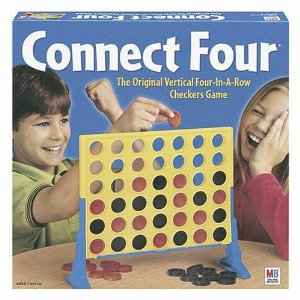
Exercise – Game Trees – Monte Carlo

This exercise follows on from the Game Trees MinMax tutorial.

You do not need to have completed the MinMax tutorial to be able to implement Monte Carlo, but you will need the Game Trees Starter Project.

In this exercise you are tasked with implementing the **Monte Carlo Tree Search** (MCTS) 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> ).

For an overview of the starting project see the MinMax tutorial.

A Monte Carlo Opponent:

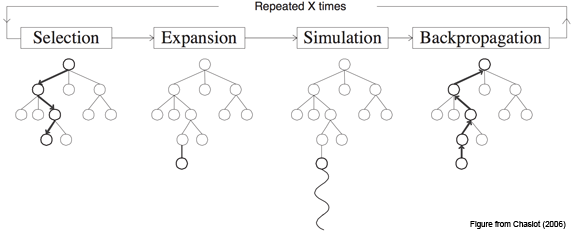
We will look at some of the setup required for our new A.I. opponent but we will not cover everything in detail; ***you will need to set up the class yourself***.

MCTS works by using:

* Selection
* Expansion
* Simulation
* Backpropagation

Our MCTS implementation will work as follows:

* **Selection**: Our A.I. selects one of its potential moves it could perform.
* **Expansion**: Our A.I. creates a clone of the game state and performs the selected action.
* **Simulation**: Our A.I. then simulates the game randomly, randomising its and its opponent’s moves, until the game ends.
* **Backpropagation**: The game is given a score and this score is returned and accumulated for that selected action.

We loop these steps for a certain number of **playouts**. This allows us to try and formulate a statistical average chance of each action resulting in a win. Whichever action has the highest accumulated score is assumed to have the best chance of winning. Our A.I. simply returns this action as its chosen move to perform.

We begin by creating a derived **AIPlayer** class, **MonteCarloAI**. The constructor for the class can be used to specify how many playouts we want it to perform, which potentially can be used as a difficulty modifier: more playouts the harder the A.I. will be. Less playouts the more likely it is to make a mistake.

We can then implement its **makeDecision()** method to be similar to the following pseudocode:

Expansion would be a function that takes the current game state and a selected action, creates a clone of the game state and then simulates the cloned game to completion, eventually returning a score; positive for a win, negative for a loss, and 0 for a draw.

def makeDecision(gameState)

actions = gameState valid actions

scoreForAction : map< action, score >

for each playout

select random action in actions

score = expand gameState for action

scoreForAction[ action ] += score

return action with highest accumulated score

We could create the expand method in the following way:

def expand(gameState, action)

gameClone = gameState clone

gameClone perform action

winner = simulate gameClone

delete gameClone

if winner is this AI then return positive

else if winner is opponent then return negative

else return 0

Simulation would simply be a method that loops while the game is not over and performs a random action for whichever player’s turn it is:

def simulate(gameClone)

while gameClone not game over

actions = gameClone valid actions

action = random action in actions

gameClone perform action

return gameClone winner or draw

You are now responsible for implementing the preceding psuedocode within the Game Trees Starter Project within your **MonteCarloAI** class:

class MonteCarloAI : public GameTree::AIPlayer {

public:

MonteCarloAI(int playouts) : m\_playouts(playouts) {}

virtual ~MonteCarloAI() {}

int m\_playouts;

// perform playouts on the current game state

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

// clone game state and simulate the action being performed

float expand(const GameTree::Game& game, int action);

// randomise all turns until game is over

GameTree::ePlayState simulate(GameTree::Game& game);

};

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

m\_ai = new MonteCarloAI(10000);