學號 : １１０２０６５

姓名：游竣捷

111程式設計(一)

期末專題

五行棋對抗程式

說明文件

目錄

* [摘要 3](#摘要)
* [系統說明 4](#系統說明)
  + [程式流程圖 4](#程式流程圖)
  + [讀取傳入參數 5](#讀取傳入參數)
  + [找出所有棋子的起始位置 6](#找出所有棋子的起始位置)
  + [路徑搜索 7](#路徑搜索)
  + [找出危險座標 11](#找出危險座標)
  + [決定移哪一子 13](#決定移哪一子)
  + [檢查棋子 14](#檢查棋子)
  + [找出安全座標 15](#找出安全座標)
  + [決定移動路徑 17](#決定移動路徑)
  + [移動到目標位置 18](#移動到目標位置)
  + [保存路徑 19](#保存路徑)
* [執行與測試 20](#執行與測試)
  + [執行 20](#執行)
    - [執行結果 20](#執行執行結果)
    - [執行流程 21](#執行執行流程)
  + [測試(一) 吃掉對方棋子 25](#測試一吃掉對方棋子)
    - [執行結果 25](#測試一執行結果)
    - [執行流程 26](#測試一執行流程)
  + [測試(二) 移動到安全座標 30](#測試二移動到安全座標)
    - [執行結果 30](#測試二執行結果)
    - [執行流程 31](#測試二執行流程)
* [未完成(需改進) 35](#未完成需改進)
  + [阻礙搜索(新增時間2023/01/09) 35](#阻礙搜索)
    - [執行結果 35](#阻礙搜索測試結果)
    - [說明 35](#阻礙搜索說明)
    - [改進方法 36](#阻礙搜索改進方法)

摘要

此程式為五行棋對抗程式

棋盤大小為5x5

起始盤面如下

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  (0,0) | 0  (1,0) | 0  (2,0) | 0  (3,0) | 2  (4,0) |
| 0  (0,1) | 0  (1,1) | 0  (2,1) | 0  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 0  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 0  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 2  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 1  (4,4) |

規則：

每人有兩顆子，雙方輪流執子，每次移動一子，可移動一到五步，每步一格，經過的位置不能重複且只能經過沒有棋子的位置，只有在第五步時，才能吃下對手的棋子。

判定：

沒有位置可下與沒子可下判輸，移動不符規定視為違規。違規包括逾時(1秒內未落子)、移動違規、程式包含病毒與惡意程式碼與刪除檔案等惡意行為。

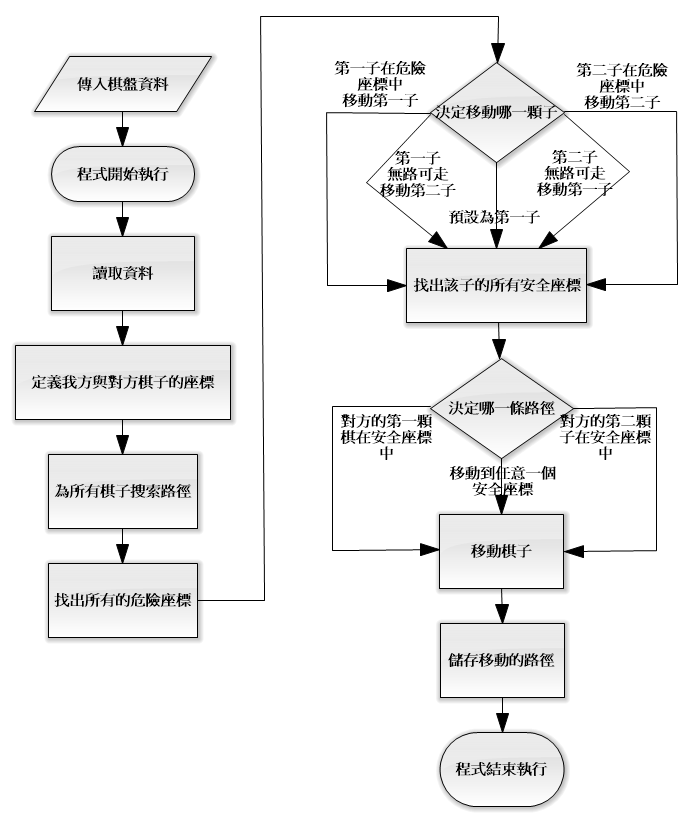
**內容摘要:**

透過分析場上每一顆棋子的資訊，決定出要移動哪一顆棋子、要移動到哪裡並且棋子所移動的路徑會被記錄下來。

此程式是透過先對場上的每一顆棋子，包含自己的棋子與敵方的棋子，進行路徑搜索並儲存到鏈結串列的四元樹中，透過分析對手的路徑得出對手在第五步的棋的座標，再與自己的棋子的路徑交叉比對，即可得出安全的座標。

此程式會不斷移動到安全座標，直到對手的棋移動到我方棋子可以吃的座標，就算對手會反吃也沒關係，因為只要對手只剩一顆棋子，我方的棋子就不會被吃光。

系統說明

程式流程圖

讀取傳入參數

傳入程式參數 共有28個

argv[0] 為自己的程式碼

argv[1] 為對手的學號

argv[2] 為先後手(1或2)

argv[3] ~ argv[27]為棋盤的資料

首先透過以上傳入程式的參數，將資料存入以下變數

char other[10] //對手的學號10碼

int whoami //我方是誰 假設是1代表我方是先手

int whoisother //對方是誰 假設我方是1 對手即是2

int A[H][W] //將argv[3]~argv[27]共25格棋盤資料存入二維陣列。

註:H與W已透過 #define給值，皆為5因為棋盤為5x5。

此時二維陣列，就像是一個棋盤了

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  (0,0)  A[0][0] | 0  (1,0)  A[0][1] | 0  (2,0)  A[0][2] | 0  (3,0)  A[0][3] | 2  (4,0)  A[0][4] |
| 0  (0,1)  A[1][0] | 0  (1,1)  A[1][1] | 0  (2,1)  A[1][2] | 0  (3,1)  A[1][3] | 0  (4,1)  A[1][4] |
| 0  (0,2)  A[2][0] | 0  (1,2)  A[2][1] | 0  (2,2)  A[2][2] | 0  (3,2)  A[2][3] | 0  (4,2)  A[2][4] |
| 0  (0,3)  A[3][0] | 0  (1,3)  A[3][1] | 0  (2,3)  A[3][2] | 0  (3,3) A[3][3] | 0  (4,3)  A[3][4] |
| 2  (0,4)  A[4][0] | 0  (1,4)  A[4][1] | 0  (2,4)  A[4][2] | 0  (3,4)  A[4][3] | 1  (4,4)  A[4][4] |

找出所有棋子的起始位置

首先，先決定所有棋子的位置，包含我方的棋與對手的棋，使用以下的變數來儲存。

struct Position self\_position[2] = { 0 }; //我方棋的位置

struct Position other\_position[2] = { 0 }; //敵方棋的位置

int self\_n = 0; //我方棋的數量

int other\_n = 0; //敵方棋的數量

使用void determinePosition ()來找出以上資訊。

所需參數為:

int A[H][W], int whoami, int whoisother,

struct Position self\_position[2],

struct Position other\_position[2],

int& self\_n, int& other\_n

透過雙重迴圈掃描整個棋盤，

self\_n = 0;

other\_n = 0;

int i, j;

for (i = 0; i < H; i++) { //i值為0~4

for (j = 0; j < W; j++) //j值為0~4

{

if (A[i][j] == whoami) //當A[i][j] == 我方棋子

{

self\_position[self\_n].y = i; //我方棋的y坐標即等於i

self\_position[self\_n].x = j; //我方棋的x坐標即等於j

self\_n++; //我方棋的數量加1

}

if (A[i][j] == whoisother) //當A[i][j] == 敵方棋子

{

other\_position[other\_n].y = i; //敵方棋的y坐標即等於i

other\_position[other\_n].x = j; //敵方棋的x坐標即等於j

other\_n++; //敵方棋的數量加1

}

}

}

路徑搜索

利用結構搭配指標使用鏈結串列做出四元樹

struct Position

{

int x; //該路徑節點的 x座標

int y; //該路徑節點的 y座標

int step; //走到該路徑節點所耗費的步數

struct Position\* father; //該路徑節點上一個節點是誰(父節點)

struct Position\* Up; //該路徑節點的上面是誰

struct Position\* Down; //該路徑節點的下面是誰

struct Position\* Left; //該路徑節點的左邊是誰

struct Position\* Right; //該路徑節點的右邊是誰

};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  (0,0) | 0  (1,0) | 0  (2,0) | 0  (3,0) | 2  (4,0) |
| 0  (0,1) | 0  (1,1) | 0  (2,1) | 0  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 0  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 0  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 2  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 1  (4,4) |

例如: 我方是 1，現在要從左上(0,0)移動到正中間(2,2)，

假設路徑為 (0,0)->(0,1)->(0,2)->(1,2)->(2,2) 共移動四步，

此路徑節點在(2,2)的資訊為

x = 2，y = 2,

step = 4,

father = (1,2)

\*Up = (2,1)

\*Down = (2,3)

\*Right = (3,2)

注意因為不能走回頭路所以\*Left = NULL 不等於 (1,2)

使用以下變數，儲存每顆棋的路徑，預設值為NULL。

struct Position\* self1\_path = new struct Position;

struct Position\* self2\_path = new struct Position;

struct Position\* other1\_path = new struct Position;

struct Position\* other2\_path = new struct Position;

self1\_path = self2\_path = other1\_path = other2\_path = NULL;

if (self\_n == 2) //如果我方棋數量為2

{

//為我方第一顆棋做路經搜索

self1\_path = searchPath(self1\_path, A, self\_position[0].x,

self\_position[0].y, whoami, whoisother);

//為我方第二顆棋做路經搜索

self2\_path = searchPath(self2\_path, A, self\_position[1].x,

self\_position[1].y, whoami, whoisother);

}

else 否則

{

//只為我方第一顆棋做路經搜索

self1\_path = searchPath(self1\_path, A, self\_position[0].x,

self\_position[0].y, whoami, whoisother);

}

if (other\_n == 2) //如果敵方棋數量為2

{

//為敵方第一顆棋做路經搜索

other1\_path = searchPath(other1\_path, A, other\_position[0].x,

other\_position[0].y, whoisother, whoami);

//為敵方第二顆棋做路經搜索

other2\_path = searchPath(other2\_path, A, other\_position[1].x,

other\_position[1].y, whoisother, whoami);

}

else 否則

{

//只為敵方第一顆棋做路經搜索

other1\_path = searchPath(other1\_path, A, other\_position[0].x,

other\_position[0].y, whoisother, whoami);

}

使用struct Position\* searchPath()搜索路徑,

所需參數

struct Position\* path //搜索過後的路徑都為儲存在內。

int A[H][W] //目前棋盤資料

int x //路徑起始點的x座標

int y //路徑起始點的y座標

int who //自己是誰(1或2)

int other //敵方是誰(1或2)

int step = 0 //走到該路徑時，所要花費的步數

struct Position\* father = NULL //該路徑的父節點是誰

首先先將棋盤A[H][W] 存入，變數int map[H][W];

避免在路徑搜索時，更改到原先棋盤資料。

**涵式內容:**

當步數為0的時候，初始化path。

if (step == 0) {

//動態記憶體宣告新的路徑

path = NewPath(x, y, step, father);

//因為 0,1,2都被用掉了所以用step+3來改值，避免走回頭路。

map[y][x] = step + 3;

if (y > 0) { //如果y>0代表上面不是邊界可以往上搜索

path->Up = searchPath(path->Up, map, x, y - 1, who, other, step + 1, path);

}

if (y < H-1) { //如果y<4代表下面不是邊界可以往下搜索

path->Down = searchPath(path->Down, map, x, y + 1, who, other, step + 1, path);

}

if (x > 0) { //如果x>0代表左邊不是邊界可以往左搜索

path->Left = searchPath(path->Left, map, x - 1, y, who, other, step + 1, path);

}

if (x < W-1) { //如果x<4代表右邊不是邊界可以往右搜索

path->Right = searchPath(path->Right, map, x + 1, y, who, other, step + 1, path);

}

}

step\_limit 透過 #define給值 = 5

當步數 <= 步數限制時執行。

else if (step <= step\_limit) {

//如果這個座標 == 0代表這個位置是空的可以走

if (map[y][x] == 0) {

//動態記憶體宣告新的路徑

path = NewPath(x, y, step, father);

//改值避免走回頭路。

map[y][x] = step + 3;

if (y > 0) { //如果y>0代表上面不是邊界可以往上搜索

path->Up = searchPath(path->Up, map, x, y - 1, who,

other, step + 1, path);

}

if (y < H-1) { //如果y<4代表下面不是邊界可以往下搜索

path->Down = searchPath(path->Down, map, x, y + 1, who,

other, step + 1, path);

}

if (x > 0) { //如果x>0代表左邊不是邊界可以往左搜索

path->Left = searchPath(path->Left, map, x - 1, y, who,

other, step + 1, path);

}

if (x < W-1) { //如果x<4代表右邊不是邊界可以往右搜索

path->Right = searchPath(path->Right, map, x + 1, y, who,

other, step + 1, path);

}

}

//如果這個座標 == 是對手且步數等於第五步，可以吃掉對手。

else if (map[y][x] == other && step == step\_limit) {

//動態記憶體宣告新的路徑

path = NewPath(x, y, step, father);

//因為已經到第五步(步數上限)所以就不用在往下搜索

//但map一樣改值，方便show出棋盤查看路徑

map[y][x] = step + 3;

}

}

找出危險座標

// 用於儲存所有危險座標

// 因為棋盤大小為5\*5，因此座標不可能超過25個

struct Position DangerPosition[25] = { 0 };

//用於儲存危險座標共有幾個

int DangerPosition\_n = 0;

// 如果手有兩顆子

if (other\_n == 2) {

將這兩顆子在第五步的路徑存入危險座標，並會回傳危險座標的數量

DangerPosition\_n = FindDangerPositions(other1\_path,

DangerPosition);

DangerPosition\_n = FindDangerPositions(other2\_path,

DangerPosition);

}

else {

DangerPosition\_n = FindDangerPositions(other1\_path,

DangerPosition);

}

使用int FindDangerPositions()找到所有危險的座標

所需參數:

struct Position\* other\_path //對手的路徑

struct Position DangerPosition[] //用於儲存危險座標

**涵式內容:**

static int i = 0; //靜態變數用於計算危險座標的數量

if (other\_path == NULL) { //如果路徑不通就回傳值中斷

return i;

}

//如果路徑的步數等於第五步 代表我方棋子移動到這些位置會被對方吃掉。

//因此這些座標即是危險座標。

if (other\_path->step == step\_limit) {

//檢查危險座標是否重複，因為不同路徑下可以抵達的危險座標可能相同

if (!checkInside(other\_path, DangerPosition, i)) {

//將資料儲存到 DangerPosition[]中。

DangerPosition[i].x = other\_path->x;

DangerPosition[i].y = other\_path->y;

DangerPosition[i].step = other\_path->step;

i++; //危險座標數量加1。

}

}

//向 上、下、左、右 搜索路徑

i = FindDangerPositions(other\_path->Up, DangerPosition);

i = FindDangerPositions(other\_path->Down, DangerPosition);

i = FindDangerPositions(other\_path->Left, DangerPosition);

i = FindDangerPositions(other\_path->Right, DangerPosition);

return i; //回傳危險座標的總共數量。

決定移哪一子

struct Position\* chess = new struct Position; //要移動的棋子

//如果我方第一顆棋子在危險座標中，第一顆棋就是要移動的棋子。

if (checkInside(self1\_path, DangerPosition, DangerPosition\_n)) {

chess = self1\_path;

}

//如果我方第二顆棋子在危險座標中，第二顆棋就是要移動的棋子。

else if (checkInside(self2\_path, DangerPosition, DangerPosition\_n)) {

chess = self2\_path;

}

//如果我方第一顆棋子被包圍住，第二顆棋就是要移動的棋子。

//注意需在第二顆棋存活的情況下才能執行。

else if (checkSurround(self1\_path) && self2\_path != NULL) {

chess = self2\_path;

}

//如果我方第二顆棋子被包圍住，第一顆棋就是要移動的棋子。

else if (checkSurround(self2\_path)) {

chess = self1\_path;

}

//預設移動的棋子為第一顆棋

else {

chess = self1\_path;

}

檢查棋子

透過bool checkInside()檢查棋子座標是否在一陣列當中

所需參數:

struct Position\* chess //要檢查的棋子

struct Position Position[] //要檢查的陣列

int n //陣列的大小

**涵式內容:**

bool checkInside(struct Position\* chess, struct Position Position[],  
 int n)

{

//如果棋子不存在於棋盤上(即為NULL)回傳false

if (chess == NULL) {

return false;

}

//透過迴圈比對是否在陣列中

for (int i = 0; i < n; i++) {

if (chess->x == Position[i].x && chess->y == Position[i].y) {

return true;

}

}

return false;

}

找出安全座標

道理與找危險座標相同，只不過透過危險座標就能反推出安全座標。

struct Position SafePosition[25] = { 0 };

int SafePosition\_n = 0;

SafePosition\_n = FindSafePosition(chess, SafePosition, DangerPosition, DangerPosition\_n);

使用int FindSafePosition()找出一個棋子所有路徑下的安全座標。

所需參數:

struct Position\* chess //要找的棋子的路徑

struct Position SafePosition[] //用於儲存所有安全座標

struct Position DangerPosition[] //用於比對的危險座標

int DangerPosition\_n //危險座標的數量

**涵式內容:**

static int i = 0; //靜態變數用於儲存安全座標的數量

if (chess == NULL) return i; //如果路經不通，回傳中斷。

//如果步數小於等於5步執行

if (chess->step <= step\_limit) {

//檢查這個路經的座標，是否在危險座標當中，不在的話即是安全座標。

if (!checkInside(chess, DangerPosition, DangerPosition\_n)) {

//檢查安全座標是否重複，因為不同路徑下可以抵達的安全座標可能

相同。

if (!checkInside(chess, SafePosition, i)) {

//將資料儲存到 SafePosition []中。

SafePosition[i].x = chess->x;

SafePosition[i].y = chess->y;

SafePosition[i].step = chess->step;

i++; //安全座標數量+1

}

}

}

//向 上、下、左、右 搜索路徑

i = FindSafePosition(chess->Up, SafePosition, DangerPosition,

DangerPosition\_n);

i = FindSafePosition(chess->Down, SafePosition, DangerPosition,

DangerPosition\_n);

i = FindSafePosition(chess->Left, SafePosition, DangerPosition,

DangerPosition\_n);

i = FindSafePosition(chess->Right, SafePosition, DangerPosition,

DangerPosition\_n);

return i; //回傳安全座標的總共數量

決定移動路徑

struct Position path[6] = { 0 }; //用於紀錄移動路徑

int step = 0; //用於紀錄移動花費的步數

//假如對手的第一顆子在安全座標中

if (checkInside(other1\_path, SafePosition, SafePosition\_n)) {

//吃掉對手的第一顆子

step = GotoTargetPosition(path, chess, other1\_path);

}

//假如對手的第二顆子在安全座標中

else if (checkInside(other2\_path, SafePosition, SafePosition\_n)) {

//吃掉對手的第二顆子

step = GotoTargetPosition(path, chess, other2\_path);

}

else 否則移動到任意一個安全座標

{

//比對所有安全座標

for (int i = 0; i < SafePosition\_n; i++) {

//安全座標不能等於起始位置(即不能原地踏步)

if (SafePosition[i].x != chess->x || SafePosition[i].y != chess->y) {

//與對手的位置進行安全座標篩選

for (int j = 0; j < other\_n; j++) {

//安全座標的位置，不能在對手位置的上下左右

if (SafePosition[i].x + 1 != other\_position[j].x &&  
 SafePosition[i].y + 1 != other\_position[j].y &&  
 SafePosition[i].x - 1 != other\_position[j].x &&  
 SafePosition[i].y - 1 != other\_position[j].y)

{

//移動到安全的座標

step = GotoTargetPosition(path, chess, SafePosition[i]);

}

}

}

}

}

移動到目標位置

利用 int GotoTargetPosition()找出到目標位置的任一路徑。

所需參數:

struct Position path[6] //用於儲存移動的路徑

struct Position\* chess //要移動的棋子的所有路徑

struct Position TargetPosition //目標位置的座標

**涵式內容:**

static int step = 0; //用於計算花費的步數。

if (chess == NULL) { //如果路徑不通，回傳中斷

return step;

}

//如果路徑的步數 <= 5 執行

else if (chess->step <= step\_limit) {

//如果該路徑的座標等於目標位置的座標，代表路徑找到了。

if (chess->x == TargetPosition.x && chess->y == TargetPosition.y) {

step = chess->step; //此時步數等於該路徑的步數。

//當路徑不為NULL時執行。

while (chess != NULL) {

//該路徑的步數就等於path[]的第幾步

path[chess->step] = \*chess;

//該路徑的等於它的父節點(往回找路徑)

chess = chess->father;

}

}

}

//路徑找到後，因為已經給值給step與path[]所以不會等於NULL。

//否則，會向路徑的上、下、左、右，搜尋路徑。

if (path[step].step == NULL) {

step = GotoTargetPosition(path, chess->Up, TargetPosition);

step = GotoTargetPosition(path, chess->Down, TargetPosition);

step = GotoTargetPosition(path, chess->Left, TargetPosition);

step = GotoTargetPosition(path, chess->Right, TargetPosition);

}

return step; //回傳總共移動的步數

保存路徑

利用savePath()保存路徑

所需參數:

struct Position path[6] //移動的路徑

int step //移動的總步數

**涵式內容:**

void savePath(struct Position path[6], int step)

{

ofstream file;

file.open("play.txt");

//存入 play.txt

int i;

for (i = 0; i <= step; i++) {

file << path[i].x << " " << path[i].y << " ";

}

file.close();

}

執行與測試

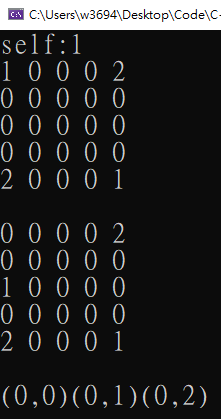
執行

起始盤面:

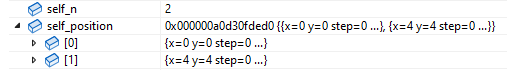
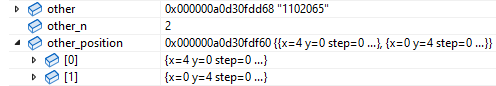
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  (0,0) | 0  (1,0) | 0  (2,0) | 0  (3,0) | 2  (4,0) |
| 0  (0,1) | 0  (1,1) | 0  (2,1) | 0  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 0  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 0  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 2  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 1  (4,4) |

我方為1，敵方為2。

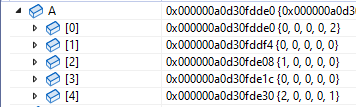
我方先手。

執行結果:  


**執行流程:**

1. **讀取資料 並 定義雙方棋子資訊**⇧我方資訊

⇧敵方資訊

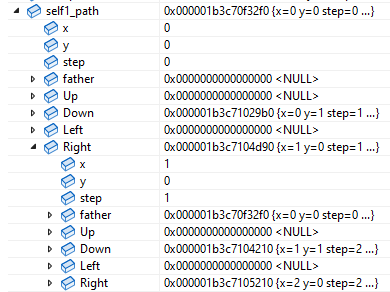


⇧棋盤資料

1. **搜索每顆棋子路徑  
   **

⇧我方路徑



⇧敵方路徑  


//因為不能走回頭路，所以Left是NULL

以此類推

//父節點為上一個節點

//起始位置路徑下的右節點

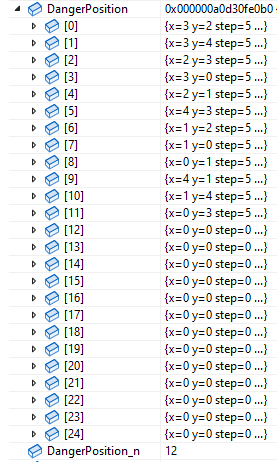
//上方及左邊無路可走

所以Up, Left為NULL

//起始位置

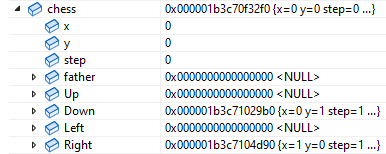
//因為是起始位置所以沒有父節點

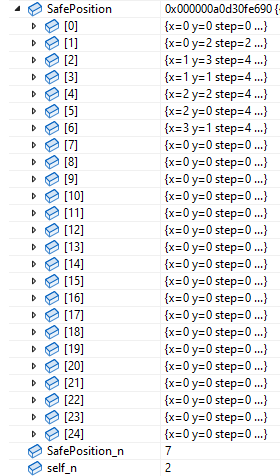
⇧展開後

1. **找出所有危險座標  
   **

內容為對手所的棋在第五步能走到的位置

⇧危險座標

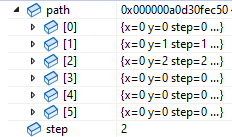
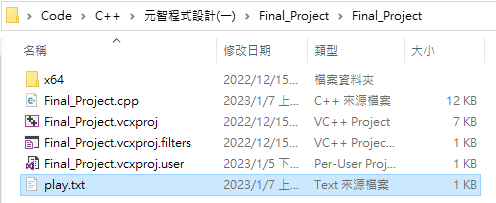
**4. 決定要移動哪一顆子**因為我方沒有任何棋子可能被敵方吃掉(在危險位置中)也沒有任何棋子被包圍住。  
所以預設移動第一顆棋子。  
 ⇧要移動的棋子

1. **找出所有安全座標**

內容為該棋子所有的路徑下的安全座標

(即移動到該座標，不會被對方吃掉)

⇧安全座標

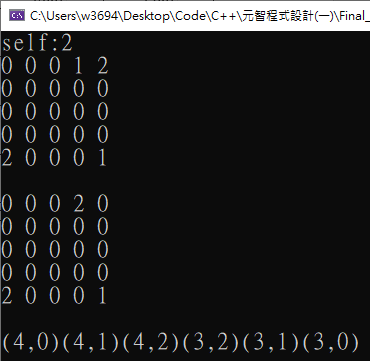
1. **決定移動路徑與步數**因為安全座標中沒有任何對方的棋子，所以棋子移動到任意一個安全的座標。  
   ****  
   ⇧移動路徑
2. **保存路徑至play.txt  
   **⇧目錄底下的play.txt  
    ****  
   ⇧play.txt內容

測試(一) 吃掉對方棋子

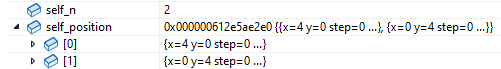
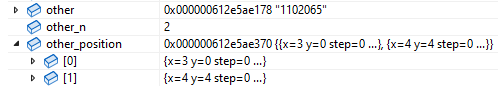
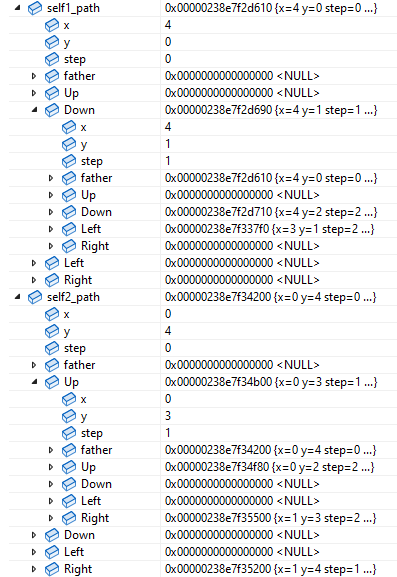
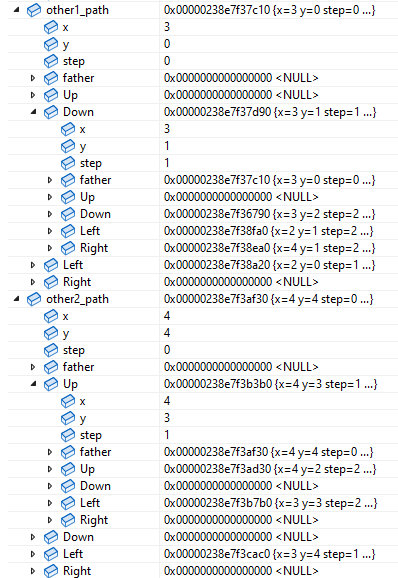
盤面資料:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0  (0,0) | 0  (1,0) | 0  (2,0) | 1  (3,0) | 2  (4,0) |
| 0  (0,1) | 0  (1,1) | 0  (2,1) | 0  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 0  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 0  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 2  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 1  (4,4) |

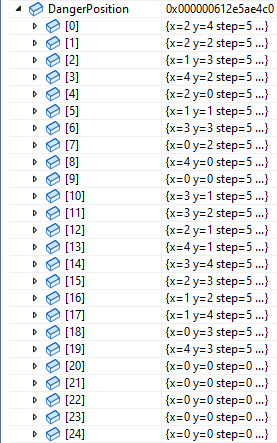
我方為2，目前棋面上(4,0)的2可以吃掉(3,0)的1。

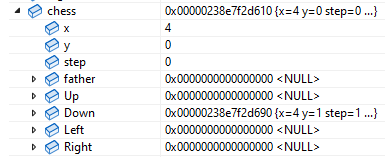
測試結果:   


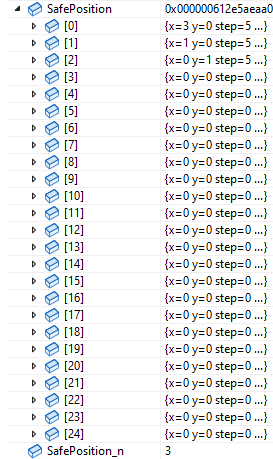
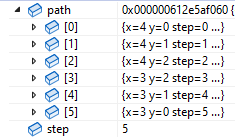
**執行流程:**

1. **讀取資料 並 定義雙方棋子資訊  
   **⇧我方資訊  
   ****  
   ⇧敵方資訊
2. **搜索每顆棋子的路徑  
     
   **⇧敵方棋子路徑

⇧我方棋子路徑

1. **找出所有危險座標  
   **⇧危險座標
2. **決定移動哪一子**

**因為第一子的座標(4,0)在危險座標中，所以移動第一子。  
**⇧要移動的棋子

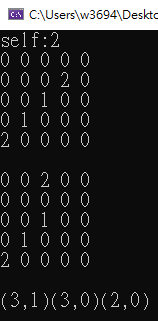
1. **找出所有安全座標  
   **⇧安全座標
2. **決定移動路徑與步數  
   **⇧移動路徑
3. **保存路徑至play.txt**

測試(二) 移動到安全座標

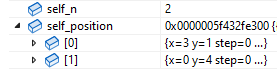
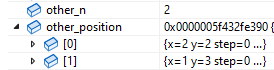
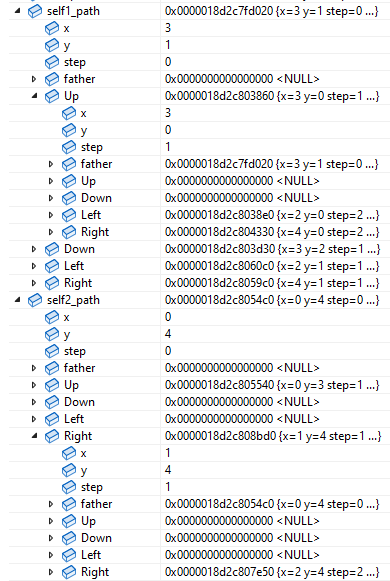
盤面資料:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0  (0,0) | 0  (1,0) | 0  (2,0) | 0  (3,0) | 0  (4,0) |
| 0  (0,1) | 0  (1,1) | 0  (2,1) | 2  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 1  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 1  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 2  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 0  (4,4) |

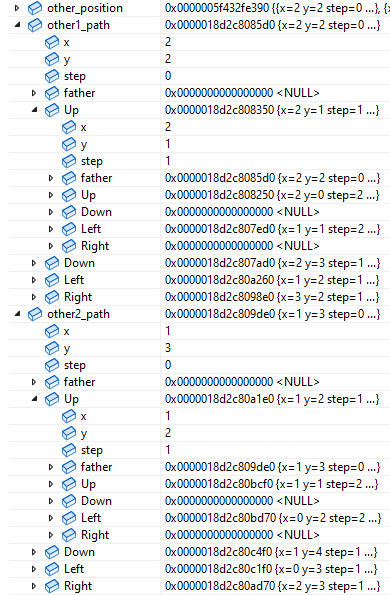
我方為二。

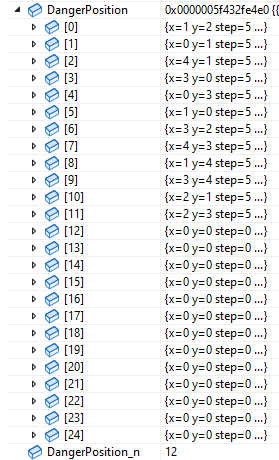
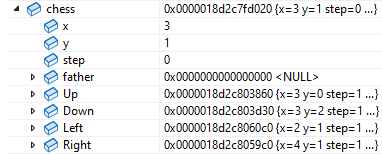
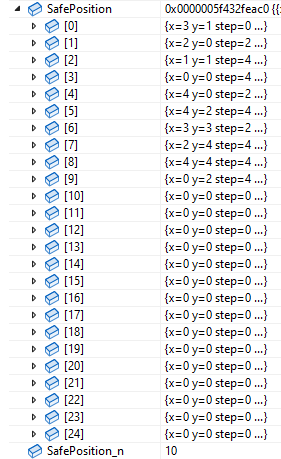
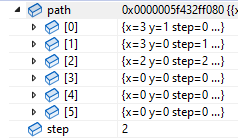
執行結果:  


**執行流程:**

1. **讀取資料 並 定義雙方棋子資訊  
   **⇧我方資訊  
   ****  
   ⇧敵方資訊
2. **搜索每顆棋子的路徑  
   **

⇧我方棋子路徑

****⇧敵方棋子路徑

1. **找出所有危險座標  
   **⇧危險座標
2. **決定移動哪一子**因為我方沒有任何一子在危險座標中也沒有被包圍，所以預設執第一子。  
   ****  
   ⇧要移動的棋子
3. **找出所有安全座標  
   **⇧安全座標
4. **決定移動路徑與步數  
   **⇧移動路徑
5. **儲存路徑至play.txt**

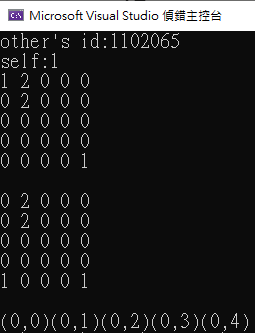
未完成(需改進)

**阻礙搜索(新增時間 2023/1/9)**

盤面資料:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  (0,0) | 2  (1,0) | 0  (2,0) | 0  (3,0) | 0  (4,0) |
| 0  (0,1) | 2  (1,1) | 0  (2,1) | 0  (3,1) | 0  (4,1) |
| 0  (0,2) | 0  (1,2) | 0  (2,2) | 0  (3,2) | 0  (4,2) |
| 0  (0,3) | 0  (1,3) | 0  (2,3) | 0  (3,3) | 0  (4,3) |
| 0  (0,4) | 0  (1,4) | 0  (2,4) | 0  (3,4) | 1  (4,4) |

我方為一。

測試結果:  


說明:  
因為在一開始對敵方棋子路徑搜索時，(0,0)的棋子阻礙到搜索路徑，所以誤判(0,4)為安全位置。

誤判 (0,4)為安全位置的原因是在一開始路經搜索時，因為(0,0)的1卡著(1,0)的2，所以無法走到(0,4)，因此判斷為安全位置。  
但(0,0)的1移動後，(1,0)的2阻礙被清除，就可以走到(0,4)了。

改進方法:

移動到(0,4)後，再對敵方旗子做一次路徑搜索，判斷移動後的位置，是否在對方第五步可以到達的位置。