Alpha-Beta剪枝算法实验报告

一、实验需求

1. 以`Alpha-Beta`剪枝算法作为核心，编写中国象棋博弈程序，实现人机对弈功能，要求界面实现可视化。
2. 棋局评估方法可以参考已有文献，要求具有相应的棋盘界面，可以适当添加开始界面等元素。关于界面编程方面可以参考现成的程序。

二、实验环境

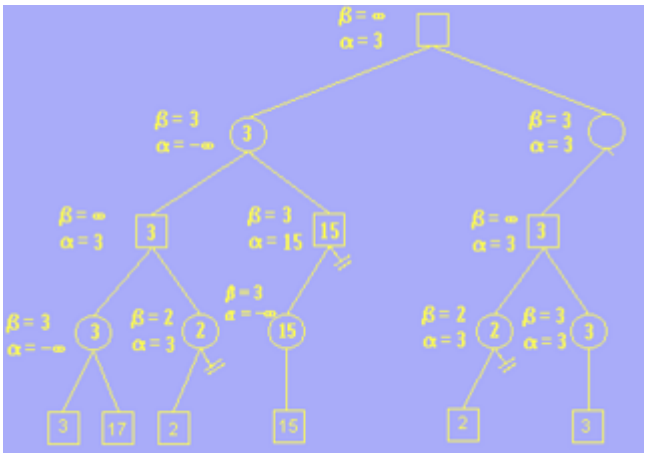
- 操作系统：`Windows10`
- 实现可视化编程的软件：`Qt5.0`
- 编程语言：`C++`

三、实验过程

(1) 象棋逻辑底层实现

编程思想：面向对象编程(OOP)

基本思想：极大极小搜索，与Alpha-Beta剪枝



基本架构：

1. 在象棋博弈的逻辑层代码的编写中，我们首先定义了父类`chess`，之后所有不同种类的棋子类都继承于此父类`chess`，共同拥有其内部的相关属性与方法。不同的子类在对应类定义内部重定义棋子移动合法性函数`judge_move()`，以及相关棋子的种类序号`level`。

   基本父类`chess`代码定义：

```
// Meta chess class
```

```
class chess{
public:
    chess(){
        this->alive = true;
    }
    chess(int level_){
        this->level = level_;
        this->alive = true;
        if(level_ != 0){
            this->id = chess::ID++;
        }else {
            this->id = -1;
        }
    }
    int getLevel(){
        return this->level;
    }
    bool isAlive(){ return alive; }
    void setRowCol(int row_, int col_){
        row = row_;
        col = col_;
    }

    int level;
    bool alive;
    int row, col;
    int id;

    static int ID;

};
```

2. 定义完不同种类棋子的`class`之后，需要构建整体棋盘类`chessBoard`，将棋盘全局操作封装为类方法，同时全局棋盘须存储当前出手方(玩家或AI)，以及随时判断游戏结束与否(gameOver)。

   棋盘类定义：

```
// ChessBoard definition
class chessBoard{
public:
    chessBoard();
    void initBoard();
    chess getChess(pair<int, int> pos);
    int getLevel(pair<int, int> pos);
    bool playerMove(pair<int, int> srcPos, pair<int, int>dstPos);
    bool eatBlackChess(int x, int y);
    // AI Methods
    int AlphaBetaJudge(int level, int score);
    int EstimateValue();
    moveInfo* aiMove();

    // AI moving judge Methods
```

```
    moveInfo* getBestMove();
    void getAllPossibleMove(vector<moveInfo*>& steps);
    int getMaxScore(int level, int score);
    void saveMove(chess& src, chess& target, int row, int col, vector<moveInfo*>& steps);
    // Fake moving and unmoving
    void doMove(moveInfo* info);
    void undoMove(moveInfo* info);

    static bool gameOver;

private:
    vector<vector<chess> > board;
    vector<chess> chessList;
    bool playerSide = true;

};
```

3. `chessBoard` 类拥有二维 `vector` 结构，用于存储棋子在棋盘上的排列信息，`chessList` 记录当前棋子序列(主要用于辨别棋子存活与否状态)。此类内部的核心函数即为对应的 `Alpha-Beta` 剪枝函数 -- `AlphaBetaJudge()`，用于评判AI下一步的局部最优解走步(同时用剪枝减少评判时间)，以及全局棋盘棋力评估函数 `EstimateValue()`，在递归过程中对整体的局面进行评分，将相关分数值返回给极大极小搜索函数 `getMaxScore()` 进行走步选取操作。

重点函数具体实现：

1. `EstimateValue()` -- 局面评估函数

   此处采用的是朴素的双方当前场上的对应棋子棋力总和之差，作为AI实施当前走步的 `好坏` 评判标准。对于每个棋子对应的棋力数值，结合了现成象棋程序的棋力数组定义，分别从 `兵` 到 `将` 的棋力序列为 `{15, 80, 100, 60, 20, 20, 1000}`，发现使用效果还算不错。

   评估函数具体定义：

```
// Simple chess values adding to estimate
int chessBoard::EstimateValue(){
    int redScore = 0, blackScore = 0;
    int chessScore[7] = {15, 80, 100, 60, 20, 20, 1000};
    // Calculate Black values
    for(int i=0; i<16; i++){
        if(!chessList[i].isAlive()) continue;
        blackScore += chessScore[chessList[i].level-1];
    }
    // Calculate Red values
    for(int i=16; i<32; i++){
        if(!chessList[i].isAlive()) continue;
        redScore += chessScore[-chessList[i].level-1];
    }
    return blackScore - redScore;
}
```

2. `AlphaBetaJudge()` -- AlphaBeta剪枝函数

   通过结合极大极小值搜索方法，定义迭代深度为4，每一步迭代采用`getAllPossibleMove()`列出当前AI的所有合法走步，通过`doMove()`函数对每一走步进行模拟行走，求出模拟之后的对应局面评估值，采取剪枝操作去除那些没有必要进行搜索的分支(具体为定义下界alpha，上界beta，所有不符合界限约束的分支均执行剪枝)，然后再执行状态回退`undoMove()`维持原始状态，用于下一步走步模拟。如此迭代过程结束之后，获得的最终解即为AI下一走步的局部最优解。

   剪枝函数定义：

```
int chessBoard::AlphaBetaJudge(int level, int score){
    if(level == 0)
        return EstimateValue();
    vector<moveInfo*> steps;
    getAllPossibleMove(steps);
    // Make Max
    int alpha = 1000000;
    while(steps.size() > 0){
        moveInfo* temp = steps.back();
        steps.pop_back();

        doMove(temp);
        int beta = getMaxScore(level-1, alpha);
        undoMove(temp);
        delete temp;
        // Alpha-Beta Cut
        if(beta <= score){
            while(steps.size() > 0){
                moveInfo* step = steps.back();
                steps.pop_back();
                delete step;
            }
            return beta;
        }
        if(beta < alpha){
            alpha = beta;
        }
    }
    return alpha;
}
```

3. `getAllPossibleMove()` -- 获取当前所有可能的走步

   结合当前棋盘可行棋子情况，采用每个不同种类棋子对应实例的`judge_move()`函数，获取当前合法的走步，并存储到结果中。

   函数定义：

```
// MaxMin-search for all possible moves
```

```
void chessBoard::getAllPossibleMove(vector<moveInfo*>& steps){
    int min, max;
    if(playerSide == true){
        min = 16;
        max = 32;
    }else {
        min = 0;
        max = 16;
    }
    for(int i=min; i<max; i++){
        if(!chessList[i].isAlive())
            continue;
        for(int row=0; row<=9; row++){
            for(int col=0; col<=8; col++){
                int targetID = -1;
                for(int j=0; j<32; j++){
                    if(chessList[j].row == row && chessList[j].col == col &&
chessList[j].isAlive()){
                        targetID = j;
                        break;
                    }
                }
                if(playerSide && targetID >= 16)
                    continue;
                else if(!playerSide && targetID < 16 && targetID >= 0)
                    continue;
                chess target = empty(0);
                if(targetID != -1){
                    target = chessList[targetID];
                }
                saveMove(chessList[i], target, row, col, steps);
            }
        }
    }
}
```

(2) 可视化界面编程

编程思想：面向GUI编程

基本架构：

1. 使用`Qt5.0`进行窗口程序的编写，包括对窗口各项组件的逻辑控制，以及窗口UI的设计。在`Qt`程序编写中，需要通过定义不同的`window`类为对应的`.ui`文件提供各种方法定义，此处新建了`mainwindow`与`gamenwindow`，分别负责象棋初始界面，以及进入之后的棋盘界面。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
```

```
    explicit MainWindow(QWidget *parent = 0);

    void initActions();

    ~MainWindow();

private slots:
    // Tool functions
    void showMainWindow();
    void start();
    void help();
    void exit();

signals:
    void showGameWindow();

private:
    Ui::MainWindow* ui;
    QMediaPlayer *player;

    bool gameStart;
    bool gameOver;

};
```

```
class GameWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit GameWindow(QWidget* parent = 0);

    void initActions();

    void resetBoardChessImg();
    bool isGameOver();
    void setGameOver(bool over);

    ~GameWindow();

protected:
    void mousePressEvent(QMouseEvent* event);

private slots:
    void showGame();
    // Game logic methods
    void aiMove();
    // Chess Board click events
    void clickOnChess(QString a);
    void setButton(int index);
    void blackBtnPress(int index);
```

```
signals:
    void GameOver();

private:
    Ui::GameWindow* ui;
    QSoundEffect player;

    chessBoard* board;
    chess clickTarget;
    // Mappers
    QSignalMapper* mapper;
    QSignalMapper* indexMapper;
    QSignalMapper* blackBtnMapper;
    // UI's buttons
    QPushButton* btn[16];
    QPushButton* blackBtn[16];
    QPushButton* button;
    // Tool variables
    QString image[16];
    pair<int, int> targetPos;
    int targetIndex;
    int targetLevel;

    // Play flags
    bool playerSide;
    int turn;

};
```

2. 此处的`gamewindow`即为象棋可视化程序的主要界面，首先在初始化函数中实例化之前的逻辑层`chessBoard`对象，提供底层的各项函数支持，其次需要在可视化类上为棋子定义点击事件函数`onClick()`，响应用户的下棋操作，并将相应的走步信息(包括目标棋子具体信息，起点与终点坐标)传递给基本棋盘对象`board`进行合法性判断，执行相应的用户走步操作。
3. 当玩家用户执行完操作之后，轮到AI进行当前最优走步的选取，此时UI层执行逻辑层对应的`aiMove()`函数，获取AI需移动的目标棋子与终点坐标，执行走步。除此之外，UI层还需要处理吃棋行为，具体的实现为：实现`eatChessListener`对玩家的点击事件进行监听，判断是否为吃棋操作，若是则需要UI层对图层样式进行更新，同时要将相应的信息传递给底层棋盘对象进行状态更新。

重点函数具体实现：

1. `GameWindow()` -- 基本构造函数，初始化各项基本对象变量。

```
GameWindow::GameWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::GameWindow)
{
    clickTarget = chess(0);
    board = new chessBoard();
    ui->setupUi(this);
```

```
    mapper = new QSignalMapper(this);
    indexMapper = new QSignalMapper(this);
    blackBtnMapper = new QSignalMapper(this);

    initActions();
    playerSide = true;
    turn = 0;
}
```

2. `initAction()` -- 全局初始化函数，为每个棋盘上的棋子挂载点击事件触发与监听器，同时初始化基本棋盘对象。

```
void GameWindow::initActions(){
    board->initBoard();

    int index = 0;
    // Red btns init
    btn[index++] = ui->redRook1;  btn[index++] = ui->redHorse1;
    btn[index++] = ui->redElephant1;  btn[index++] = ui->redGuard1;
    btn[index++] = ui->redGeneral;  btn[index++] = ui->redGuard2;
    btn[index++] = ui->redElephant2;  btn[index++] = ui->redHorse2;
    btn[index++] = ui->redRook2;  btn[index++] = ui->redCannon1;
    btn[index++] = ui->redCannon2;  btn[index++] = ui->redSolider1;
    btn[index++] = ui->redSolider2;  btn[index++] = ui->redSolider3;
    btn[index++] = ui->redSolider4;  btn[index++] = ui->redSolider5;
    // Black btns init
    index = 0;
    blackBtn[index++] = ui->blackRook1;  blackBtn[index++] = ui->blackHorse1;
    blackBtn[index++] = ui->blackElephant1;  blackBtn[index++] = ui->blackGuard1;
    blackBtn[index++] = ui->blackGeneral;  blackBtn[index++] = ui->blackGuard2;
    blackBtn[index++] = ui->blackElephant2;  blackBtn[index++] = ui->blackHorse2;
    blackBtn[index++] = ui->blackRook2;  blackBtn[index++] = ui->blackCannon1;
    blackBtn[index++] = ui->blackCannon2;  blackBtn[index++] = ui->blackSolider1;
    blackBtn[index++] = ui->blackSolider2;  blackBtn[index++] = ui->blackSolider3;
    blackBtn[index++] = ui->blackSolider4;  blackBtn[index++] = ui->blackSolider5;
    for(int i=0; i<16; i++){
        blackBtnMapper->setMapping(blackBtn[i], i);
        connect(blackBtn[i], SIGNAL(clicked()), blackBtnMapper, SLOT(map()));
        connect(blackBtnMapper, SIGNAL(mapped(int)), this, SLOT(blackBtnPress(int)));
    }

    index = 0;
    // Init Red chesses locations
    for(int i=0; i<=8; i++){
        mapper->setMapping(btn[index++], QString::number(9*10 + i));
    }
    mapper->setMapping(btn[index++], QString::number(7*10 + 1));
    mapper->setMapping(btn[index++], QString::number(7*10 + 7));
    for(int i=0; i<=8; i+=2){
        mapper->setMapping(btn[index++], QString::number(6*10 + i));
    }
```

```
    for(int i=0; i<16; i++){
        connect(btn[i], SIGNAL(clicked()), mapper, SLOT(map()));
        connect(btn[i], SIGNAL(clicked()), indexMapper, SLOT(map()));
        indexMapper->setMapping(btn[i], i);
    }
    connect(mapper, SIGNAL(mapped(QString)), this, SLOT(clickOnChess(QString)));
    connect(indexMapper, SIGNAL(mapped(int)), this, SLOT(setButton(int)));
}
```

3. `clickOnChess()` 点击响应函数 -- 为用户点击事件作出响应，同时播放特定选中音效。

```
void GameWindow::clickOnChess(QString a){
    // Judge whether is player turn
    if(!playerSide){
        return;
    }
    qDebug() << a;
    int number = a.toInt();
    int x = number / 10, y = number % 10;
    int level = board->getLevel(make_pair(x, y));
    if(level == 0){
        return;
    }
    player.setSource(QUrl("qrc:/audios/Choose.wav"));
    player.play();
    targetPos = make_pair(x, y);
    targetLevel = level;
    clickTarget = board->getChess(targetPos);
}
```

4. `isGameOver()` 判断游戏是否结束

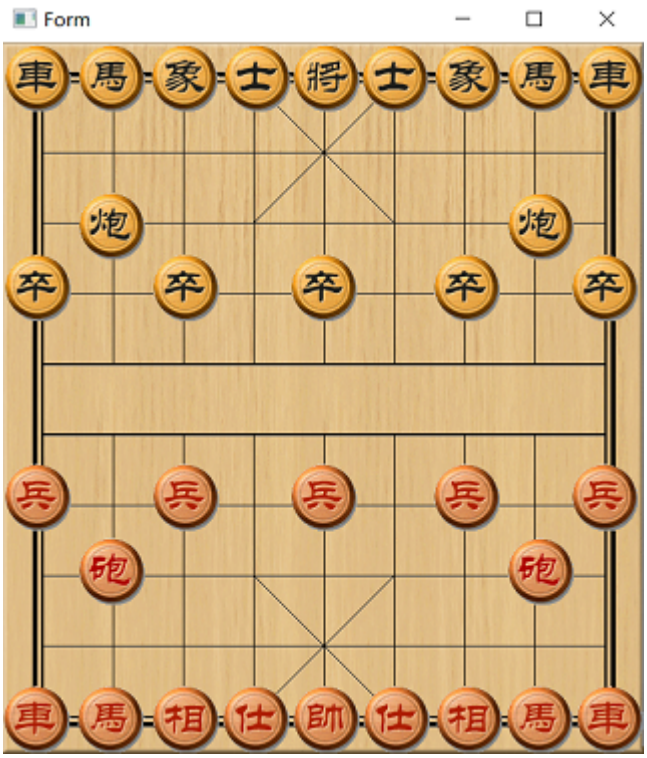
```
bool GameWindow::isGameOver(){
    string result;
    if(playerSide == true){
        result = "游戏结束，玩家获胜";
    }else {
        result = "游戏结束，AI获胜";
    }
    int res = QMessageBox::question(NULL, "Game Over", result.c_str(), QMessageBox::Yes);

    this->close();
}
```

四、实验结果

1. 象棋可视化程序初始界面

2. 象棋程序的游戏主界面



3. 具体测试视频已放在文件夹`tc`下，为`chess.mp4`，可自行查看测试结果。

五、参考文献

1. 人工智能 -- alpha-beta剪枝算法及实践
2. 一看就懂得Alpha-Beta剪枝算法详解
3. 使用Qt开发界面