Report

Team_52
0612255 潘怡汝
0616302 劉承昀
0316205 張立有

A. Description (in Team_52.cpp)

```
struct chess { int pos_X, pos_Y; }
```

記住一顆棋子的位置

```
 →  Y
 |
 |
 X
```

```
void init(vector< vector<int> > board, int label, vector<chess>& c);
```

把和 label 相同的棋子找出來

```
void choose_chess(vector< vector<int> > board, bool is_black,
                  vector<chess> c, vector<chess>& chosen);
```

評估 c 中的棋子並從中選出三個或以上的棋子放入 chosen 以計算 minimax

```
int computeH(vector< vector<int> > board, bool is_black);
```

根據己方和敵方的 minimax 預估做評估

```
bool moveCheck_and_move
 (vector< vector<int> >& board, int label, chess& c, int X_move,
  int Y_move,  vector< vector<int> >& step, vector< vector<bool> >& visitTable);
```

選出的棋子移動後是否合理

```
void moving(vector< vector<int> >& board, bool is_black, chess c,
         vector< vector<int> >& step);
```

選出的棋子如何移動

```
void Max(vector< vector<int> > board, bool is_black,
       vector< vector<int> >& step);
```

預設己方的戰略，並計算 minimax

```
int Min(std::vector< std::vector<int> > board, bool is_black, int a);
```

猜測敵方的戰略

```
std::vector< std::vector<int> > GetStep
    (std::vector< std::vector<int> >& board, bool is_black) {……}
```

回傳選出的棋子的移動方式

B. Design

1. in moving
   (1) if enemies are beside
   → if it is valid which is check by "moveCheck_and_move", then kill it.
   (2) if friends are beside
   → if it is valid which is check by "moveCheck_and_move", then jump over it.
   (3) move one step
   → move one step if it is valid.
2. in choose_chess
   initial priority: 0
   (1) if enemies are beside → has the largest priority
   (2) if already reach the destination and no enemy is beside → has the lowest priority
   (3) if near destination → has the second largest priority
   (4) if friends are beside → piority++
   (5) if enemies are on the diagonal → piority—
   (6) choose the top 3 or more and return
3. in moveCheck_and_move
   (1) if the chess will out of the board → return invalid
   (2) if the chess wil jump to the position where lies chess → return invalid
   (3) else → return true
4. in Max & Min
   → try all the chosen chess and use a heuristic (come from computeH) to update minimax
5. in compute
   → use (1)friends in the goal (2)friends out of goal
   (3)enemies in the goal (4)enemies out of goal to compute heuristic

C. Strategies

1. 以吃掉敵方棋子為主
2. 盡可能向終點移動
3. 用 minimax 尋找較佳解

D. Reasons

1. in choose_chess

   為了排除嘗試所有棋子造成的時間長度問題，我們決定評估各個棋子的prority，選出部分棋子來做嘗試，以減少時間花費。

   評估方式：

   i. 可以吃掉敵方棋子 → 既可減少敵方棋子又可前進，故具有最大優先權

   ii. 差一步即可抵達終點 → 擁有第二優先權

   iii. 非 i 和 ii，但有己方棋子在四周可以進行 hop

   → 可以移動比較多，所以 priority++

   iv. 對角線有敵方棋子 → 移動一格有可能被吃掉，所以 priority--

   v. 如果已經抵達終點且無敵方棋子可吃

   → 因已經抵達終點，所以優先權最低

   最後根據每個棋子的 priority 值，選出最大的三個，如果有值相同的也會選，所以可能會選出超過三個棋子來做嘗試

2. in moving

   為了排除嘗試一顆棋子移動的所有可能產生的大量時間花費，所以對於棋子的每一個移動，我們會經過評估選擇出一個我們認為最好的方向。

   選擇方法：

   i. 可以吃掉對方棋子✓

   ii. 非 i，但有己方棋子在四周(會撇除後退方向)✓

   iii. 非 i、ii 且 step.size()是 1(代表接下來是第一步)

   → 選一個合理方向移動✓

3. in computeH

   根據兩方棋子數量和抵達終點的棋子數量來計算。

   計算原則：

   i. 抵達終點才算成績，所以抵達終點的棋子數權重最大

   ii. 存活數量愈多、棋子愈靠近終點愈具有優勢，

   所以愈靠近終點的棋子權重愈大

   以黑棋為例(6、7 是終點)(y_position, weight)

   →(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)

   白棋同理

   iii. 將抵達終點棋子數量和棋子數量依權重相加，

   以己方總和減去敵方總和計算，結果即為 heuristic

E. Weak point

1. 以吃掉敵方棋子為優先目標

   → 導致對手因棋子少可較快抵達終點，而己方棋子數量雖多，但抵達數少

   → lose

2. 沒有 trace 所有方法，並且決定每一步時只考慮當前狀況而忽略後面其他可能
   → 可能有可以吃掉較多子或者向前走較多步的路徑被忽略
3. 選擇方向時以前進為主，但無法前進時一律都是往下優先於往上
   → 缺乏隨機，容易忽略一部分可能性
4. minimax 只做了兩層

```
#include "STcpClient.h"

#include <iostream>

using namespace std;

struct chess {
        //where my chess is
        int  pos_X, pos_Y;
        //   ---->Y
        //  |
        //  X
};

/*Store the position of the chess with label.*/
void init(vector< vector<int> > board, int label, vector<chess>& c);
/*Choose some chess from c by computing their priority.*/
void choose_chess(vector< vector<int> > board, bool is_black,
        vector<chess> c, vector<chess>& chosen);
/*Compute heuristic.*/
int computeH(vector< vector<int> > board, bool is_black);
/*Check the movement. If legal, move the chess. (Single movement)*/
bool moveCheck_and_move(vector< vector<int> >& board, int label,
        chess& c, int X_move, int Y_move, vector< vector<int> >& step,
        vector< vector<bool> >& visitTable);
/*Move the chess.*/
void moving(vector< vector<int> >& board, bool is_black, chess c,
        vector< vector<int> >& step);
/*Max in minimax.*/
void Max(vector< vector<int> > board, bool is_black,
        vector< vector<int> >& step);
/*Min in minimax*/
int Min(std::vector< std::vector<int> > board, bool is_black, int a);

void init(vector< vector<int> > board, int label, vector<chess>& c)
{
        for (int i = 0; i<8; i++) {
                for (int j = 0; j<8; j++) {
                        chess NC;
                        if (board[i][j] == label) {
                                NC.pos_X = i;
                                NC.pos_Y = j;
                                c.push_back(NC);
                        }
                }
        }
        return;
}

void choose_chess(vector< vector<int> > board, bool is_black,
        vector<chess> c, vector<chess>& chosen)
{
        int enemy = (is_black) ? 2 : 1;
        int max = -10;
        int p[9] = { 0 };
        for (int i = 0; i < c.size(); i++) {
```

```
        //if the chess next to the enemy, give the largest priority
        if ((c[i].pos_X < 7 && c[i].pos_X>0) && (c[i].pos_Y < 7 && c[i].pos_Y>0) &&
            (board[c[i].pos_X + 1][c[i].pos_Y] == enemy ||
                board[c[i].pos_X - 1][c[i].pos_Y] == enemy ||
                is_black && board[c[i].pos_X][c[i].pos_Y + 1] == enemy ||
                !is_black && board[c[i].pos_X][c[i].pos_Y - 1] == enemy))
        {
            chosen.push_back(c[i]);
            continue;
        }

        //If the chess is in destination, give the lowest priority
        if ((is_black && c[i].pos_Y > 5) || (!is_black && c[i].pos_Y < 2))
        {
            p[i] = -7;
            continue;
        }

        //if the chess is at y == 2 or y == 6, give the second largest priority
        if ((is_black && c[i].pos_Y == 5 && board[c[i].pos_X][c[i].pos_Y + 1] == 0) ||
            (!is_black && c[i].pos_Y == 2 && board[c[i].pos_X][c[i].pos_Y - 1] ==
0))
        {
            p[i] = 50;
            continue;
        }

        //if a chess have some chess next to (on its right, left, above, or botton)it,
        //than increase the piority
        if (c[i].pos_X != 7 && board[c[i].pos_X + 1][c[i].pos_Y] != 0) p[i]++;
        if (c[i].pos_X != 0 && board[c[i].pos_X - 1][c[i].pos_Y] != 0) p[i]++;
        if (is_black && c[i].pos_Y != 7 && board[c[i].pos_X][c[i].pos_Y + 1] != 0)
p[i]++;
        if (!is_black && c[i].pos_Y != 0 && board[c[i].pos_X][c[i].pos_Y - 1] != 0)
p[i]++;

        //if a chess have emeny(black == 1 --> emeny == 2) chess on the diagonal
        //lower the piority
        if (c[i].pos_X != 7 && c[i].pos_Y != 7 && board[c[i].pos_X + 1][c[i].pos_Y + 1]
== enemy) p[i]--;
        if (c[i].pos_X != 7 && c[i].pos_Y != 0 && board[c[i].pos_X + 1][c[i].pos_Y - 1]
== enemy) p[i]--;
        if (c[i].pos_X != 0 && c[i].pos_Y != 7 && board[c[i].pos_X - 1][c[i].pos_Y + 1]
== enemy) p[i]--;
        if (c[i].pos_X != 0 && c[i].pos_Y != 0 && board[c[i].pos_X - 1][c[i].pos_Y - 1]
== enemy) p[i]--;

        if (p[i] > max)max = p[i];
    }

    if (chosen.size() > 2) return;

    for (int i = 0; i < 9; ++i)
    {
        if(p[i]==50)    chosen.push_back(c[i]);
    }
```

```
	if (chosen.size() > 2) return;

	/*Choose chess which doesn't have the largest or the second largest priority.*/
	while (chosen.size() < 3 && chosen.size() < c.size())
	{
		vector<int> pool;
		for (int i = 0; i < 9; ++i)
		{
			if (p[i] == max && i < c.size())
			{
				chosen.push_back(c[i]);
				p[i] = -10;
			}
		}
		max = -10;
		for (int i = 0; i < 9; ++i)
			if (p[i] > max && i < c.size())		max = p[i];
	}
	return;
}

int computeH(vector< vector<int> > board, bool is_black)
{
	int numB = 0, numW = 0, numBD = 0, numWD = 0;

	/*Record the number of black chess, black chess in destination,
	  white chess, and white chess in destination.*/
	for (int i = 0; i<8; i++) {
		for (int j = 0; j<2; j++) {
			if (board[i][j] == 1) numB += j + 1;
			if (board[i][j] == 2) numWD++;
		}
		for (int j = 2; j<6; j++) {
			if (board[i][j] == 1) numB += j + 1;
			if (board[i][j] == 2) numW += 8 - j;
		}
		for (int j = 6; j<8; j++) {
			if (board[i][j] == 1) numBD++;
			if (board[i][j] == 2) numW += 8 - j;
		}
	}

	//compute H
	int H;
	if (is_black) H = 2 * numB + 15 * numBD - numW - 15 * numWD;
	else H = 2 * numW + 15 * numWD - numB - 15 * numBD;

	return H;
}

bool moveCheck_and_move(vector< vector<int> >& board, int label,
	chess& c, int X_move,  int Y_move, vector< vector<int> >& step,
	vector< vector<bool> >& visitTable)
{
	int nX = c.pos_X + 2 * X_move;
	int nY = c.pos_Y + 2 * Y_move;
	if (nX > 7 || nX < 0 || nY > 7 || nY < 0) return false;

	//If legal, move it.
	if (board[c.pos_X + X_move][c.pos_Y + Y_move] == label &&
		board[nX][nY] == 0 && !visitTable[nX][nY]) {
		vector<int> each_step;
		each_step.push_back(nX);
		each_step.push_back(nY);
		step.push_back(each_step);
		visitTable[nX][nY] = 1;
		board[nX][nY] = board[c.pos_X][c.pos_Y];
		board[c.pos_X][c.pos_Y] = 0;
		if (label != board[nX][nY])
			board[c.pos_X + X_move][c.pos_Y + Y_move] = 0;
		c.pos_X = nX;
		c.pos_Y = nY;
		return true;
	};
	return false;
}

void moving(vector< vector<int> >& board, bool is_black, chess c,
	vector< vector<int> >& step)
{
	int enemy = (is_black) ? 2 : 1;
	int partner = (is_black) ? 1 : 2;
	vector<int> each_step;
	each_step.push_back(c.pos_X);
	each_step.push_back(c.pos_Y);
	step.push_back(each_step);
	vector< vector<bool> > visitTable;
	for (int i = 0; i < 8; ++i)
	{
		vector<bool> r(8, 0);
		visitTable.push_back(r);
	}
	visitTable[c.pos_X][c.pos_Y] = 1;

	//search
	int iter = 0;
	while (iter < 50) {
		iter++;

		if (c.pos_X > 7 || c.pos_X < 0 ||
			c.pos_Y > 7 || c.pos_Y < 0) return;

		//find way - 1 enemy beside
		if (is_black)
		{
			if (moveCheck_and_move(board, enemy, c, 0, 1, step, visitTable))
				continue;
		}
		else {
			if (moveCheck_and_move(board, enemy, c, 0, -1, step, visitTable))
				continue;
```

```
        }
        if (moveCheck_and_move(board, enemy, c, 1, 0, step, visitTable))
            continue;
        if (moveCheck_and_move(board, enemy, c, -1, 0, step, visitTable))
            continue;

        //find way - 2 friend beside
        if (is_black)
        {
            if (moveCheck_and_move(board, partner, c, 0, 1, step, visitTable))
                continue;
        }
        else {
            if (moveCheck_and_move(board, partner, c, 0, -1, step, visitTable))
                continue;
        }
        if (moveCheck_and_move(board, partner, c, 1, 0, step, visitTable))
            continue;
        if (moveCheck_and_move(board, partner, c, -1, 0, step, visitTable))
            continue;

        //find way - 3 forward
        //if has jumped in case 1 or 2, won't into case 3
        if (is_black &&step.size() == 1 && c.pos_Y != 7 && board[c.pos_X][c.pos_Y + 1]
== 0) {
            each_step.clear();
            each_step.push_back(c.pos_X);
            each_step.push_back(c.pos_Y + 1);
            step.push_back(each_step);
            board[c.pos_X][c.pos_Y + 1] = board[c.pos_X][c.pos_Y];
            board[c.pos_X][c.pos_Y] = 0;
            c.pos_Y += 1;
        }
        else if (!is_black && step.size() == 1 && c.pos_Y != 0 &&
board[c.pos_X][c.pos_Y - 1] == 0) {
            each_step.clear();
            each_step.push_back(c.pos_X);
            each_step.push_back(c.pos_Y - 1);
            step.push_back(each_step);
            board[c.pos_X][c.pos_Y - 1] = board[c.pos_X][c.pos_Y];
            board[c.pos_X][c.pos_Y] = 0;
            c.pos_Y -= 1;
        }
        else if (step.size() == 1 && c.pos_X != 7 && board[c.pos_X + 1][c.pos_Y] == 0)
{
            each_step.clear();
            each_step.push_back(c.pos_X + 1);
            each_step.push_back(c.pos_Y);
            step.push_back(each_step);
            board[c.pos_X + 1][c.pos_Y] = board[c.pos_X][c.pos_Y];
            board[c.pos_X][c.pos_Y] = 0;
            c.pos_X += 1;
        }
        else if (step.size() == 1 && c.pos_X != 0 && board[c.pos_X - 1][c.pos_Y] == 0)
{
            each_step.clear();
```

```
			each_step.push_back(c.pos_X - 1);
			each_step.push_back(c.pos_Y);
			step.push_back(each_step);
			board[c.pos_X - 1][c.pos_Y] = board[c.pos_X][c.pos_Y];
			board[c.pos_X][c.pos_Y] = 0;
			c.pos_X -= 1;
		}

		//none of above
		return;
	}
	return;
}

void Max(vector< vector<int> > board, bool is_black,
	vector< vector<int> >& step)
{
	int enemy = (is_black) ? 2 : 1;
	int partner = (is_black) ? 1 : 2;

	vector<chess> mine, chosen;
	init(board, partner, mine);
	choose_chess(board, is_black, mine, chosen);
	int max = -9999;

	//Try all the chosen chess and choose best
	for (int i = 0; i < chosen.size(); ++i)
	{
		vector< vector<int> > tempBoard = board;
		vector< vector<int> > tempStep;
		moving(tempBoard, is_black, chosen[i], tempStep);
		if (tempStep.size() == 1) continue;
		int temp = Min(tempBoard, is_black, max);
		if (temp > max && temp != 99999)
		{
			max = temp;
			step = tempStep;
		}
	}

	return;
}

int Min(std::vector< std::vector<int> > board, bool is_black, int a) {
	int partner = (is_black) ? 2 : 1;

	vector<chess> mine, chosen;
	init(board, partner, mine);
	choose_chess(board, is_black, mine, chosen);
	int min = 99999;

	//Try all the chosen chess and choose best
	if (chosen.empty()) min = computeH(board, is_black);
	for (int i = 0; i < chosen.size(); ++i)
	{
		vector< vector<int> > tempBoard = board;
```

```
                vector< vector<int> > step;
                moving(tempBoard, !is_black, chosen[i], step);
                int temp = computeH(tempBoard, is_black);
                if (temp < min)
                        min = temp;
                if (min < a)
                        return -9999;
        }
        return min;
}

std::vector< std::vector<int> > GetStep(std::vector< std::vector<int> >& board, bool is_black)
{
        std::vector< std::vector<int> > step;

        Max(board, is_black, step);

        return step;
}

int main() {
        int id_package;
        std::vector< std::vector<int> > board, step;
        bool is_black;
        while (true) {
                if (GetBoard(id_package, board, is_black))
                        break;

                step = GetStep(board, is_black);
                SendStep(id_package, step);
        }
}
```