In this code, I have created an AI to optimally play mine-race. I used recursion to iterate through all possible game-state, starting from the provided game-state

I used min-maxing algorithm to determine optimal moves, used a map data structure to utilize dynamic programming and avoid redundancy in the game-state iterations, storing in it game state inputs as the keys, the win-lose outcome as the value t said key. Finally I added alpha-beta pruning to further optimize the code. The program automatically generates the game tree at start-up.

//Code of Bot.h

#pragma once

#include <SFML/Graphics.hpp>

#include <random>

#include <thread>

#include <chrono>

#include "Mine.h"

#include <fstream>

#include <string>

#include <iostream>

#include <map>

class Bot {

public:

Bot(){

map = {};

std::random\_device rd;

eng.seed(rd());

}

//Function to generate a game tree and store it in file

void generateGameTree(std::ofstream &tree, int playerNum = 1, int remainingDistance = Mine::length, int depth = 0);//Calculates number of total possible ways of winning for player 1 from any position

//minMax algorithm to calculate winning and losing paths, starting from a specified remaining distance

int minMax(Mine& mine, int remainingDistance = Mine::length, bool isBot = true, int depth = 0);

//easy mode version, the AI that chooses moves at random

void chooseRandom(Mine& mine);

private:

std::mt19937 eng;

std::map<std::pair<int, int>, int > map;//Map to use dynamic programming and optimize min-max algorith

};

void Bot::generateGameTree(std::ofstream &tree, int playerNum, int remaining, int depth) {

if (depth == 0)

map = {};//buffer Map can be reused for minMaxing. But not for generating game tree (bacause I said so)

if (playerNum > 2)

playerNum = 2 - playerNum % 2;

if (map.count({ remaining, depth }) != 0 && remaining > 0) { //check If we have already had an identical game state

//If the game is already over. This won't optimize much and its better to just say who won

tree << "But this same scenario has happened already.\n";

return;

}

if (remaining > 0)

tree << remaining << " meters remain.\n";

else

tree << "Player " << 2 - (playerNum + 1) % 2 << " wins\n";

if (remaining <= 0) {

if (playerNum == 1)

{

map[{remaining, depth}] = 0;

return;

}

else {

map[{remaining, depth}] = 0;

return;

}

}

//Add padding to the tree so we can better see what is happening

for (int i = 0; i < depth; i++)

tree << "|";

tree << "Player " << playerNum << " uses pickaxe. ";

generateGameTree(tree, playerNum + 1, remaining - 5, depth + 1);

for (int i = 0; i < depth; i++)

tree << "|";

tree << "Player " << playerNum << " uses drill. ";

generateGameTree(tree, playerNum + 1, remaining - 10, depth + 1);

for (int i = 0; i < depth; i++)

tree << "|";

tree << "Player " << playerNum << " uses nuke. ";

generateGameTree(tree, playerNum + 1, remaining - 30, depth + 1);

map[{remaining, depth}] = 0;

if (depth == 0)

map = {};//The map is a buffer. So we empty it after each use

}

int Bot::minMax(Mine& mine, int remaining, bool isBot, int depth) {

//if (map.count({ remaining, depth }) != 0) { //Check f we have already had an identical game state

// return map[{remaining, depth}];

//}

if (remaining <= 0) {//If the end of mine has already been reached, the previous player has won

if (!isBot)//If the current turn is not bot's, then previous turn was, so bot won

return map[{remaining, (int)isBot}] = 1;

else//The bot lost in this case

return map[{remaining, (int)isBot}] = 0;

}

//Calculating if these moves lead to winning or losing

int pickaxe = minMax(mine, remaining - 5, !isBot, depth + 1);

if (pickaxe == 1 && isBot == true && depth > 0) //Min-max algorithm alpha-beta pruning

return 1;

if (pickaxe == 0 && isBot == false && depth > 0)//Min-max algorithm alpha-beta pruning

return 0;

int drill = minMax(mine, remaining - 10, !isBot, depth + 1);

if (drill == 1 && isBot == true && depth > 0)//Min-max algorithm alpha-beta pruning

return 1;

if (drill == 0 && isBot == false && depth > 0)//Min-max algorithm alpha-beta pruning

return 0;

int nuke = minMax(mine, remaining - 30, !isBot, depth + 1);

if (drill == 1 && isBot == true && depth > 0)//Min-max algorithm alpha-beta pruning

return 1;

if (drill == 0 && isBot == false && depth > 0)//Min-max algorithm alpha-beta pruning

return 0;

if (depth == 0) {//Once everyhing has been executed, the bot makes the move. The best option

if (pickaxe > drill && pickaxe > nuke) {

mine.dig(5);

std::cout << "Pickaxe was chosen: " << pickaxe << " points\n";

}

else if (drill > nuke) {

mine.dig(10);

std::cout << "Drill was chosen: " << drill << " points\n";

}

else {

mine.dig(30);

std::cout << "nuke was chosen: " << nuke << " points\n";

}

}

//Min-max algorithm

if (isBot)

return std::max(pickaxe, std::max(drill, nuke));

else

return std::min(pickaxe, std::min(drill, nuke));

}

void Bot::chooseRandom(Mine& mine){

std::uniform\_int\_distribution<int> distr(1, 3);

int rand = distr(eng);

switch (rand) {

case 1:

mine.dig(5);

break;

case 2:

mine.dig(10);

break;

case 3:

mine.dig(30);

break;

}

}

//Code of Mine.h

#pragma once

#include <SFML/Graphics.hpp>

class Mine

{

private:

public:

Mine(sf::Vector2f size, sf::Vector2f position);

void dig(float distance);

bool endReached()const { return minedDistance >= length; }

void restart();

sf::RectangleShape getEarthShape()const { return earthShape; }

sf::Texture getEarthTexture()const { return earthTexture; }

sf::RectangleShape getMined()const { return minedArea; }

float getMinedDistance()const { return minedDistance; }

float getRemainingDistance()const { return length - minedDistance; }

private://parameters

float minedDistance = 0.f;

sf::RectangleShape earthShape;

sf::Texture earthTexture;

sf::RectangleShape minedArea;

public:

static constexpr float length = 100.f;

};

Mine::Mine(sf::Vector2f size, sf::Vector2f position):

earthShape(size),

minedArea({ size.x, size.y / 12 })

{

earthShape.setPosition(position);

earthShape.setFillColor(sf::Color(140, 40, 40));

earthTexture.loadFromFile("images\\Earth.jpg");

minedArea.setFillColor(sf::Color(20, 10, 1));

minedArea.setScale(0.f, 1.f);

minedArea.setPosition(position.x, position.y + size.y \* 0.7f);

}

inline void Mine::dig(float distance){

minedDistance += distance;

minedArea.setScale(minedDistance / length, 1.f);

}

inline void Mine::restart(){

minedDistance = 0;

minedArea.setScale(0.f, 1.f);

}

//Code of Button.h

#pragma once

#include <SFML/Graphics.hpp>

#include <iostream>

class Button {

public:

Button(sf::Vector2f position, sf::Vector2f size, sf::String textureFile);

Button(sf::Vector2f position, sf::Vector2f size, sf::Color color = sf::Color::Green, sf::String text = "");

~Button() { delete texture; }

bool isSelected(sf::RenderWindow& window);

sf::Texture\* const& getTexture()const { return texture; }

sf::RectangleShape const &getRect()const { return rect; }

sf::String const& getText()const { return buttonText; }

private:

sf::Texture \*texture = nullptr;

sf::RectangleShape rect;

sf::String buttonText = "";

};

Button::Button(sf::Vector2f position, sf::Vector2f size, sf::String textureFile){

texture = new sf::Texture;

texture->loadFromFile(textureFile);

rect.setSize(size);

rect.setPosition(position);

rect.setTexture(texture);

}

inline Button::Button(sf::Vector2f position, sf::Vector2f size, sf::Color color, sf::String text){

rect.setSize(size);

rect.setPosition(position);

rect.setOutlineColor(color);

rect.setOutlineThickness(3.f);

rect.setFillColor(sf::Color::Transparent);

buttonText = text;

}

inline bool Button::isSelected(sf::RenderWindow& window)

{

sf::Vector2i pos = sf::Mouse::getPosition(window);

sf::Vector2i rPos = (sf::Vector2i)rect.getPosition();//position of the rectangle

sf::Vector2i rSize = (sf::Vector2i)rect.getSize(); //size of the vector

return pos.x >= rPos.x && pos.y >= rPos.y && pos.x <= rPos.x + rSize.x && pos.y <= rPos.y + rSize.y;

}

//Code of Displayer.h

#pragma once

#include <SFML/Graphics.hpp>

#include <string>

#include "Button.h"

#include "Mine.h"

class Displayer {

public:

Displayer(sf::RenderWindow& window) : window(window) {}

void draw(Mine& mine);

void draw(Button& button);

void setFont(std::string fontFile) { font.loadFromFile(fontFile); }

void displayText(sf::Vector2f position, sf::String text, int size = 30, sf::Color color = sf::Color::White)const;

void displayMined(int minedDistance);

void displayTurn(int turn);

private:

sf::RenderWindow& window;

sf::Font font;

};

void Displayer::draw(Mine& mine) {

//Setting the texture of the earth and drawing it

sf::Texture earthTexture = mine.getEarthTexture();

sf::RectangleShape earthShape = mine.getEarthShape();

earthShape.setTexture(&earthTexture);

window.draw(earthShape);

sf::RectangleShape uiBorder({ (float)window.getSize().x, 5.f });

uiBorder.setFillColor(sf::Color::White);

//Displaying border between user interface and the mine from top and bottom

uiBorder.setPosition(0.f, earthShape.getPosition().y);

window.draw(uiBorder);

uiBorder.setPosition(0.f, earthShape.getPosition().y + earthShape.getSize().y);

window.draw(uiBorder);

//Displaying the area that has been mined

window.draw(mine.getMined());

}

void Displayer::draw(Button& button){

sf::RectangleShape rect = button.getRect();

sf::Texture\* texture = button.getTexture();

if(texture != nullptr)

rect.setTexture(texture);

else {

displayText({ rect.getPosition().x + 23.f, rect.getPosition().y + 5.f}, button.getText(), 30, rect.getOutlineColor());

}

window.draw(rect);

}

void Displayer::displayText(sf::Vector2f position, sf::String textInp, int size, sf::Color color)const {

sf::Text text(textInp, font, size);

std::string str = textInp;

text.setFillColor(color);

text.setPosition(position);

window.draw(text);

}

void Displayer::displayMined(int minedDistance){

sf::String text;

if (minedDistance >= 100)

text = "The mining is over.";

else

text = std::to\_string(minedDistance) + " meters have been mined";

displayText({50.f, 80.f }, text);

}

inline void Displayer::displayTurn(int turn){

displayText({ 500.f, 80.f }, "Player " + std::to\_string(2 - turn % 2) + "'s turn");

}

//code of Main.cpp

#include <SFML/Graphics.hpp>

#include "Displayer.h"

#include "Mine.h"

#include "Button.h"

#include "Bot.h"

int main()

{

sf::RenderWindow window(sf::VideoMode(800, 800), "mine-race 0.1", sf::Style::Close | sf::Style::Titlebar);

std::ofstream gameTree("Game Tree.txt");

enum class GameState { menu, ongoing, over }gameState = GameState::menu;

sf::Vector2f winSize = (sf::Vector2f)window.getSize();

int turnCounter = 1;

int playerNum = 1; //By default you are player 1

Bot bot;

Displayer displayer(window);

displayer.setFont("fonts\\comic.ttf");

Mine mine({ winSize.x - 50, winSize.y \* 0.6f }, { 50, winSize.y \* 0.15f });

const sf::Vector2f buttonSize = { 180.f, 180.f };

Button SelectPlayer1({ 200.f, winSize.y / 2 }, { 150.f, 50.f }, sf::Color::Red, "Player 1");

Button SelectPlayer2({ 450.f, winSize.y / 2 }, { 150.f, 50.f }, sf::Color::Blue, "Player 1");

Button robot({ 300.f, 250.f }, { 200.f, 200.f }, "images\\robot.png");

Button pickaxe({ 10.f, winSize.y - (buttonSize.y + 5.f) }, buttonSize, "images\\pickaxe.png");

Button drill({ (winSize.x - buttonSize.x) / 2.f, winSize.y - (buttonSize.y + 5.f) }, buttonSize, "images\\drill.jpg");

Button nuke({ winSize.x - 10.f - buttonSize.x, winSize.y - (buttonSize.y + 5.f) }, buttonSize, "images\\nuke.jpg");

Button restart({ winSize.x / 2 - 120.f, 400.f }, { 240.f, 100.f }, "images\\restart.png");

std::map< std::pair<int, int>, int> map = {};

bot.generateGameTree(gameTree);

while (window.isOpen())

{

sf::Event event;

while (window.pollEvent(event))

{

switch (event.type)

{

case sf::Event::Closed:

window.close();

case sf::Event::MouseButtonPressed:

switch (gameState)

{

case GameState::menu:

if (SelectPlayer1.isSelected(window)) {

playerNum = 1;

gameState = GameState::ongoing;

}

if (SelectPlayer2.isSelected(window)) {

playerNum = 2;

gameState = GameState::ongoing;

}

break;

case GameState::ongoing:

//If it is players turn

if (turnCounter == playerNum) {

if (pickaxe.isSelected(window)) {

mine.dig(5);

++turnCounter;

}

else if (drill.isSelected(window)) {

mine.dig(10);

++turnCounter;

}

else if (nuke.isSelected(window)) {

mine.dig(30);

++turnCounter;

}

}

// If it is bots turn

else {

if (!robot.isSelected(window)) continue;

bot.minMax(mine, mine.getRemainingDistance());

while (window.pollEvent(event));

++turnCounter;

}

if (mine.endReached()) {

gameState = GameState::over;

++turnCounter;

}

break;

case GameState::over:

if (restart.isSelected(window)) {

turnCounter = 1;

mine.restart();

gameState = GameState::menu;

}

break;

}

break;

}

}

if (turnCounter > 2) turnCounter = 2 - turnCounter % 2;

window.clear();

switch (gameState) {

case GameState::menu:

displayer.draw(SelectPlayer1);

displayer.draw(SelectPlayer2);

displayer.displayText({ 100.f, 300.f }, "Are you player 1 or player 2?", 40, sf::Color::Green);

break;

case GameState::ongoing:

displayer.draw(mine);

displayer.draw(pickaxe);

displayer.draw(drill);

displayer.draw(nuke);

displayer.displayMined((int)mine.getMinedDistance());

displayer.draw(robot);

displayer.displayTurn(turnCounter);

break;

case GameState::over:

displayer.draw(mine);

sf::String resultMessage;

if (turnCounter == playerNum)

displayer.displayText({ 235.f, 300.f }, "YOU WON!", 70, sf::Color::Yellow);

else

displayer.displayText({ 260.f, 300.f }, "you lost", 70, sf::Color::Yellow);

displayer.draw(restart);

break;

}

window.display();

}

return 0;

}