTreesApp** **setup()** method:

m\_ai = new MinMaxAI();