围棋AI技术路线报告

目录

[1 大作业概述 2](#_Toc377016386)

[1.1 大作业背景 2](#_Toc377016387)

[1.2 大作业要求 2](#_Toc377016388)

[1.3 小组工作和本报告内容 2](#_Toc377016389)

[2 技术路线 2](#_Toc377016390)

[2.1 Monte Carlo方法 2](#_Toc377016391)

[2.2 UCT算法 3](#_Toc377016392)

[3 围棋引擎框架 4](#_Toc377016393)

[4 AI构建思路 4](#_Toc377016394)

[4.1 围棋引擎的数据结构 4](#_Toc377016395)

[4.1.1 棋局数据 5](#_Toc377016396)

[4.1.2 UCT Tree数据 5](#_Toc377016397)

[4.2 MC模块实现 6](#_Toc377016398)

[4.2.1 MC算法的核心实现 6](#_Toc377016399)

[4.2.2 候选步的产生方式及管理机制 10](#_Toc377016400)

[4.2.3 选择节点(UCT选择) 12](#_Toc377016401)

[4.2.4 展开节点 13](#_Toc377016402)

[4.2.5 棋局模拟 14](#_Toc377016403)

[4.2.6 回馈更新 15](#_Toc377016404)

[4.3棋步合法性判断模块 16](#_Toc377016405)

[4.4 更新棋盘（提子）模块 17](#_Toc377016406)

[4.5 胜负计算模块 20](#_Toc377016407)

[4.6 多线程实现 20](#_Toc377016408)

[4.7 伪气更新 21](#_Toc377016409)

1 大作业概述

1.1 大作业背景

博弈是人工智能的重要研究主题，围棋是博弈的一种，属双人零和博弈[2]。搜索空间大、计算机难于处理模糊概念且难于设计学习算法，造成了计算机围棋程序的棋力难于提高。围棋是检验人工智能发展水平的良好环境，开发出与人类棋手水平相当的围棋程序也有助于对人类认知能力的理解。所以计算机围棋研究具有重要理论意义和实用价值。

计算机围棋的难点之一，便在于缺乏良好的局面评估函数[4]，计算机围棋大多借鉴一些经验法则，以静态的评估为主，而动态的搜寻则仅用于局部的、目标明确的棋串攻杀，较少的全局搜寻。因此，人类的经验如何应运用于计算机围棋，就成了设计的重点。

1.2 大作业要求

在gtp和GOGUI基础上，开发博弈算法，实现围棋引擎。

棋盘大小：13\*13；

下子时限：首步60s，此后每步10s。

1.3 小组工作和本报告内容

小组工作：

前期：一起查找资料并讨论，确定技术路线；初步实现算法，打败brown。

后期：调整模拟深度、局数的选择；优化模拟速度；分析算法缺点，加入围棋知识；实现多线程。

1.4 主要分工：

陈燕峰：前期，uct树有关代码编写；后期，围棋技术指导，报告整理，算气算法

陈燕晶：前期：候选点选取；后期，性能评测，算气算法

马子泰：前期：调研，他人代码学习；后期，多线程，候选点选取算法改进，围棋先验知识实现

罗若天：前期：MC模拟代码；后期，多线程，伪气算法。

2 技术路线

通过对文献的学习和分析，我们决定以UCT为主要技术路线进行算法设计。下面分别介绍涉及的MC算法和UCT算法。

2.1 Monte Carlo方法

Monte Carlo[11~12]方法主要理论基础是依据大数定理，在随机取样的情况下，可以获得有误差的评估值，取样的数量越多，误差将越小，评估值将越准确。

将Monte Carlo方法应用于围棋[13]，其核心的思想是，在于通过统计许多模拟棋局的结果，进行局面的优劣判断。亦即将蒙特卡罗方法作为局面评估函数，以决定着手的好坏。

其中，所谓的模拟棋局，指的是对某一目标盘面，由电脑随机落子，直到终盘而可以判定胜负为止。在随机落子时，除了基本的围棋规则外，只有一个限制：不得自填眼位，这个限制是为了防止棋局无法结束而设定的。模拟棋局的结果，与目前常见的只判断黑胜或白胜不同，而是会判断输赢的目数，在决定着手优劣时，则是统计此着手下所有模拟棋局平均的输赢目数来决定的。

2.2 UCT算法

UCT又名UCB for Tree Search，是UCB(Upper Confidence Bound)在Tree Search上的应用。而UCB本来是为了解决吃角子老虎机问题(Bandit Problem)而产生的。所谓的吃角子老虎机问题，简述如下：目前有若干台吃角子老虎机，每台机器可以投钱并拉动操纵杆，此时会得到收益(reward)，投钱、拉杆、得到收益的过程，称之为一个Play。每台吃角子老虎机有不同的收益率，倘若玩家想要在若干次的Play里获得最大总收益，那么该怎么做？

一般来说，玩家会开始动手玩，并且依照目前积累的经验来决定下一次的Play要选择哪一台机器，这称之为开发(exploitation)。相对地，如果玩家不断地依照目前所获得的经验来决定，而不试图尝试其它的其他的机器，则可能会忽略收益率更高的机器，因此适度地尝试其他机器是必须的，这称之为探险(exploration)。如何在开发与探险之间保持平衡，就是UCB试图解决的ExE(exploitation vs. exploration)问题。

UCB根据目前获得的信息，配合上一个调整值，试图在开发与探险间保持平衡。大致上来说，每一次Play时，UCB会根据每一台机器目前的平均收益值(亦即其到目前为止的表现)，加上一个额外的参数，得出本次Play此台机器的UCB值，然后根据此值，挑选出拥有最大UCB值的机器，作为本次Play所要选择的机器。其中，所谓额外参数，会随每一台机器被选择的次数增加而相对减少，其目的在于让选择机器时，不过分拘泥于旧有的表现，而可以适度地探索其他机器。UCB公式表示如下(也称为UCB1)[14]：

(3)



是第j台机器到目前为止的平均收益，是第j台机器被测试的次数，n是所有机器目前被测试的总次数。让公式(3)的值最大的机器将是下一个被选择的机器。前项即为此台机器的过去表现，后项则是调整参数。



而UCB1-TUNED是相对于UCB1实验较佳的配置策略[14]。UCB1-TUNED的公式如下：

(4)



(5)



让公式(5)的值最大的机器将是下一个被告选择来测试的机器。

UCT(UCB for Tree Search)其实就是把UCB1或UCB1-TUNED(统称为UCB)等公式运用于Tree Search上的一个方法[14]。以概念而言，UCT把每一个节点都当作是一个吃角子老虎机问题，而此节点的每一个分支，都是一台吃角子老虎的机器。选择分支，就会获得相应的收益。

Tree Search开始时，UCT会建立一棵Tree，然后：

1. 从根节点开始
2. 利用UCB公式计算每个子节点的UCB值，选择UCB值最高的子节点
3. 若此子节点并非叶节点(从未拜访过的节点)，则由此节点开始，重复(2)
4. 直到遇到叶节点，则计算叶节点的收益值，并依此更新根节点到此一节点路 径上所有节点的收益值
5. 由(1)开始重复，直到时间结束，或达到某一预设次数
6. 由根节点的所有子节点中，选择平均收益值最高者，作为最佳节点

此一节点，就是UCT的结果。

3 围棋引擎框架

围棋引擎的总体功能模块图如图所示。

棋盘模块：棋盘的初始化，棋盘的赋值，棋盘空间的释放

提子模块：包括移除棋串(remove\_string)，更新棋盘（play\_move）

模拟棋局：MC模拟棋局（play\_random\_game），模拟建立UCT树（play\_simulation）,

产生后继节点（get\_candidate\_move）

UCT树：计算UCT（get\_uct），初始化节点(init\_node)，扩展节点(create\_children)，获得最好节点(get\_best\_child)，选择后继(uct\_select)，释放树空间（free\_tree）

多线程模块。

以及一些琐碎的工具类的模块。、

4 AI构建思路

4.1 围棋引擎的数据结构

在程序中主要有两类数据结构：一是与棋局相关数据，用于记录棋盘信息和保证棋局的正常进行；二是与UCT算法相关数据，里面保存了创建UCT Tree的节点信息及模拟棋局相关参数和数据。

4.1.1 棋局数据

我们这里用一个结构体Bundle来表示一个棋局。分别包含，board数组，go\_string数组（棋串信息），伪气信息，最终结果信息，劫位，上一步位置数据，已下步数。

typedef struct Bundle{

int \*board;

Stone \*go\_string;

int \*fake\_liberty;

int \*final\_status;

int ko\_i, ko\_j;

int last\_i,last\_j;

int step;

} Bundle;

为了存储棋串并方便查找是否属于同一个棋串、进行棋串合并，并且计算伪气，使用结构体Stone，定义如下：

/\*棋串，同一个棋串构成一个循环链，有同一个head\*/

typedef struct Stone{

int next\_stone;

int head;

} Stone;

由于下棋过程常常对棋子的位置进行二维位置与一位位置的转换，进行棋子上下左右位置的搜索，所以用以下数组和宏定义方便进行搜索。

/\* 用于对棋子周围循环查看 \*/

static int deltai[4] = {-1, 1, 0, 0};

static int deltaj[4] = {0, 0, -1, 1};

/\* 一维坐标和二维坐标的转换. \*/

#define POS(i, j) ((i) \* board\_size + (j))

#define I(pos) ((pos) / board\_size)

#define J(pos) ((pos) % board\_size)

/\* 对方棋子的颜色 \*/

#define OTHER\_COLOR(color) (WHITE + BLACK - (color))

4.1.2 UCT Tree数据

这里定义了UCT Tree结点的结构，说明了结点代表的着手，胜利的次数，被访问的次数，以及结点之间的关系。

/\* UCT Tree的节点 \*/

typedef struct Node{

int wins; /\* 胜利的次数 \*/

int visits; /\* 被访问的次数 \*/

int x,y; /\* 该节点代表的着手 \*/

struct Node \*child,\*sibling,\*parent; /\* 节点的子节点和伙伴节点 \*/

int more; /\* 赢得目数\*/

} Node;

4.2 MC模块实现

4.2.1 MC算法的核心实现

UCT算法流程实现主要在play\_simulation函数，而uct\_search是调用play\_simulation函数来提供产生棋步着手的接口函数。

uct\_search函数的功能是在进行UCT搜索之前进行初始化工作，包括创建和初始化根节点，首次展开节点。主体是进行UCT搜索（持续时间<10s），并在调用play\_simulation之前复制棋局信息。当UCT搜索时间>=10s时，从根节点下面找到最佳子节点并返回该节点代表的着手和释放UCT树，恢复原棋盘相关数据。

在uct\_search函数中根据实验要求，控制其搜索的时间长度，在其他条件不变的情况下，通常初期由于棋盘的空位太多，模拟一次的时间较长，所以模拟次数有限，之后，随着棋盘的棋子数增多，棋盘模拟的次数增多，棋力也会越强。开局时因为占角有利，我们运用已有的围棋知识直接下子，不做模拟。代码如下：

if (step<=8){

if (get\_board(&current,3,2)==EMPTY){

\*x=3;

\*y=2;

return;

}

if (get\_board(&current,3,2)==OTHER\_COLOR(color)&&get\_board(&current,4,4)==EMPTY){

\*x=4;

\*y=4;

return;

}

if (get\_board(&current,3,4)==EMPTY){

\*x=3;

\*y=4;

return;

}

if (get\_board(&current,3,4)==OTHER\_COLOR(color)&&get\_board(&current,4,2)==EMPTY){

\*x=4;

\*y=2;

return;

}

if (get\_board(&current,9,10)==EMPTY){

\*x=9;

\*y=10;

return;

}

if (get\_board(&current,9,10)==OTHER\_COLOR(color)&&get\_board(&current,8,8)==EMPTY){

\*x=8;

\*y=8;

return;

}

在uct\_search函数中根节点初始化时子节点为空，被访问次数为0，进行模拟之前必须对其展开子节点(create\_children)，才能在play\_simulation函数中选择子节点。展开子节点时包含了对子节点裁减的判断，控制搜索棋步的广度。

在uct\_search函数中get\_best\_child函数是获得根节点下面被访问次数最多的子节点，此节点代表的着手被认为是UCT算法产生的最佳着手。

void uct\_search(int \*x,int \*y,int color){

log\_file=fopen("log\_file.txt","at");

fprintf(log\_file,"uct\_search\n");

fclose(log\_file);

if (quick\_judge(x, y, color))

return;

Node \*root =(Node \*) malloc(sizeof(Node));

init\_node(-1,-1,root,NULL);

create\_children(current,root,color);

if (root->child==NULL){

\*x=-1;

\*y=-1;

return;

}

int start\_time=clock();

//init thread

readcount=0;

mutex=CreateMutex(NULL,FALSE,NULL);

wrt=CreateMutex(NULL,FALSE,NULL);

chd=CreateMutex(NULL,FALSE,NULL);

//

play\_simulation\_thread(current,root,color);

//close

CloseHandle(mutex);

CloseHandle(wrt);

CloseHandle(chd);

//

Node \*n=get\_best\_child(root);

\*x=n->x;

\*y=n->y;

log\_file=fopen("log\_file.txt","at");

fprintf(log\_file,"####\ntime spent: %dms\nsimulate count: %d\n####\n",clock()-start\_time,root->visits);

free\_tree(root);

fclose(log\_file);

}

play\_simulation函数功能是实现了UCT算法的流程，表现为递归选择子节点直到选中符合要求的叶节点(在必要时进行展开节点)，然后进行模拟棋局，最后将棋局胜负结果回馈更新。

在play\_simulation函数中变量NUM\_VISITS控制节点被扩展时的次数，能降低UCT Tree节点在内存的增长速率，在相同模拟次数的情况下，增加对节点搜索模拟的次数，在一定程度上增加程序搜索的有效程度，负面影响是会减少棋步搜索的深度。

在play\_simulation函数中play\_random\_game函数包含了模拟棋局的功能：棋形模式匹配、长气逃子和吃子。uct\_select函数中要优先选择未被访问过的子节点，这也是一个策略问题。关于play\_random\_game和uct\_select函数的功能在后面小节会有详细说明。

至于play\_move函数则是根据围棋规则对落子进行更新棋盘处理，关于基本围棋规则会在讨论play\_random\_grame函数模拟棋局功能时进行介绍。而update\_node函数是当模拟棋局胜利时将胜率加1，被访问次数不管胜利与否都加1。注意update\_node更新节点信息时，模拟棋局胜负的结果对于不同深度的节点是相反的。某一深度的节点的父节点和子节点代表对方着手的节点，如果模拟棋局结果对该节点来说是胜利着手，则模拟棋局对于对方而言就是失败的，所以更新到该节点的父节点和子节点的胜负结果是相反的。实际上也就是说，UCT树本质是一棵极小极大(minimax)树，对于己方有利的棋步对对方则是不利的，反之亦然。

/\*模拟棋局\*/

//color是下一步要走的人的颜色，node是现在这步

void play\_simulation(Bundle \*bundle,Node \*n,int color){

int randomresult=0;

while (1){

if (n->child==NULL && n->visits<NUM\_VISITS){

break;

}

else{

if(n->child==NULL)

create\_children(bundle,n,color);

Node \*next=uct\_select(n);

play\_move(bundle,next->x,next->y,color);

n=next;

color=OTHER\_COLOR(color);

}

}

play\_random\_game(bundle,n,color);

}

4.2.2 候选步的产生方式及管理机制

棋局中，用int数组candidate\_moves表示候选点，为0表示非候选点，权值越大的越容易成为最佳候选节点。

具体实现如下：

void get\_candidate\_moves(Bundle \*bundle,int candidate\_moves[][MAX\_BOARD],int color){

int i;

int ai, aj;

int k;

int s\_count,o\_count;

for (ai = 0; ai < board\_size; ai++)

for (aj = 0; aj < board\_size; aj++) {

candidate\_moves[ai][aj]=0;

if (legal\_move(bundle,ai, aj, color) && !suicide(bundle,ai, aj, color)) { //这个点合理且不是自杀点

if (!suicide(bundle,ai, aj, OTHER\_COLOR(color)))//不是对方自杀点

candidate\_moves[ai][aj]=1;

else { //是对方自杀点，且周围有对方的棋子

for (k = 0; k < 4; k++) {

int bi = ai + deltai[k];

int bj = aj + deltaj[k];

if (on\_board(bi, bj) && get\_board(bundle,bi, bj) == OTHER\_COLOR(color)) {

candidate\_moves[ai][aj]=1;

break;

}

}

}

if(!suicide(bundle,ai,aj,OTHER\_COLOR(color)) ){

//对方非自杀点，且周围有空位，则权值为2

for (k = 0; k < 4; k++) {

int di = ai + deltai[k];

int dj = aj + deltaj[k];

if (on\_board(di, dj) && get\_board(bundle,di, dj) != EMPTY) {

candidate\_moves[ai][aj]=2;

}

}

}

}

如果只选这些点，可能忽略一些边上的点，因而，当搜索数达到一定数目时，需用下面一些机制来产生候选节点。

由于每下一子，就会让周围的空点成为下一步合适的候选步。因此，产生候选步的方式就是对于每一个空点，检查其周围是否有下子，看是否将其纳入候选步。而检查的方式是根据空点所在线，周围棋子与空点之间的距离和已下棋子的步数来判断的。其中两个棋子的距离是指两个棋子横坐标差的绝对值和纵坐标差的绝对值之和，即有P1(X1, Y1)和P2(X2, Y2)，则两个棋子距离为

|P1P2| = |X1-X2| + |Y1-Y2|

相对应的方法有一线空点，二线空点，以及三线空点。

在实际应用中，我们队这三种情况作了一个近似，以减少其搜索复杂度。

4.2.3 选择节点(UCT选择)

上文中提到过的uct\_select函数是从父节点中选择uct值最大的节点进行搜索，其中uct值的计算如下：

/\*计算结点parent的儿子child的uct值\*/

float get\_uct(Node \*parent,Node \*child)

{

float UCTK=0.5;

float uct;

if(child->visits>0)

{

/\* 胜率, UCB公式的过去表现 \*/

float winrate=(float)child->wins/child->visits;

/\* UCB公式 = 过去表现 + 调整参数 \*/

uct=winrate + UCTK\*sqrt(log((float)parent->visits)/child->visits);

}

else

/\* 总是优先随机选择一个未访问过的子节点 \*/

uct = 10000 + 1000\*rand();

return uct;

}

选择节点时，若有未被访问过的子节点，则优先以随机方式选择其中一个子节点落子然后执行模拟棋局，否则继续使用节点选择公式选择子节点。

当子节点从未被访问过时，使用下面的公式计算uct值：

uct = 10000 + 1000 \* rand(); (1)

当节点被访问的次数很少时，用公式

uct= winrate + UCTK \* sqrt(log(parent->visits) / child->visits) (2)

计算出来的uct值实际很小，远远小于用公式(1)计算出来的uct值，这就确保了能够优先随机未被访问过的子节点。

有了uct值，就可以进行节点选择。

/\*选择uct值最大的节点，输出这个节点\*/

Node\* uct\_select(Node \*node)

{

if(!node->child)

return NULL;

Node\* parent=node;

Node\* child=node->child;

Node\* max\_node=child;

float max\_uct=get\_uct(parent,child);

//对parent的儿子进行搜索，找到uct值最大的节点

while(child->sibling)

{

child=child->sibling;

float tmp\_uct=get\_uct(parent,child);

if (max\_uct<tmp\_uct)

{

max\_node=child;

max\_uct=tmp\_uct;

}

}

return max\_node;

}

4.2.4 展开节点

当节点为叶节点并且该节点被访问的次数达到指定的次数时，进行展开子结点。展开时对候选步作筛选，去除不适合的候选步，再将筛选后的候选步展开成子节点并随机选择其中一个节点。

候选步筛选最基本的就是将不合法及不适合的候选步予以去除，此处用到的是棋步合法性判断，具体实现将在后面谈到。筛选后，展开节点，用函数creat\_children()。

/\*展开节点\*/

void create\_children(Bundle \*bundle,Node\* node,int color){

int i,j;

int candidate\_moves[MAX\_BOARD][MAX\_BOARD];

Node \*last=NULL;

get\_candidate\_moves(bundle,candidate\_moves,color); //获取候选点

for(i=0;i<board\_size;i++)

for(j=0;j<board\_size;j++)

if (candidate\_moves[i][j]){

Node \*tmp =(Node \*) malloc(sizeof(Node));

init\_node(i,j,tmp,node); //把候选点用节点表示

tmp->sibling=last; //加入子节点

last=tmp;

}

node->child=last;

if (node->child==NULL){

Node \*tmp =(Node \*) malloc(sizeof(Node));

init\_node(-1,-1,tmp,node);

node->child=tmp;

tmp->sibling=NULL;

}

}

4.2.5 棋局模拟

模拟棋局的好坏是决定节点(着手)好坏的一个重要依据，因此，模拟棋局的合理性与正确性是相当重要的。这里我们限制了模拟的次数，这是因为我们不希望重复出现同一棋局的情况，所以限制模拟次数，防止出现死循环。

int play\_random\_game(Bundle \*bundle, Node \*node, int color) {

Bundle \*s\_bundle=(Bundle \*)malloc(sizeof(Bundle));

init\_board(s\_bundle);

clone(s\_bundle,bundle);

int pass\_num = 0;

int current\_color = color;

int count=0;

while (pass\_num < 2 &&count<300) {

int i;

int j;

count++;

s\_generate\_move(bundle, &i, &j, current\_color);

if (pass\_move(i, j))

pass\_num++;

else

pass\_num = 0;

play\_move(bundle, i, j, current\_color);

current\_color = OTHER\_COLOR(current\_color);

}

if (count==300){

free\_board(s\_bundle);

return 0;

}

int score = get\_final\_win(bundle, color);

free\_board(s\_bundle);

update\_node(-score, node);

return score;

}

4.2.6 回馈更新

每次更新的时候，每个点不仅更新，还更新赢得目数。通过循环将值一直传递到根节点。

void update\_node(int result,Node \*n){

wait(wrt);

int res;

if (result>0)

res=1;

else

res=-1;

while (n!=NULL){

n->wins+=res;

n->more+=result;

n->visits++;

n=n->parent;

result=-result;

res=-res;

}

signal(wrt);

}

4.3棋步合法性判断模块

由于使用中国围棋规则，因此自杀着手是禁着的一种。第二种是打劫，第三种是真眼。自杀判断： 合法性判断：

/\* 判断是否为自杀\*/

static int suicide(Bundle \*bundle,int i, int j, int color)

{

int k;

for (k = 0; k < 4; k++)

//判断该点周围的点是否能给他提供气

if (provides\_liberty(bundle,i + deltai[k], j + deltaj[k], i, j, color))

return 0;

return 1;

}

int legal\_move(Bundle \*bundle,int i, int j, int color)

{

int other = OTHER\_COLOR(color);

/\* 虚着是合法的 \*/

if (pass\_move(i, j))

return 1;

/\* 下在非空点是不合法的 \*/

if (get\_board(bundle,i, j) != EMPTY)

return 0;

/\* 打劫的判断 \*/

if (i == \*(bundle->ko\_i) && j == \*(bundle->ko\_j)

&& ((on\_board(i - 1, j) && get\_board(bundle,i - 1, j) == other)

|| (on\_board(i + 1, j) && get\_board(bundle,i + 1, j) == other)))

return 0;

return 1;

}

4.4 更新棋盘（提子）模块

落子时，分以下情况：（1）虚着，即pass，则什么都不做；（2）落子为自杀，则提我方棋串；（3）落子后提对方子，此时需要记录提子数目，做为劫争点的判断依据；（4）落子后没有提子。对后两种情况，都需要棋盘上记录落子信息，并进行棋串合并。需要注意的是，提子时，并不需要更改棋串信息，只需要对同一棋串上的子记录这里是空的就可以。因为对空点，我们不会去查询棋串信息；落子时，空点上的棋串信息会被更新，不会有影响。

void play\_move(Bundle \*bundle,int i, int j, int color)

{

bundle->step++;

bundle->last\_i=i;

bundle->last\_j=j;

int pos = POS(i, j);

int captured\_stones = 0;

int k;

/\* Reset the ko point. \*/

bundle->ko\_i = -1;

bundle->ko\_j = -1;

if (pass\_move(i, j))

return;

if (suicide(bundle,i, j, color)) {

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)

&& get\_board(bundle,ai, aj) == color)

remove\_string(bundle,ai, aj);

}

return;

}\*/

bundle->board[pos] = color;

bundle->go\_string[pos].next\_stone = pos;

bundle->go\_string[pos].head=pos;//保À¡ê证¡è了¢?next\_stone[pos]不?会¨¢受º¨¹之?前¡ã在¨²这a个?点Ì?上¦?的Ì?子Á¨®的Ì?影®¡ã响¨¬

int lib=0;

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)){

int c=get\_board(bundle,ai,aj);

if (c==EMPTY)

lib++;

else

--bundle->fake\_liberty[bundle->go\_string[POS(ai,aj)].head];

}

}

bundle->fake\_liberty[bundle->go\_string[pos].head]=lib;

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)){

int c=get\_board(bundle,ai,aj);

if (c==color){

if (!same\_string(bundle,pos,POS(ai,aj))){

combine(bundle,pos,POS(ai,aj));

}

}

else

{

if (c==OTHER\_COLOR(color)){

if (bundle->fake\_liberty[bundle->go\_string[POS(ai,aj)].head]==0){

captured\_stones+=remove\_string(bundle,ai,aj);

}

}

}

}

}

/\*

//把ã?对?方¤?的Ì?子Á¨®吃?了¢?

// Not suicide. Remove captured opponent strings.

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)

&& get\_board(bundle,ai, aj) == OTHER\_COLOR(color)

&& !has\_additional\_liberty(bundle,ai, aj, i, j))

captured\_stones += remove\_string(bundle,ai, aj);

}\*/

/\* Put down the new stone. Initially build a single stone string by

\* setting next\_stone[pos] pointing to itself.

\*/

/\*

bundle->board[pos] = color;

bundle->next\_stone[pos] = pos;//保À¡ê证¡è了¢?next\_stone[pos]不?会¨¢受º¨¹之?前¡ã在¨²这a个?点Ì?上¦?的Ì?子Á¨®的Ì?影®¡ã响¨¬

\*/

/\* 合?并¡é棋?串ä?（ê¡§四?周¨¹都?要°a看¡ä一°?下?）ê?

\*/

/\*

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

int pos2 = POS(ai, aj);

// 因°¨°为a有®D可¨¦能¨¹两¢?个?方¤?向¨°的Ì?子Á¨®原-来¤¡ä在¨²一°?个?棋?串ä?里¤?，ê?所¨´以°?，ê?连¢?接¨®的Ì?时º¡À候¨°要°a保À¡ê证¡è不?会¨¢重?复¡ä添¬¨ª加¨®进?棋?串ä?

if (on\_board(ai, aj) && bundle->board[pos2] == color && !same\_string(bundle,pos, pos2)) {

// The strings are linked together simply by swapping the the

// next\_stone pointers.

int tmp = bundle->next\_stone[pos2];

bundle->next\_stone[pos2] = bundle->next\_stone[pos];

bundle->next\_stone[pos] = tmp;

}

}\*/

/\* 防¤¨¤止1回?提¬¨¢

\*/

if (captured\_stones == 1 && bundle->go\_string[pos].next\_stone == pos) {

int ai, aj;

/\* Check whether the new string has exactly one liberty. If so it

\* would be an illegal ko capture to play there immediately. We

\* know that there must be a liberty immediately adjacent to the

\* new stone since we captured one stone.

\*/

for (k = 0; k < 4; k++) {

ai = i + deltai[k];

aj = j + deltaj[k];

if (on\_board(ai, aj) && get\_board(bundle,ai, aj) == EMPTY)

break;

}

if (!has\_additional\_liberty(bundle,i, j, ai, aj)) {

bundle->ko\_i = ai;

bundle->ko\_j = aj;

}

}

}

/\* 合并棋串（四周都要看一下） \*/

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

int pos2 = POS(ai, aj);

/\*

因为有可能两个方向的子原来在一个棋串里，所以，连接的时候要保证不会重复添加进棋串

\*/

if (on\_board(ai, aj) && bundle->board[pos2] == color && !same\_string(bundle,pos, pos2)) {

int tmp = bundle->go\_string[pos2].next\_stone;

int head = bundle->go\_string[pos2].head;

bundle->go\_string[pos2].next\_stone = bundle->go\_string[pos].head;

Stone union\_stone = bundle->go\_string[pos];

while(union\_stone.next\_stone!=NULL)

{

union\_stone.head = head;

union\_stone = bundle->go\_string[union\_stone.next\_stone];

}

union\_stone.next\_stone = tmp;

}

}

/\* 防止回提

\*/

if (captured\_stones == 1 && bundle->go\_string[pos].next\_stone == NULL) {

int ai, aj;

/\* 检查是否被提的子和新落子的棋串是否都是一个子。若是，将称为可能的劫争点。需要进行判断并记录\*/

for (k = 0; k < 4; k++) {

ai = i + deltai[k];

aj = j + deltaj[k];

if (on\_board(ai, aj) && get\_board(bundle,ai, aj) == EMPTY)

break;

}

if (!has\_additional\_liberty(bundle,i, j, ai, aj)) {

\*(bundle->ko\_i) = ai;

\*(bundle->ko\_j) = aj;

}

}

}

4.5 胜负计算模块

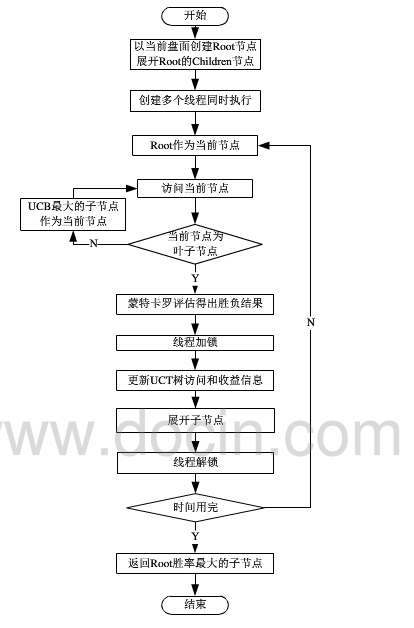
对胜负计算，我们以实战为最高原则，在双方都pass后，认为棋局结束，开始进行数子以计算胜负。数子时需要对棋盘上的子的死活状态进行判断。我们这里使用了brown的compute\_final\_status,可以非常快速的解决肉类。

4.6 多线程实现

为了提高模拟效率，很直观的一个方法就是进行多线程。这里使用的是win32的多线程API。

其中首先将参数传入play\_simulation\_thread，然后play\_simulation复制好棋盘，然后分别将参数信息打包封装，启动线程play\_thread。

在play\_thread中将参数再解包，送到play\_simulation中执行。（提高约2倍）



void play\_simulation\_thread(Bundle \*bundle,Node \*n,int color){

PMYDATA param[MAX\_THREAD];

HANDLE hThread[MAX\_THREAD];

DWORD hid[MAX\_THREAD];

int i;

for(i=0;i<NUM\_THREAD;i++){

param[i]=(PMYDATA) malloc(sizeof(MyDATA));

param[i]->bundle = (Bundle \*)malloc(sizeof(Bundle));

init\_board(param[i]->bundle);

clone(param[i]->bundle,bundle);

param[i]->node = n;

param[i]->color = color;

hThread[i] = CreateThread( NULL, 0, play\_thread, param[i], 0, &hid[i]);

}

// Wait until all threads have terminated.

WaitForMultipleObjects(NUM\_THREAD, hThread, TRUE, INFINITE);

for(i=0;i<NUM\_THREAD;i++){

free\_board(param[i]->bundle);

}

// Close all thread handles and free memory allocations.

for(i=0; i<NUM\_THREAD; i++)

{CloseHandle(hThread[i]);

}

}

DWORD WINAPI play\_thread(PVOID Param)

{

PMYDATA parameter;

parameter = (PMYDATA)Param;

Bundle \*bundle=parameter->bundle;

Node \*root=parameter->node;

int color=parameter->color;

Bundle \*s\_bundle=(Bundle \*)malloc(sizeof(Bundle));

init\_board(s\_bundle);

clone(s\_bundle,bundle);

clock\_t start\_time;

start\_time=clock();

while((clock()-start\_time)<RUN\_TIME){

play\_simulation(s\_bundle,root,color);

clone(s\_bundle,bundle);

}

free\_board(s\_bundle);

return 0;

}

4.7 伪气更新

起初我们希望找到一个行而有效的办法记录每个点的气的大小，结果发现需要考虑的情况太多，最后被我们放弃了。

而伪气则是我们采取的一个气的近似值。它是我们这个程序非常重要的部分，因为它显著地提高的模拟的效率（约1.4倍）。

“伪气”指的是只要有一个空的交叉点，那么这个交叉点周围的每个棋子都能得到一口气，这就叫伪气。

├┼┼┼┼

├┼┼┼┼

○○○┼┼

●●○┼┼

└●○┴┴

上图黑棋真实的气只有一口，但是按伪气来说，就有两口，因为那个空点连着两个黑子，每个黑子都算有一口气。

伪气能够非常方便的计算一个棋串是否没气。当一个棋串没气，当且仅当它的伪气也是零。伪气的更新也非常简单，每下一子，就减掉上下左右共4口气，每提走一子，则加上四口气。

//提子加气

int remove\_string(Bundle \*bundle,int i, int j)

{

int pos = POS(i, j);

int removed = 0;

do {

bundle->board[pos] = EMPTY;

bundle->go\_string[pos].head=-1;

removed++;

pos = bundle->go\_string[pos].next\_stone;

} while (pos != POS(i, j));

int k;

do {

int ai=I(pos);

int aj=J(pos);

for(k=0;k<4;k++){

int bi=ai+deltai[k];

int bj=aj+deltaj[k];

if (on\_board(bi, bj) && get\_board(bundle,bi,bj)!=EMPTY)

bundle->fake\_liberty[bundle->go\_string[POS(bi,bj)].head]++;

}

pos = bundle->go\_string[pos].next\_stone;

} while (pos != POS(i, j));

return removed;

}

//合并棋串

void combine(Bundle \*bundle,int pos1,int pos2){

int pos = pos2;

int lib1=bundle->fake\_liberty[bundle->go\_string[pos2].head];

do {

bundle->go\_string[pos].head=bundle->go\_string[pos1].head;

pos = bundle->go\_string[pos].next\_stone;

} while (pos != pos2);

int tmp = bundle->go\_string[pos2].next\_stone;

bundle->go\_string[pos2].next\_stone = bundle->go\_string[pos1].next\_stone;

bundle->go\_string[pos1].next\_stone = tmp;

bundle->fake\_liberty[bundle->go\_string[pos1].head]+=lib1;//伪气直接相加

}

要注意更新时，要判断这个子两个方向上的邻子是否在同一个棋串。在判断是否是同一个棋串方面，比棋串的合并要复杂一些。因为棋串每朝一个方向检测就会合并，因此可以只记住上一个检测点并用same\_string()函数与之比较；但是判断要不要更新气时，要和之前检测过的没一个方向比较。一开始忽略了这个问题，后来和组员分析后，用循环变量kk进行再次检测。

一个是play\_move()函数中的落子后，对这个子单独检测四周，根据空点计算气。合并棋串时的相关操作：

bundle->board[pos] = color;

bundle->go\_string[pos].next\_stone = pos;

bundle->go\_string[pos].head=pos;

int lib=0;

//四周的棋串先各减去一个气（不需要管是否属于一个棋串）

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)){

int c=get\_board(bundle,ai,aj);

if (c==EMPTY)

lib++;

else

--bundle->fake\_liberty[bundle->go\_string[POS(ai,aj)].head];

}

}

bundle->fake\_liberty[bundle->go\_string[pos].head]=lib;

//颜色一样的合并，不一样的提子

for (k = 0; k < 4; k++) {

int ai = i + deltai[k];

int aj = j + deltaj[k];

if (on\_board(ai, aj)){

int c=get\_board(bundle,ai,aj);

if (c==color){

if (!same\_string(bundle,pos,POS(ai,aj))){

combine(bundle,pos,POS(ai,aj));

}

}

else

{

if (c==OTHER\_COLOR(color)){

if (bundle->fake\_liberty[bundle->go\_string[POS(ai,aj)].head]==0){

captured\_stones+=remove\_string(bundle,ai,aj);

}

}

}

}

}

伪气的最大好处在于在做运算的时候无需考虑一个点是否相邻棋串中的多个点，事实上，这就是伪气所求的。我们通过这种方法，能够将计算自杀的函数的复杂度从4个棋串的长度减小到4个点。