About the Project

(The dos and don’ts and a summary of code logic)

The greatest feature of this project sourcecode is its portability.
It is entirely written in standard C++.
No `#include<windows.h>`, no use of external libraries like ncurses.
Hence, though parts of the code have to be commented out or uncommented, it is possible to compile and run the program on Windows, Linux and even on online compilers.
Code in `/**Windows only*/  /**Windows end*/` needs to be enabled on Windows.
Code in `/**Linux only*/  /**Linux end*/` should be enabled for Linux builds.

Since this project aims to create a game, it is essential to clear the screen every-so-often.
The `cls` macro was defined to achieve this.

```
#define cls system("cls") //clear screen on Windows
#define cls system("clear") //clear screen on Linux
#define cls cout << "\033[2J\033[1;1H" //ANSI code to
clear screen on both Windows 10 and Linux.
Not used to enable execution on old Windows OSes.
```

Note that although the ANSI code is more versatile, Windows does not support ANSI codes. Windows 10 does, but it is not default behaviour.
[See: https://docs.microsoft.com/en-us/windows/console/console-virtual-terminal-sequences and https://en.wikipedia.org/wiki/ANSI_escape_code#DOS,_OS/2,_and_Windows]

The settings are stored in the settings.ini file.
I advise against directly modifying the configuration file.
In case of unintended modifications or program errors, delete the file and you are good to go. [Default values will be used in such a case.]

The macro STEPS defines the number of moves the minimax algorithm will calculate in advance to find the optimal path.
Increase it for a more challenging game.

Higher values lead to more resource consumption and lower execution speed.

AI vs AI matches are possible, but not recommended.
This is because some of the assumptions made about the player's behaviour are invalid for an AI.

The execution begins from the `main()` method. Other than the `cls` macro, all calls to `system()` are made here. It displays the welcome screen, seeds the random number generator, loads the settings and shows the Main Menu.

The game mechanism is handled by the `play()` method. It interacts with the user, deploys the relevant game engines and synchronizes the entire play process.

The `Board` is where the game is played. It stores details about the player's units and provides methods to interact with itself.

The `Units` are those that are moved by the players and take a beating from those of the opponent.

Refer to Documentation for details.

In the Minimax algorithm the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.
Every board state has a value associated with it.
In a given state if the maximizer has upper hand, then the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value.
The values of the board are calculated by some heuristics which are unique to every game.
There is a difference between `AIgreedy()` and `AIminimax()` with value of STEPS set to 1. While `AIminimax()` ranks the state of board after a move is made, `AIgreedy()` ranks the moves themselves.

Sourcecode

Header.h

```
/
*************************************************************
***************************
* @author Swastik Pal
* @version 1.0
* Date 28/03/2021
* About: This header file contains everything
needed for this project
*************************************************************
*****************************/

#ifndef HEADER_H
#define HEADER_H

//macros here
#define TOTAL_COL 120
#define TOTAL_ROW 50
// #define TOTAL_COL 200
// #define TOTAL_ROW 60
#define PLAYER_1 0
#define PLAYER_2 1
#define COMPUTER 1
/**Windows only*/
#define cls system("cls") //clear screen on Win-
dows
/**Windows end*/
/**Linux only**/
//#define cls system("clear") //clear screen on
Linux
/**Linux end*/
//#define cls cout << "\033[2J\033[1;1H" //code to
clear screen on both Windows 10 and Linux. Not
used to enable execution on old Windows OSes

//header files here
```

```
#include<iostream>
#include<fstream>
#include<string>

//namespace items here
using std::cin;
using std::cout;
using std::endl;
using std::fstream;
using std::ifstream;
using std::ofstream;
using std::string;

//global variables here
enum controller{human, randomAI, greedyAI, minim-
axAI};
extern controller P1control, P2control;
extern string controllerNames[];
extern char unitLabels[2][4];
extern string unitNames[];
extern bool enableTraits;

//classes here
struct Pos//Unit locations in alphabetical order
K, P, R, S//Unit locations as HV; like A1, B6
{
    char col;
    int row;
};

class Unit
{
    private:
    int attack;
    int defence;
    int movement;
    int health;
    int contribution;//for debugging only
    int index;
    public:
    bool isDead;
    Pos position;
    Unit(int);
    Unit* getCurrentInstance(){return this;}
    string getName(){return unitNames[index];}
    int getAttack(){return attack;}
    int getDefence(){return defence;}
    int getMovement(){return movement;}
    int getHealth(){return health;}
    void setHealth(int hp){health = hp;}
    int getIndex(){return index;}
    int getContribution(){return contribution;}
    void setContribution(int c){contribution = c;}
};

class Board
{
    private:
    static const char HLabels[10];//
={'X','A','B','C','D','E','F','G','H','I'};
    static const int VLabels[9];//
={1,2,3,4,5,6,7,8,9};
    char playArea[9][9];
    double state;//calculated as (Player_2-
Player_1) (total_health -
units_dead*WEIGHTAGE_OF_DEATH)
    class Player
    {
        public:
        Unit unitSet[4]={Unit(0), Unit(1),
Unit(2), Unit(3)};

    } player[2];

    public:

    Board();
    void clearPos(char&, int&);
    void clone(Board board);
    void display();
    Board* getCurrentInstance(){return this;}
    double getState();
    Unit* getUnit(int, int);
    bool hasRemainingUnits(int);
    bool isLegalMove(int, Unit*, char, int);
    int isOccupied(char, int);
    int makeMove(int, Unit*, char, int);
    void reset();
    void setPos(int, char, int, int);
    void showDetails();
};

//functions here
void timed_delay(double);
void movexy(int, int);
int sgn(int);
void welcomeScreen();
void play();
void help();
void credits();
void settings();
void menu();
bool hasEnded();
void showUnit(Unit*);
void manual(int=PLAYER_1);
void AIrandom(int=COMPUTER);
void AIgreedy(int=COMPUTER);
void AIminimax(int=COMPUTER);
void loadState(string, Board*);
void saveState(string, Board*);

#endif
```

Main.cpp

```
/
***************************************************
****************************
* @author Swastik Pal
* @version 1.0
* Date 28/03/2021
* About: The main() method and the different
screens
***************************************************
*****************************/

//macros here

//header files here
#include"header.h"
#include<ctime>
#include<cstdlib>

//namespace items here

//global variables
controller P1control, P2control;
string controllerNames[] = {"human", "randomAI",
"greedyAI", "minimaxAI"};
bool enableTraits;

//functions here
void timed_delay(double t)/**makes program wait
for t seconds*/
{
    //complete
    clock_t delay_st,delay_end;
    delay_st=clock();
    delay_end=clock();
    while((double)(delay_end-delay_st)/
CLOCKS_PER_SEC<t)
        delay_end=clock();
```

```
}

void movexy(int col, int row)/**moves cursor by
specified steps while filling up the screen*/
{
    //complete
    for(int i=1;i<=row;i++)
        cout<<'\n';
    for(int i=1;i<=col;i++)
        cout<<' ';
}

void welcomeScreen()/**title screen*/
{
    //complete
    cls;
    char text[]="Little Soldiers";
    movexy(TOTAL_COL/2 - 8, TOTAL_ROW/2);
    for(int i=0;text[i]!='\0';i++)
    {
        cout<<text[i];
        timed_delay(0.1);
    }
    timed_delay(0.15);
}

void help()/**shows help screen*/
{
    //complete
    cls;
    ifstream helptext;
    helptext.clear();
    helptext.open("helptext.txt");
    if(helptext.fail())//checks whether read oper-
ation failed
        cout<<"Unable to open help file\nError";
    else
        cout<<helptext.rdbuf();
    helptext.close();
    cout<<"\n\n\nPress X to continue...";
    char ch;
    do
    {
        cin>>ch;
    }while(ch!='X');
}

void credits()/**shows credits*/
{
    //complete
    cls;
    movexy(TOTAL_COL/2 - 10, TOTAL_ROW/2);
    cout<<"Thanks for playing.";
    movexy(TOTAL_COL/2 - 13, 1);
    cout<<"Hope you enjoyed the game.";
    timed_delay(1.5);
    cls;
    movexy(TOTAL_COL/2 - 4, TOTAL_ROW/2);
    cout<<"Made with C++";
    timed_delay(1.15);
    cls;
    movexy(TOTAL_COL/2 - 12, TOTAL_ROW/2);
    cout<<"This game is a freeware.";
    movexy(TOTAL_COL/2 - 12, 1);
    cout<<"Feel free to distribute.";
    timed_delay(1.2);
    cls;
    movexy(TOTAL_COL/2 - 5, TOTAL_ROW/2);
    cout<<"With love:";
    movexy(TOTAL_COL/2 - 8, 1);
    cout<<"From Swastik Pal";
    timed_delay(1.2);
}

void loadSettings()
{
     fstream inifile;
     inifile.clear();
    inifile.open("settings.ini", std::ios::in);
    if(inifile.fail())//checks whether read opera-
tion failed
    {
        cout<<"Unable to open settings file\
nError";
        //load defaults
        P1control = controller::human;
        P2control = controller::minimaxAI;
    }
    else
    {
        char ch;
        int propertyNo = 0;
        while(!inifile.eof())
        {
            ch = inifile.get();
            if((ch == '=') && (inifile.peek()>='0'
&& inifile.peek()<='9'))
            {
                ch = inifile.get();
                ++propertyNo;
                switch(propertyNo)
                {
                    case 1:
                        P1control =
static_cast<controller>(ch - '0');
                        break;
                    case 2:
                        P2control =
static_cast<controller>(ch - '0');
                        break;
                    case 3:
                        enableTraits =
static_cast<bool>(ch - '0');
                }
            }
        }
        if(propertyNo==0)
        {
            //load defaults
            P1control = controller::human;
            P2control = controller::minimaxAI;
            enableTraits = false;
        }
    }
    inifile.close();
}

void settings()
{
    cls;
    fstream inifile;
    inifile.clear();
    inifile.open("settings.ini", std::ios::out);
    if(inifile.fail())//checks whether read opera-
tion failed
        cout<<"Unable to open settings file\
nError";
    else
    {
        char ch;
        while(true)
        {
            cls;
            movexy(TOTAL_COL/2 - 3, TOTAL_ROW/2 -
6);
            cout<<"Menu:";
            movexy(TOTAL_COL/2 - 15, 2);
            cout<<"1> Player 1 Controller: "<<con-
trollerNames[P1control];
            movexy(TOTAL_COL/2 - 15, 1);
            cout<<"2> Player 2 Controller: "<<con-
trollerNames[P2control];
            movexy(TOTAL_COL/2 - 15, 1);
            cout<<"3> Enable Traits: "<<(enable-
Traits?"true":"false");
            movexy(TOTAL_COL/2 - 15, 1);
            cout<<"4> Back";
            movexy(TOTAL_COL/2 - 6, 2);
            cout<<"Your choice?";
            cin>>ch;
            if(ch=='4')
                break;
            switch(ch)
            {
                case '1':
                    while(true)
                    {
                        char choice;
                        cls;
                        movexy(TOTAL_COL/2 - 10,
TOTAL_ROW/2 - 6);

                        cout<<"Player 1 Control-
ler: ";

                        for(int i=0; i<4; i++)
                        {
                            movexy(TOTAL_COL/2 -
5, 1);

                            cout<<(i+1)<<">
"<<controllerNames[i];
                        }
                        movexy(TOTAL_COL/2 - 10,
1);

                        cout<<"Enter choice:";
                        cin>>choice;
                        if(choice>='1' ||
choice<='4')

                        {
                            P1control =
static_cast<controller>(choice - '1');
                            break;
                        }
                    }
                    break;
                case '2':
                    while(true)
                    {
                        char choice;
                        cls;
                        movexy(TOTAL_COL/2 - 10,
TOTAL_ROW/2 - 6);

                        cout<<"Player 2 Control-

ler: ";

                        for(int i=0; i<4; i++)
                        {
                            movexy(TOTAL_COL/2 -
5, 1);

                            cout<<(i+1)<<">
"<<controllerNames[i];
                        }
                        movexy(TOTAL_COL/2 - 10,
1);

                        cout<<"Enter choice:";
                        cin>>choice;
                        if(choice>='1' ||
choice<='4')
                        {
                            P2control =
static_cast<controller>(choice - '1');
                            break;
                        }
                    }
                    break;
                case '3':
                    while(true)
                    {
                        char choice;
                        cls;
                        movexy(TOTAL_COL/2 - 7,
TOTAL_ROW/2 - 6);

                        cout<<"Enable Traits: ";
                        movexy(TOTAL_COL/2 - 7,
1);

                        cout<<"Enter choice(Y/

N):";
```

```
                        cin>>choice;
                        if(choice=='n' ||
choice=='N')
                        {
                            enableTraits = false;
                            break;
                        }
                        else if(choice=='y' ||
choice=='Y')
                        {
                            enableTraits = true;
                            break;
                        }
                    }
                    break;
            }
        }
    }
    inifile<<"[Settings]\n";
    inifile<<"P1control="<<(int)P1control<<"\n";
    inifile<<"P2control="<<(int)P2control<<"\n";
    inifile<<"enableTraits="<<(int)enable-
Traits<<"\n";
    inifile.close();
}

void showCharacterCards()
{
    //complete
    cls;
    ifstream cards;
    cards.clear();
    cards.open("character_cards");
    if(cards.fail())//checks whether read opera-
tion failed
        cout<<"Unable to open cards file\nError";
    else
        cout<<cards.rdbuf();
    cards.close();
    cout<<"\n\n\n\nPress X to continue...";
    char ch;
    do
    {
        cin>>ch;
    }while(ch!='X');
}

void menu()/**shows main menu*/
{
    char ch;
    while(true)
    {
        cls;
        movexy(TOTAL_COL/2 - 3, TOTAL_ROW/2 - 6);
        cout<<"Menu:";
        movexy(TOTAL_COL/2 - 6, 1);
        cout<<"1> Play";
        movexy(TOTAL_COL/2 - 6, 1);
        cout<<"2> Help";
        movexy(TOTAL_COL/2 - 6, 1);
        cout<<"3> Show Character Cards";
        movexy(TOTAL_COL/2 - 6, 1);
        cout<<"4> Settings";
        movexy(TOTAL_COL/2 - 6, 1);
        cout<<"5> Exit";
        movexy(TOTAL_COL/2 - 6, 2);
        cout<<"Your choice?";
        cin>>ch;
        switch(ch)
        {
            case '1':
                play();
                break;
            case '2':
                help();
                break;
            case '3':
                showCharacterCards();
                break;
                break;
            case '4':
                settings();
                break;
            case '5':
                credits();
                exit(0);
        }
    }
}

int sgn(int x)/**signum function*/
{
    //complete
    return (x > 0) ? 1 : ((x < 0) ? -1 : 0);
}

int main()
{
    /**Windows only*/
    system("@ECHO OFF ");
    // system("mode 200, 60 ");//Large screen
    system("mode con: cols=120 lines=50 ");//Small
screen
    system("color 3F ");//Console Colour
    system("title Little Soldiers ");//Console
Title
    /**Windows end*/
    /**Linux only*/
    //system("PS1=$");//preparations for the next
command on Ubuntu 16.04 and above
    //system("PROMPT_COMMAND=\'echo -ne \"\
033]0;Little Soldiers\007\"\'");//Console Title
    /**Linux end*/
    cls;
    welcomeScreen();
    srand((unsigned)time(nullptr));
    loadSettings();
    menu();
    return 0;
}
```

Play.cpp

```
/
*****************************************************
*****************************
* @author Swastik Pal
* @version 1.0
* Date 28/03/2021
* About: The AIengine and Game Mechanism
*****************************************************
*****************************/

//macros here
#define STEPS 4 //must be even
#define PAUSE_BEFORE_NEXT 3 //number of seconds to
wait before refreshing screen

//header files here
#include"header.h"
#include<climits>

//namespace items here

//global variables
Board board;
struct Move
{
    int player;
    Unit* unit;
    char col;
    int row;
} moveSequence[STEPS];

//functions
bool hasEnded()/**Checks if the game has ended and
displays messages accordingly*/
{
    if(!board.hasRemainingUnits(PLAYER_2))
    {
        cls;
        movexy(TOTAL_COL/2 - 7, TOTAL_ROW/2);
        cout<<"Player 1 Wins\n";
        timed_delay(6);//Let the winner celebrate
        return true;
    }
    else if(!board.hasRemainingUnits(PLAYER_1))
    {
        cls;
        movexy(TOTAL_COL/2 - 7, TOTAL_ROW/2);
        cout<<"Player 2 Wins\n";
        timed_delay(6);//Let the winner celebrate
        return true;
    }
    return false;
}

void initialize()
{
    //complete
    board.reset();
    char col;
    int row;
    for(int i=0; i<4; i++)
    {
        cls;
        cout<<"Place your units on the board:\
n(Unit locations as HV; like A1, B6)\n\n";
        board.display();
        cout<<'\n'<<unitNames[i]<<'\n';
        cout<<"Column(A~I):";
        cin>>col;
        if(col<'A' || col>'I')
        {
            --i;
            continue;
        }
        cout<<"Row(7~9):";
        cin>>row;
        if(row<7 || row>9)
        {
            --i;
            continue;
        }
        if(board.isOccupied(col, row))
        {
            cout<<"Occupied\n";
            --i;
            timed_delay(0.3);
            continue;
        }
        board.setPos(PLAYER_1, col, row, i);
        board.setPos(PLAYER_2, col, 10 - row, i);
    }
}

void play()
{
    initialize();
    // int turn = 0;//variable turn for debugging
only
    while(true)
    {
        if(hasEnded())
            return;
        cls;
        // ++turn;
         // cout<<"Turn:"<<turn<<'\n';
        board.display();
        board.showDetails();
        switch(P1control)
        {
            case controller::human:
                manual();
                break;
            case controller::randomAI:
                AIrandom(PLAYER_1);
                break;
            case controller::greedyAI:
                AIgreedy(PLAYER_1);
                break;
            case controller::minimaxAI:
                AIminimax(PLAYER_1);
                break;
        }
        if(hasEnded())
            return;
        cout<<"Waiting for opponent...\n";
        switch(P2control)
        {
            case controller::human:
                manual(PLAYER_2);
                break;
            case controller::randomAI:
                AIrandom();
                break;
            case controller::greedyAI:
                AIgreedy();
                break;
            case controller::minimaxAI:
                AIminimax();
                break;
        }
    }
}

void manual(int player)
{
    //complete
    char col;
    int row;
    while(true)
    {
        cls;
        board.display();
        board.showDetails();
        int ch;
        cout<<"Choose unit:(1, 2, 3, 4)\n";
        cin>>ch;
        if(ch<1 || ch>4)
            continue;
        Unit* unit=board.getUnit(ch-1, player);
        if(unit->isDead)
        {
            cout<<"Cannot choose a dead unit\n";
            timed_delay(0.3);
            continue;
        }
        cout<<"Move to - \n";
        cout<<"Column(A~I):";
        cin>>col;
        cout<<"Row(1~9):";
        cin>>row;
        if(!board.isLegalMove(player, unit, col,
row))
        {
            cout<<"Illegal move";
            timed_delay(0.3);
            continue;
        }
        board.makeMove(player, unit, col, row);
        return;
    }
}

void AIrandom(int player)
{
    //complete
    Unit* unit = nullptr;
    while(true)
    {
        int idx = rand()%4;
        unit = board.getUnit(idx, player);
        if((unit->isDead))
            continue;
        int i;
        char desCol = -1;
        int desRow = -1;
        for(i=unit->getMovement(); i>0; i--)
        {
            if(board.isLegalMove(player, unit,
(desCol=unit->position.col), (desRow=unit->posi-
tion.row+i))) //Move down
            {
                break;
            }
            else if(board.isLegalMove(player,
unit, (desCol=unit->position.col), (desRow=unit-
>position.row-i))) //Move up
            {
                break;
            }
            else if(board.isLegalMove(player,
unit, (desCol=unit->position.col+i), (desRow=unit-
>position.row))) //Move right
            {
                break;
            }
            else if(board.isLegalMove(player,
unit, (desCol=unit->position.col-i), (desRow=unit-
>position.row))) //Move left
            {
                break;
            }
        }
        if(i>0)
        {//Movement possible in atleast one direc-
tion
            while(true)
             {
                 int temp = rand()%4;
                 switch(temp)//Scramble direction
                 {
                     case 0:
```

```
                                desCol = unit->posi-
tion.col;
                                desRow = unit->posi-
tion.row + i;
                                break;
                           case 1:
                                desCol = unit->posi-
tion.col;
                                desRow = unit->posi-
tion.row - i;
                                break;
                           case 2:
                                desCol = unit->posi-
tion.col + i;
                                desRow = unit->posi-
tion.row;
                                break;
                           case 3:
                                desCol = unit->posi-
tion.col - i;
                                desRow = unit->posi-
tion.row;
                                break;
                      }
                      if(board.isLegalMove(player, unit,
desCol, desRow))//checks if movement is possible
                           break;
                 }
            }
            cout<<"Computer moved "<<unit->get-
Name()<<" from "<<unit->position.col<<unit->posi-
tion.row;
            board.makeMove(player, unit, desCol, des-
Row);
            cout<<" to "<<desCol<<desRow<<endl;
            timed_delay(PAUSE_BEFORE_NEXT);
            return;
    }
}
```

```
void AIgreedy(int player)
{
    Unit* unit = nullptr;
    int idx;
    Board savestate;
    savestate.clone(board);
    int saveindex = -1;
    char col = -1;
    int row = -1;
    int maxScore = INT_MIN;
    for(idx=0; idx<4; idx++)
    {
        unit = board.getUnit(idx, player);
        if((unit->isDead))
            continue;
        int score = 0;
        int i;
        for(i=1; i<=unit->getMovement(); i++)
        {
            if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row+i)) //Move
down
            {
                score = board.makeMove(player,
unit, unit->position.col, unit->position.row+i);
                board.clone(savestate);
                if(score>maxScore)
                {
                    maxScore = score;
                    col = unit->position.col;
                    row = unit->position.row + i;
                    saveindex = idx;
                }
            }
            if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row-i)) //Move
up
            {
                score = board.makeMove(player,
unit, unit->position.col, unit->position.row-i);
                board.clone(savestate);
                if(score>maxScore)
                {
                    maxScore = score;
                    col = unit->position.col;
                    row = unit->position.row - i;
                    saveindex = idx;
                }
            }
            if(board.isLegalMove(player, unit,
unit->position.col+i, unit->position.row)) //Move
right
            {
                score = board.makeMove(player,
unit, unit->position.col+i, unit->position.row);
                board.clone(savestate);
                if(score>maxScore)
                {
                    maxScore = score;
                    col = unit->position.col + i;
                    row = unit->position.row;
                    saveindex = idx;
                }
            }
            if(board.isLegalMove(player, unit,
unit->position.col-i, unit->position.row)) //Move
left
            {
                score = board.makeMove(player,
unit, unit->position.col-i, unit->position.row);
                board.clone(savestate);
                if(score>maxScore)
                {
                    maxScore = score;
                    col = unit->position.col - i;
                    row = unit->position.row;
                    saveindex = idx;
                }
            }
        }
    }
    unit = board.getUnit(saveindex, player);
    cout<<"Computer moved "<<unit->getName()<<"
from "<<unit->position.col<<unit->position.row;
    board.makeMove(player, unit, col, row);
    cout<<" to "<<col<<row<<endl;
     timed_delay(PAUSE_BEFORE_NEXT);
    return;
}

double minimax(int step, int player)
{
    /**
    In the Minimax algorithm the two players are
called maximizer and minimizer.
    The maximizer tries to get the highest score
possible while the minimizer tries to do the op-
posite and get the lowest score possible.
    Every board state has a value associated with
it.
    In a given state if the maximizer has upper
hand then, the score of the board will tend to be
some positive value.
    If the minimizer has the upper hand in that
board state then it will tend to be some negative
value.
    The values of the board are calculated by some
heuristics which are unique to every game.
    **/
    //complete
    if(board.getUnit(0, player)->isDead && board.-
getUnit(1, player)->isDead && board.getUnit(2,
player)->isDead && board.getUnit(3, player)->is-
Dead)
    {
        return 0;
    }
    Unit* unit = nullptr;
    int opponent = (player==PLAYER_1)?
PLAYER_2:PLAYER_1;
    int idx;
    Board savestate;
    savestate.clone(board);
    int saveindex = -1;
    char col = -1;
    int row = -1;
    if(step%2==1)/**Maximizer*/
    {
        double maxState = INT_MIN;
        for(idx=0; idx<4; idx++)
        {
            unit = board.getUnit(idx, player);
            if((unit->isDead))
                continue;
            double state = 0;
            int i;
            for(i=1; i<=unit->getMovement(); i++)
            {
                if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row+i)) //Move
down
                {

                    board.makeMove(player, unit,
unit->position.col, unit->position.row+i);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state>maxState)
                    {
                        maxState = state;
                        col = unit->position.col;
                        row = unit->position.row +
i;
                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row-i)) //Move
up
                {

                    board.makeMove(player, unit,
unit->position.col, unit->position.row-i);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state>maxState)
                    {
                        maxState = state;
                        col = unit->position.col;
                        row = unit->position.row -
i;
                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col+i, unit->position.row)) //Move
right
                {
                    board.makeMove(player, unit,
unit->position.col+i, unit->position.row);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state>maxState)
                    {
                        maxState = state;
                        col = unit->position.col +
i;

                        row = unit->position.row;
                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col-i, unit->position.row)) //Move
left
                {
                    board.makeMove(player, unit,
unit->position.col-i, unit->position.row);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state>maxState)
                    {
                        maxState = state;
                        col = unit->position.col -
i;

                        row = unit->position.row;
                        saveindex = idx;
                    }
                }
            }
        }
        unit = board.getUnit(saveindex, player);
        moveSequence[step-1].player = player;
        moveSequence[step-1].unit = unit;
        moveSequence[step-1].col = col;
        moveSequence[step-1].row = row;
        return maxState;
    }
    else/**Minimizer*/
    {
        double minState = INT_MAX;
        for(idx=0; idx<4; idx++)
        {
            unit = board.getUnit(idx, player);
            if((unit->isDead))
                continue;
            double state = 0;
            int i;
            for(i=1; i<=unit->getMovement(); i++)
            {
                if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row+i)) //Move
down
                {
                    board.makeMove(player, unit,
unit->position.col, unit->position.row+i);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state<minState)
                    {
                        minState = state;
                        col = unit->position.col;
                        row = unit->position.row +
i;

                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col, unit->position.row-i)) //Move
up
                {
                    board.makeMove(player, unit,
unit->position.col, unit->position.row-i);
```

```
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state<minState)
                    {
                        minState = state;
                        col = unit->position.col;
                        row = unit->position.row -
i;

                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col+i, unit->position.row)) //Move
right
                {
                    board.makeMove(player, unit,
unit->position.col+i, unit->position.row);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state<minState)
                    {
                        minState = state;
                        col = unit->position.col +
i;

                        row = unit->position.row;
                        saveindex = idx;
                    }
                }
                if(board.isLegalMove(player, unit,
unit->position.col-i, unit->position.row)) //Move
left
```

```
                {
                    board.makeMove(player, unit, 
unit->position.col-i, unit->position.row);
                    state = board.getState();
                    if(step<STEPS && board.hasRe-
mainingUnits(opponent))
                        state = minimax(step+1, op-
ponent);
                    board.clone(savestate);
                    if(state<minState)
                    {
                        minState = state;
                        col = unit->position.col - 
i;

                        row = unit->position.row;
                        saveindex = idx;
                    }
                }
            }
        }
        unit = board.getUnit(saveindex, player);
        moveSequence[step-1].player = player;
        moveSequence[step-1].unit = unit;
        moveSequence[step-1].col = col;
        moveSequence[step-1].row = row;
        return minState;
    }
}

void AIminimax(int player)
{
    //complete
    // double endState = minimax(1, player);// 
variable endState for debugging only
    // cout<<endState<<endl;
    minimax(1, player);
    cout<<"Computer moved "<<moveSequence[0].unit-
>getName()<<" from "<<moveSequence[0].unit->posi-
tion.col<<moveSequence[0].unit->position.row;
```

```
    board.makeMove(moveSequence[0].player,
moveSequence[0].unit, moveSequence[0].col,
moveSequence[0].row);
    cout<<" to
"<<moveSequence[0].col<<moveSequence[0].row<<endl;
     timed_delay(PAUSE_BEFORE_NEXT);
    return;
}

void saveState(string path, Board* state)
{
    ofstream saveFile;
    saveFile.clear();
    saveFile.open(path, std::ios::trunc |
std::ios::binary);
    saveFile.write((char*)state, sizeof(*state));
    saveFile.close();
}

void loadState(string path, Board* state)
{
    ifstream saveFile;
    saveFile.clear();
    saveFile.open(path, std::ios::binary);
    saveFile.read((char*)state, sizeof(*state));
    saveFile.close();
}
```

Board.cpp

```
/
****************************************************
****************************
* @author Swastik Pal
* @version 1.0
* Date 28/03/2021
* About: Board class definition
****************************************************
*****************************/

//macros here
#define WEIGHT_OF_DEATH 1.5

//header files here
#include"header.h"
#include<cmath>

//namespace items here
using std::abs;
using std::ceil;

//functions here
void showUnit(Unit* u)
{
    //complete
    cout<<'\t'<<u->getAttack();
    cout<<'\t'<<u->getDefence();
    cout<<'\t'<<u->getMovement();
    cout<<'\t'<<u->getHealth();
    //cout<<'\t'<<u->getContribution();
}

//class members here
const char Board::HLa-
bels[10]={'X','A','B','C','D','E','F','G','H','I'}
;
const int Board::VLabels[9]={1,2,3,4,5,6,7,8,9};
```

```
Board::Board()/**constructor*/
{
    //complete
    for(int i=0;i<9;i++)
    {
        for(int j=0;j<9;j++)
        {
            playArea[i][j]='.';
        }
    }
    state = 0;
}

void Board::clearPos(char &col, int &row)
{
    //complete
    if(!((col<'A' || col>'I') || (row<1 ||
row>9)))
        playArea[row-1][col-'A']='.';
}

void Board::clone(Board source)
{
    //complete
    //playArea[9][9];
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9; j++)
        {
            playArea[i][j] = source.playArea[i]
[j];
        }
    }
    //player[2];
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<4; j++)
        {
            player[i].unitSet[j].isDead = source.-
player[i].unitSet[j].isDead;
            player[i].unitSet[j].position.col =
source.player[i].unitSet[j].position.col;
            player[i].unitSet[j].position.row =
source.player[i].unitSet[j].position.row;
            player[i].unitSet[j].setContribu-
tion(source.player[i].unitSet[j].getContribution()
);
            player[i].unitSet[j].setHealth(source.-
player[i].unitSet[j].getHealth());
        }
    }
    //state;
    state = source.state;
}

void Board::display()/**shows the entire board*/
{
    //complete
    for(int i=0;i<10;i++)
    {
        cout<<HLabels[i]<<'\t';
    }
    for(int i=0;i<9;i++)
    {
        cout<<"\n\n\n";
        cout<<VLabels[i];
        for(int j=0;j<9;j++)
        {
            cout<<'\t'<<playArea[i][j];
        }
    }

    cout<<endl;
}

double Board::getState()/**positive if PLAYER_2 is
in the lead, negative for PLAYER_1*/
```

```
{
    //complete
    state = 0;
    if(!hasRemainingUnits(PLAYER_1))
    {
        state = 100;//very high value
    }
    else if(!hasRemainingUnits(PLAYER_2))
    {
        state = -100;//very low value
    }
    else
    {
        for(int i=0; i<4; i++)
        {
            state += player[PLAYER_2].unit-
Set[i].getHealth() -
WEIGHT_OF_DEATH*player[PLAYER_2].unitSet[i].is-
Dead;
        }
        for(int i=0; i<4; i++)
        {
            state -= player[PLAYER_1].unit-
Set[i].getHealth() -
WEIGHT_OF_DEATH*player[PLAYER_1].unitSet[i].is-
Dead;
        }
    }
    return state;
}

Unit* Board::getUnit(int unitIndex, int player)
{
    //complete
    return Board::player[player].unitSet[unitIn-
dex].getCurrentInstance();
}

bool Board::hasRemainingUnits(int pNo)
```

```
{
    if(player[pNo].unitSet[0].isDead &&
player[pNo].unitSet[1].isDead && player[pNo].unit-
Set[2].isDead && player[pNo].unitSet[3].isDead)
    {
        return false;
    }
    return true;
}

int Board::isOccupied(char col, int row)/**returns
positive codes for PLAYER_1 and negative for
PLAYER_2*/
{
    //complete
    for(int i=0; i<4; i++)
    {
        if(player[PLAYER_1].unitSet[i].posi-
tion.col==col && player[PLAYER_1].unitSet[i].posi-
tion.row==row)
            return (i+1);
    }
    for(int i=0; i<4; i++)
    {
        if(player[PLAYER_2].unitSet[i].posi-
tion.col==col && player[PLAYER_2].unitSet[i].posi-
tion.row==row)
            return -(i+1);
    }
    return 0;
}

bool Board::isLegalMove(int player, Unit *unit,
char desCol, int desRow)/**checks legality of
move*/
{
    //complete
    int occupancy = isOccupied(desCol, desRow);
    int moveH = desCol - unit->position.col;
    int moveV = desRow - unit->position.row;
    if((occupancy>0 && player==PLAYER_1) || (occu-
pancy<0 && player==PLAYER_2))
        return false;
    if(desCol<'A' || desCol>'I' || desRow<1 ||
desRow>9)
        return false;
    if((moveH != 0 && moveV != 0) || (moveH == 0
&& moveV == 0))
        return false;
    else if(abs(moveH)>unit->getMovement() ||
abs(moveV)>unit->getMovement())
        return false;
    else//checks if road is clear
    {
        int i;
        for(i=sgn(moveH); abs(i)<abs(moveH);
i+=sgn(moveH))
        {
            if(isOccupied(unit->position.col+i,
unit->position.row))
                break;
        }
        if(abs(i)<abs(moveH))
            return false;
        for(i=sgn(moveV); abs(i)<abs(moveV);
i+=sgn(moveV))
        {
            if(isOccupied(unit->position.col,
unit->position.row+i))
                break;
        }
        if(abs(i)<abs(moveV))
            return false;
    }
    return true;
}
```

```
int Board::makeMove(int player, Unit *unit, char
desCol, int desRow)
{
    int score=0;
    int opponent = (player==PLAYER_1)?
PLAYER_2:PLAYER_1;
    int occupancy = isOccupied(desCol, desRow);
    if(occupancy == 0)
    {
        setPos(player, desCol, desRow, unit->get-
Index());
    }
    else
    {
        int moveH = desCol - unit->position.col;
        int moveV = desRow - unit->position.row;
        Unit* occupant = getUnit(abs(occupancy)-1,
opponent);
        int damage = unit->getAttack() - occupant-
>getDefence(); /**damage calculation formula
here*/
        if(damage<0)
           damage = 0;

        if(enableTraits)/**traits implemented
here*/
        {
            int occCur = isOccupied(unit->posi-
tion.col, unit->position.row);
            switch(unit->getIndex())
            {
                case 0://Knight
                    damage += abs(moveH +
moveV);//charge
                    break;
                case 2://Rogue
                    if((isOccupied(desCol+1, des-
Row)==0 || isOccupied(desCol+1, desRow)==occCur)
&& (isOccupied(desCol-1, desRow)==0 || isOccu-
pied(desCol-1, desRow)==occCur) && (isOccu-
pied(desCol, desRow+1)==0 || isOccupied(desCol,
desRow+1)==occCur) && (isOccupied(desCol, desRow-
1)==0 || isOccupied(desCol, desRow-1)==occCur))
                        damage *= 2;
                    break;
                case 3://Swordsman
                    damage = unit->getAttack();//
penetrate
                    break;
            }
        }
        occupant->setHealth(occupant->getHealth()
- damage);
        occupant->setContribution(occupant->get-
Contribution() - damage);
        unit->setContribution(unit->getContribu-
tion() + damage);
        score+=damage;

        if(enableTraits)/**traits implemented
here*/
        {
            switch(occupant->getIndex())
            {
                case 1://Pikeman
                    if(!occupant->isDead)
                        unit->setHealth(unit-
>getHealth() - 1);//counter
                    break;
            }
        }

        if(occupant->getHealth()<=0)
        {
            occupant->setHealth(0);
            occupant->isDead = true;
            setPos(opponent, -1, -1, occupant-
>getIndex());
```

```
            unit->setContribution(unit->getContri-
bution() + damage);
            score+=damage;
        }

        setPos(player, desCol - sgn(moveH), desRow
- sgn(moveV), unit->getIndex());
    }
    return score;
}

void Board::reset()
{
    //complete
    //playArea[9][9];
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9; j++)
        {
            playArea[i][j] = '.';
        }
    }
    //player[2];
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<4; j++)
        {
            player[i].unitSet[j].isDead = false;
            player[i].unitSet[j].position.col = 0;
            player[i].unitSet[j].position.row = 0;
            player[i].unitSet[j].setContribu-
tion(0);
            player[i].unitSet[j].setHealth(5);
        }
    }
    //state;
    state = 0;
}
```

```
void Board::setPos(int pNo, char col, int row, int
index)
{
    clearPos(player[pNo].unitSet[index].posi-
tion.col, player[pNo].unitSet[index].posi-
tion.row);
    player[pNo].unitSet[index].position.col=col;
    player[pNo].unitSet[index].position.row=row;
    if(col>='A' && col<='I' && row>=1 && row<=9)
        playArea[row-1][col-'A']=unitLabels[pNo]
[index];
}

void Board::showDetails()/**shows details of
units*/
{
    //complete
    cout<<"\n\t\tAtk\tDef\tMov\tHP";
    for(int i=0; i<2; i++)
    {
        cout<<"\nPlayer "<<i+1<<":";
        for(int j=0; j<4; j++)
         {
             cout<<'\n'<<(j+1)<<"> "<<unitNames[j];
             if(player[i].unitSet[j].isDead)
                 cout<<"\t\t\t\t\tDEAD";
             else
                 showUnit(player[i].unitSet[j].get-
CurrentInstance());
         }
         cout<<'\n';
    }
    cout<<endl;
}
```

Units.cpp

```
/
***************************************************
****************************
* @author Swastik Pal
* @version 1.0
* Date 28/03/2021
* About: The Units
***************************************************
*****************************/

//macros here

//header files here
#include"header.h"

//functions here

//class members
Unit::Unit(int i)
{
    health = 5;
    contribution = 0;
    index = i;
    isDead = false;
    switch(i)
    {
        case 0:
            attack = 2;
            defence = 2;
            movement = 3;
        break;
        case 1:
            attack = 3;
            defence = 3;
            movement = 1;
        break;
        case 2:
            attack = 4;
            defence = 1;
            movement = 2;
        break;
        case 3:
            attack = 3;
            defence = 2;
            movement = 2;
        break;
    }
}

string unitNames[]={"Knight   ", "Pikeman  ",
"Rogue    ", "Swordsman"};
char unitLabels[2][4]={   {'K', 'P', 'R', 'S'},
                       {'k', 'p', 'r', 's'}};
```

Documentation

Header.h

Field Summary	
Modifier, Type and Field	*Description*
enum controller	Enum with values {human, randomAI, greedyAI, minimaxAI}
int Unit::attack	Stores Unit attack
int Unit::defence	Stores Unit defence
int Unit::movement	Stores Unit movement
int Unit::health	Stores Unit health
int Unit::contribution	Stores Unit contribution. Debug only
int Unit::index	Stores Unit index
bool Unit::isDead	Stores Unit isDead
Pos Unit::position	Stores Unit position
Method Summary	
Modifier, returntype and Method	*Description*
Board* Board::getCurrentInstance()	Returns this instance of Board
Unit* Unit::getCurrentInstance()	Returns this instance of Unit
string Unit::getName(){	Returns Unit name
int Unit::getAttack(){	Returns Unit attack
int Unit::getDefence(){	Returns Unit defence
int Unit::getMovement(){	Returns Unit movement
int Unit::getHealth(){	Returns Unit health
void Unit::setHealth(int hp){	Sets Unit health to hp
int Unit::getIndex(){	Returns Unit index
int Unit::getContribution(){	Returns Unit contribution. Debug only
void Unit::setContribution(int c){	Sets Unit contribution to c. Debug only

`Main.cpp`

Field Summary	
Modifier, Type and Field	*Description*
controller P1control	Stores the contoller for Player 1
controller P2control	Stores the contoller for Player 2
string controllerNames[]	Stores the names of contollers
bool enableTraits	Stores if traits are enabled
Method Summary	
Modifier, returntype and Method	*Description*
void timed_delay(double t)	Makes program wait for t seconds
void movexy(int col, int row)	Moves cursor by specified steps while filling up the screen
void welcomeScreen()	Shows title screen
void help()	Shows help screen
void credits()	Shows credits
void loadSettings()	Loads saved configuration from file
void settings()	Lets the user make changes to the settings
void showCharacterCards()	Every character has its uniqueness. Lets the user see it.
void menu()	Shows the main menu
int sgn(int x)	Signum function
int main()	The main() method. The beginning of everything.

`Play.cpp`

Field Summary	
Modifier, Type and Field	*Description*
Board board	Board object for the game
struct Move moveSequence[]	Stores the optimal path found by the

	minmax algorithm
Method Summary	
Modifier, returntype and Method	*Description*
bool hasEnded()	Checks if the game has ended and displays messages accordingly
void initialize()	Prepares the board for a new game
void play()	Takes care of user interaction during the game
void manual(int player)	Hands over player control to user
void AIrandom(int player)	Randomly chooses any move legal for the player
void AIgreedy(int player)	Makes the move which bring the largest immediate benefit to the player
double minimax(int step, int player)	The minmax algorithm. Calls itself recursively
void AIminimax(int player)	Drives the minimax() function
void saveState(string path, Board* state)	To be implemented in future
void loadState(string path, Board* state)	To be implemented in future

`Board.cpp`

Field Summary	
Modifier, Type and Field	*Description*
const char Board::HLabels[10]	Stores labels displayed on X-axis
const int Board::VLabels[9]	Stores labels displayed on Y-axis
Method Summary	
Modifier, returntype and Method	*Description*
Board::Board()	Default constructor for Board class
void Board::clearPos(char &col, int &row)	Clears the specified location(col, row) on the board
void Board::clone(Board source)	Performs deep copy of Board object from source to the current instance
void Board::display()	Shows the entire board

double Board::getState()	Returns current state of the board. Positive if Player 2 is in the lead, negative for Player 1 and zero if both are equal
Unit* Board::getUnit(int unitIndex, int player)	Returns pointer to the player’s Unit with index unitIndex
bool Board::hasRemainingUnits(int pNo)	Returns false if all Units of player pNo are dead, true otherwise
int Board::isOccupied(char col, int row)	Checks occupancy of position(col, row). Positive for Player 1, negative for Player 2 and zero if unoccupied
bool Board::isLegalMove(int player, Unit* unit, char desCol, int desRow)	Checks if a player is allowed to move unit to location (desCol, desRow)
int Board::makeMove(int player, Unit* unit, char desCol, int desRow)	Moves unit to location (desCol, desRow) on behalf of player and returns possible score (***not guaranteed to be accurate***) symbolising benefit of player from move
void Board::reset()	Initializes board object to default state
void Board::setPos(int pNo, char col, int row, int index)	Allocates location(col, row) to Unit at index of player pNo
void Board::showDetails()	Shows details of all units
void showUnit(Unit* u)	Shows details of Unit u

Units.cpp

Field Summary	
Modifier, Type and Field	*Description*
char unitLabels[][]	Stores the characters representing the player's units.
string unitNames[]	Stores the names of units.
Method Summary	
Modifier, returntype and Method	*Description*
Unit::Unit(int i)	Constructor for unit with index i.

Strengths

- ✔ This project clearly shows that a game need not have console-level graphics. Homemade games can be as enjoyable (and as addictive).
- ✔ Portable sourcecode.
- ✔ This project demonstrates the use of Artificial Intelligence in a game. It implements three different AI algorithms -
  1) An algorithm that randomly chooses any legal move.
  2) A greedy algorithm that makes the move which bring the largest immediate benefit.
  3) The minmax algorithm.
- ✔ Self-explanatory code.

Scope of Improvement

- ✗ This game does not incorporate any graphical elements. A GUI could be added to make it better.
- ✗ The project is minimally documented. Documentation of local variables and macros could be added.

Bibliography

- ◆ Creative Commons (For images)
- ◆ CppReference
- ◆ GeeksForGeeks (GFG)
- ◆ Wikipedia