北京大学暑期课《ACM/ICPC竞赛训练》

北京大学信息学院 郭炜
guo_wei@PKU.EDU.CN
http://weibo.com/guoweiofpku

课程网页：http://acm.pku.edu.cn/summerschool/pku_acm_train.htm

高级搜索算法

A*,迭代加深, IDA*,Alpha-Beta剪枝

本讲义并非全部原创，拷贝并编辑整理了大量网络资源，仅用于内部授课

PEKING UNIVERSITY

A*算法

启发式搜索算法(A算法)

- 在BFS算法中，若对每个状态n都设定估价函数f(n)=g(n)+h(n)，并且每次从Open表中选节点进行扩展时，都选取f值最小的节点，则该搜索算法为启发式搜索算法，又称A算法。
- g(n) : 从起始状态到当前状态n的代价
- h(n) : 从当前状态n到目标状态的估计代价

A算法的例子--八数码

2	6	3
1	8	
7	4	5

1	2	3
8		4
7	6	5

定义估价函数：

$$f(n) = g(n) + h(n)$$

$g(n)$为从初始节点到当前节点的步数

$h(n)$为当前节点“不在位”的方块数

h计算举例

2	6	3
1	8	
7	4	5

$\Longrightarrow$

1	2	3
8		4
7	6	5

2,6,1,8,4,5都不在位，因此h(n) = 6

A*算法

- A算法中的估价函数若选取不当，则可能找不到解，或找到的解也不是最优解。因此，需要对估价函数做一些限制，使得算法确保找到最优解（步数，即状态转移次数最少的解）。A*算法即为对估价函数做了特定限制，且确保找到最优解的A算法。

A*算法

- f*(n)=g*(n) +h*(n)

  f*(n)：从初始节点S0出发，经过节点n到达目标节点的最小步数 (真实值）。

  g*(n): 从S0出发，到达n的最少步数（真实值）

  h*(n): 从n出发，到达目标节点的最少步数（真实值）

  估价函数f(n)则是f*(n)的估计值。

A*算法

- $f(n)=g(n)+h(n)$,且满足以下限制：

$g(n)$是从s0到n的真实步数（未必是最优的），因此：

$g(n)>0$ 且$g(n)>=g^*(n)$

$h(n)$是从n到目标的估计步数。估计总是过于乐观的，即

$h(n)<=h^*(n)$

且$h(n)$相容，则A算法转变为A*算法。A*正确性证明略。

A*算法

h(n)的相容:

如果h函数对任意状态s1和s2还满足:

$h(s1) <= h(s2) + c(s1,s2)$

c(s1,s2）是s1转移到s2的步数，则称h是相容的。

h相容能确保随着一步步往前走，f递增，这样A*能更高效找到最优解。

h相容 =>

$g(s1) + h(s1) <= g(s1) + h(s2) +c(s1,s2) = g(s2)+h(S2)$

=>

$f(s1) <= f(s2)$ 即f是递增的。

A * 算法伪代码(在节点信息中记录了其父节点):

```
open=[Start]
closed=[]
while open不为空 {
    从open中取出估价值f最小的结点n
    if n == Target then
        return 从Start到n的路径 // 找到了！！！
    else {
        for n的每个子结点x {
            if x in open {
                计算新的f(x)
                比较open表中的旧f(x)和新f(x)
                if 新f(x) < 旧f(x) {
                    删掉open表里的旧f(x)，加入新f(x)
                }
            }
            else if x in closed {
                计算新的f(x)
                比较closed表中的旧f(x)和新f(x)
                if 新f(x) < 旧f(x) {
                    remove x from closed
                    add x to open
                }
            }  // 比较新f(x)和旧f(x) 实际上比的就是新旧g(x),因h(x)相等
            else {
            // x不在open，也不在close，是遇到的新结点
                计算f(x) add x to open
            }
        }
        add n to closed
    }
}
//open表为空表示搜索结束了，那就意味着无解！
```

A*算法

A*算法的搜索效率很大程度上取决于估价函数h(n)。一般说来，在满足h(n)≤h*(n)的前提下，h(n)的值越大越好。

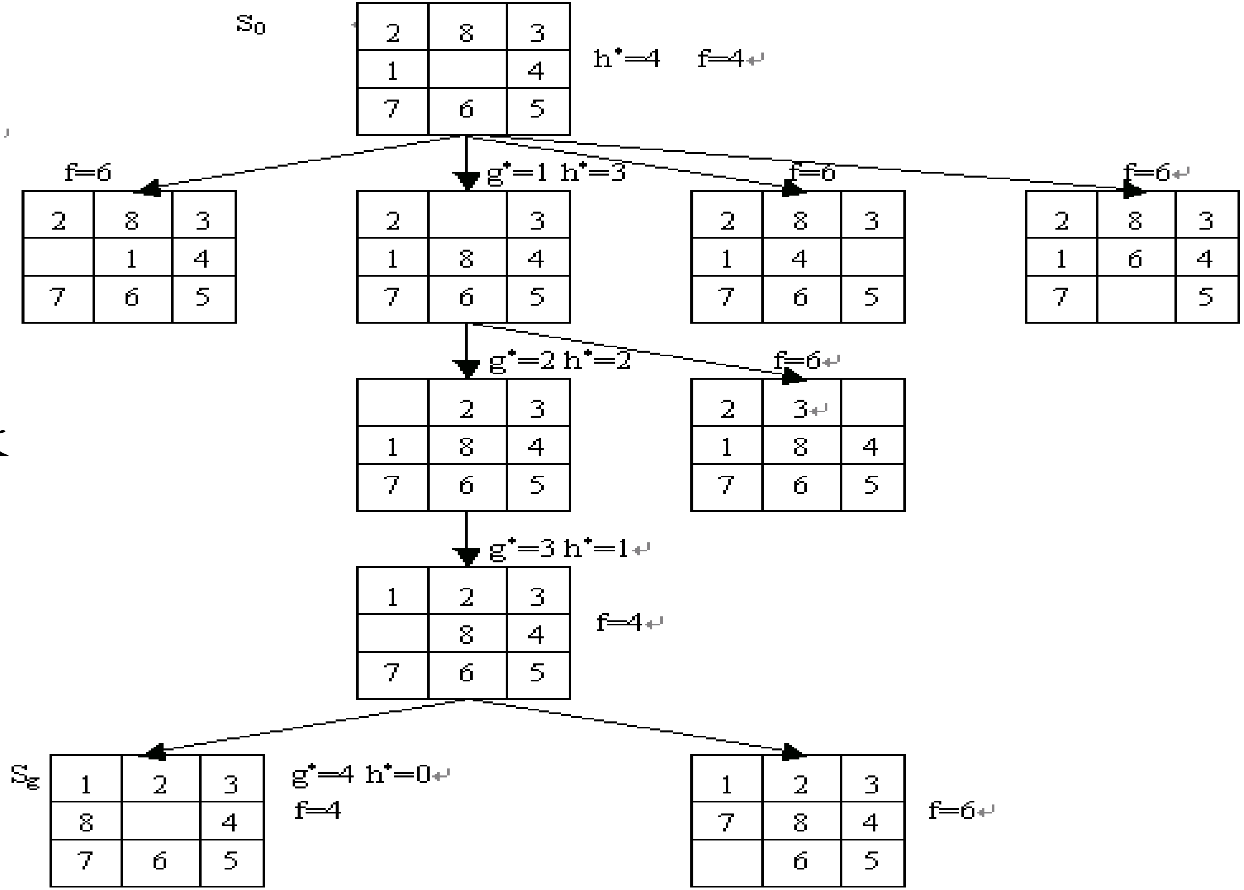
八数码问题：

方案一. h(n) 为不在位的数字个数

方案二. h(n) 为不在位的数字到其该呆的位置的曼哈顿距离和

后者优于前者

A*算法解决八数码问题



h(n) 为不在位的数字到其该呆的位置的曼哈顿距离和

```
//八数码问题， AStar POJ 250MS HDU 3734MS
#include <iostream>
#include <bitset>
#include <cstring>
#include <set>
using namespace std;
const int DIGIT_NUM = 9;
int const NODES =  362880 + 20;
int nGoalStatus;  //目标状态
struct Node {
        int status;    int f;        int g;        int h;
        int parent; // parent代表一个状态，而不是状态的索引
        char move; //经由何种动作到达本状态
};
bool operator < ( const Node & n1,const Node & n2) {        return n1.f < n2.f;  }
multiset<Node > open;
multiset<Node > closed;//也许用vector也可以。用multiset只是在从 closed表里删除元素时快些。其实closed表就是一个 362880元素的数组，可能更快
bitset<NODES> inOpen;
bitset<NODES> inClosed;
multiset<Node>::iterator openIdx[NODES]; ' openIdx[i] 就是状态 i 在open表里的地址
multiset<Node>::iterator closedIdx[NODES];
char szResult[NODES];   //结果
char szMoves[NODES]; //移动步骤 ：  u/d/r/l
char sz4Moves[] = "udrl";//四种动作
```

```
template< class T>
unsigned int GetPermutationNum(T * first, T * permutation,int len)
{ //permutation编号从0开始算, [first,first+len) 里面放着第0号 permutation，排列的每个元素都不一样
  //返回排列的编号
        unsigned int factorial[21] = {
1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,1932053504,1278945280,
2004310016,2004189184,4006445056,3396534272,109641728,2192834560 };
        bool used[21] = {0};
        int perInt[21]; //要转换成 [0,len-1] 的整数的排列
        for( int i = 0;i < len; ++i )
                for( int j = 0; j < len; ++j )  {
                        if( * ( permutation + i )  == * (first+j)) {
                                perInt[i] = j;
                                break;
                        }
                }
        unsigned int num = 0;
        for( int i = 0;i < len; ++ i ) {
                unsigned int n = 0;
                for( int j = 0; j < perInt[i]; ++ j) {
                        if(! used[j] )
                                ++n;
                }
```

```
                num += n * factorial[len-i-1];
                used[perInt[i]] = true;
        }
        return num;

}
template <class T>
void GenPermutationByNum(T * first, T * permutation,int len, unsigned int No)
//根据排列编号，生成排列
{ //[first,first+len) 里面放着第0号 permutation,，排列的每个元素都不一样
        unsigned int factorial[21] = {
1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,1932053504,1278945280,
                         2004310016,2004189184,4006445056,3396534272,109641728,2192834560
};
        bool used[21] = {0};
        int perInt[21]; //要转换成 [0,len-1] 的整数的排列
        for(int i = 0;i < len; ++ i ) {
                unsigned int tmp;
                int n = 0;
                int j;
                for( j = 0; j < len; ++j ) {
                        if( !used[j] ) {
                                if( factorial[len - i - 1] >= No+1)
                                        break;
```

```
                        else
                                No -= factorial[len - i - 1];
                }
        }
        perInt[i] = j;
        used[j] = true;
    }
    for( int i = 0;i < len; ++i )
        * ( permutation + i ) = * ( first + perInt[i]);
}
int StrStatusToIntStatus( const char * strStatus)
{
    return GetPermutationNum( "012345678",strStatus,9);
}
void IntStatusToStrStatus( int n, char * strStatus)
{
    GenPermutationByNum((char*)"012345678",strStatus,9,n);
}
int MyAbs(int a) {
    return a >= 0?a:-a;
}
```

```
int h(char * status)
{//h的值是不在位的数字到它该呆的位置的曼哈顿距离之和
		int sum  = 0;
		for( int i = 0;i < 9; ++i ) {
			if( status[i] != '0') {
				int rightx = ( status[i] - '1' )/3;
				int righty = ( status[i] - '1' )%3;
			int nowx = i / 3;
				int nowy = i % 3;
				sum += MyAbs(rightx - nowx) + MyAbs(righty-nowy);
			}
		}
		return sum;
}
int NewStatus( int nStatus, char cMove) {
//求从nStatus经过 cMove 移动后得到的新状态。若移动不可行则返回-1
		char szTmp[20];
		int nZeroPos;
		IntStatusToStrStatus(nStatus,szTmp);
		for( int i = 0;i < 9; ++ i )
			if( szTmp[i] == '0' ) {
				nZeroPos = i;
				break;
			} //返回空格的位置
```

```
        switch( cMove) {
                case 'u': if( nZeroPos - 3 < 0 )  return -1; //空格在第一行
                        else {          szTmp[nZeroPos] = szTmp[nZeroPos - 3];
                                        szTmp[nZeroPos - 3] = '0';              }
                        break;
                case 'd':       if( nZeroPos + 3 > 8 )  return -1; //空格在第三行
                        else {     szTmp[nZeroPos] = szTmp[nZeroPos + 3];
                                        szTmp[nZeroPos + 3] = '0';              }
                        break;
                case 'l':       if( nZeroPos % 3 == 0)  return -1; //空格在第一列
                        else {          szTmp[nZeroPos] = szTmp[nZeroPos -1];
                                        szTmp[nZeroPos -1 ] = '0';              }
                        break;
                case 'r':       if( nZeroPos % 3 == 2)  return -1; //空格在第三列
                        else {          szTmp[nZeroPos] = szTmp[nZeroPos + 1];
                                        szTmp[nZeroPos + 1 ] = '0';             }
                        break;
        }
        return StrStatusToIntStatus(szTmp);
}
```

```
Node AStar(int nStatus){
		open.clear();			closed.clear();			inOpen.reset();			inClosed.reset();
		char strStatus[20];
		IntStatusToStrStatus(nStatus,strStatus);
		Node nd;	nd.status = nStatus;		nd.parent = -1;
		nd.g = 0;	nd.move = -1;			nd.h = h(strStatus);
		nd.f = nd.g + nd.h;
		open.insert(nd);
		inOpen.set(nStatus,true);
		openIdx[nStatus] = open.begin();
		while ( !open.empty()) { //队列不为空
			nd = * open.begin();
			open.erase(open.begin());
			inOpen.set(nd.status,false);
			multiset<Node>::iterator p = closed.insert(nd);
			inClosed.set(nd.status,true);
			closedIdx[nd.status] = p;
			if( nd.status == nGoalStatus )
				return nd;
			for( int i = 0;i < 4;i ++ ) { //尝试4种移动
				Node newNd;
				newNd.status = NewStatus(nd.status,sz4Moves[i]);
				if( newNd.status == -1 )	continue; //不可移，试下一种
				IntStatusToStrStatus(newNd.status,strStatus);
```

```
newNd.g = nd.g + 1;
newNd.h = h(strStatus);
newNd.f = newNd.g + newNd.h;
newNd.parent = nd.status;
newNd.move = sz4Moves[i];
if( inOpen[newNd.status ]) {
        Node tmp = * openIdx[newNd.status];
        if( tmp.f > newNd.f ) {
                open.erase(openIdx[newNd.status]);
                p = open.insert(newNd);
                openIdx[newNd.status] = p;
        }
}
else if( inClosed[newNd.status]) {
        Node tmp = * closedIdx[newNd.status];
        if( tmp.f > newNd.f ) {
                closed.erase(closedIdx[newNd.status]);
                inClosed.set(newNd.status,false);
                p = open.insert(newNd);
                openIdx[newNd.status] = p;
                inOpen.set(newNd.status,true);
        }
}
```

```
                        else {
                                p = open.insert(newNd);
                                openIdx[newNd.status] = p;
                                inOpen.set(newNd.status,true);
                        }
                }
        }
        nd.status = -1;
        return nd;
}
int main(){
        nGoalStatus = StrStatusToIntStatus("123456780");
        char szLine[50];  char szLine2[20];
        while( cin.getline(szLine,48) ) {
                int i,j;
                for( i = 0, j = 0; szLine[i]; i ++ ) {
                        if( szLine[i] != ' ' ) {
                                if( szLine[i] == 'x' )  szLine2[j++] = '0';
                                else  szLine2[j++] = szLine[i];
                        }
                }
                szLine2[j] = 0;
                int sumGoal = 0;
```

```
for( int i = 0;i < 8; ++i )
        sumGoal += i -1;
int sumOri = 0;
for( int i = 0;i < 9 ; ++i ) {
        if( szLine2[i] == '0‘)                  continue;
        for( int j = 0; j < i; ++j ) {
                if( szLine2[j] < szLine2[i] && szLine2[j] != '0' )
                        sumOri ++;
        }
}
if( sumOri %2 != sumGoal %2 ) {
        cout << "unsolvable" << endl;
        continue;
}
Node nd = AStar(StrStatusToIntStatus(szLine2));
if( nd.status != -1 ) {
        int nMoves = 0;
        multiset<Node>::iterator pos;
        pos = closedIdx[nd.status];
        do {    if( pos->move != -1 ) {
                        szResult[nMoves++] = pos->move;
                        pos = closedIdx[ pos->parent];
                }
        } while(pos->move != -1);
```

```
                for( int i = nMoves -1; i >= 0; i -- )
                    cout << szResult[i];
                cout << endl;
            }
            else
                cout << "unsolvable" << endl;
        }
        return 0;
}

/*
open 队列是set，按f值排序，用set存放每个节点.删除队头 元素就是删除begin
拿出来以后，放到closed表，更新 inopen标志和inclosed标志，还有closeIdx数组
closed队列 也是set, 按f值排序，
建立一个数组，数组元素就是status在open队列里面的迭代器(假设迭代器不因为队列元素的增删而改变）
,status直接作为数组下标
建立一个数组，数组元素就是status在closed队列里面的迭代器(假设迭代器不因为队列元素的增删而改变）
,status直接作为数组下标

open表里表头拿出来的元素是 goalStatus,然后根据里面记录的 parent,(就是状态），到 closed里面找到其对应元素，然后根据其记录的 parent 和 move，找出问题的解
*/
```

POJ上可用A*算法解决的题：

1376
1324
1084
2449
1475

迭代加深搜索算法

迭代加深搜索算法

- 算法思路
  - 总体上按照深度优先算法方法进行
  - 对搜索深度需要给出一个深度限制dm，当深度达到了dm的时候，如果还没有找到解答，就停止对该分支的搜索，换到另外一个分支继续进行搜索。
  - dm从1开始，从小到大依次增大（因此称为迭代加深）
- 迭代加深搜索是最优的，也是完备的

例：旋转游戏(POJ2286)

在如下图的棋盘中，摆放着8个1，8个2和8个3，每一步你可以沿着A、B、C、D、E、F、G、H任意一个方向移动该字母所指的长块。移出边界的小块会从另一端移进来。如图，最左边的棋盘经过操作A，就会变成中间的棋盘布局，再进行操作C，就会变成右边的棋盘布局。

你需要设法使的最中间的8个格子的数字相同，问最少需要多少步。如何移动？

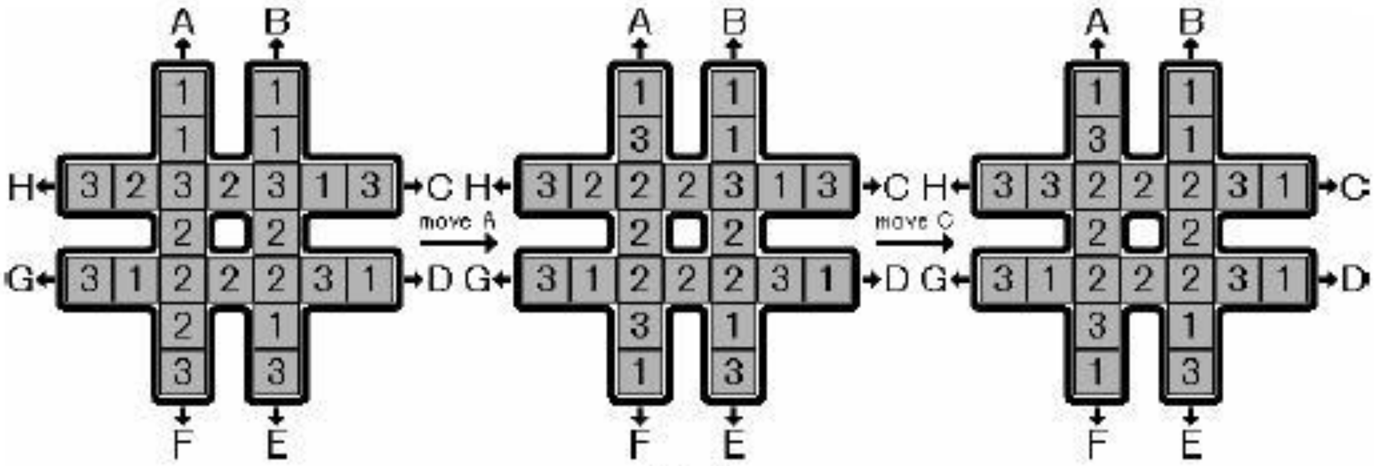


Fig.1

例：旋转游戏(POJ2286)

1. 用迭代加深dfs做，每次在限定一个最大深度的情况下做完全的dfs搜索（除非已经找到解）

2. 如何剪枝？如何选取估价函数？（乐观预测从一个局面到最终界面至少需要的代价）

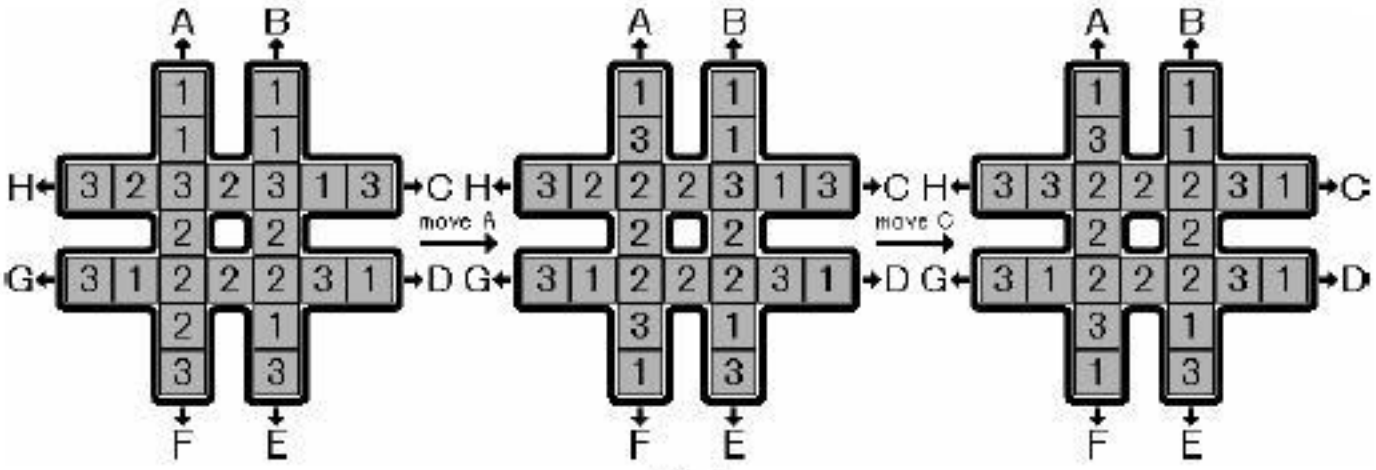


Fig.1

例：旋转游戏(POJ2286)

剪枝：

●每次移动最多只会使得中央区域包含的数字种类减少1种。求出中央区域个数最多的那个数字的个数 n， 要达到中央区域数字都相同，至少需要 8-n次操作，此即估价函数值

●可以用于可行性剪枝。已经移动的步数加上估价函数值，超过本次dfs限定的深度，则剪枝

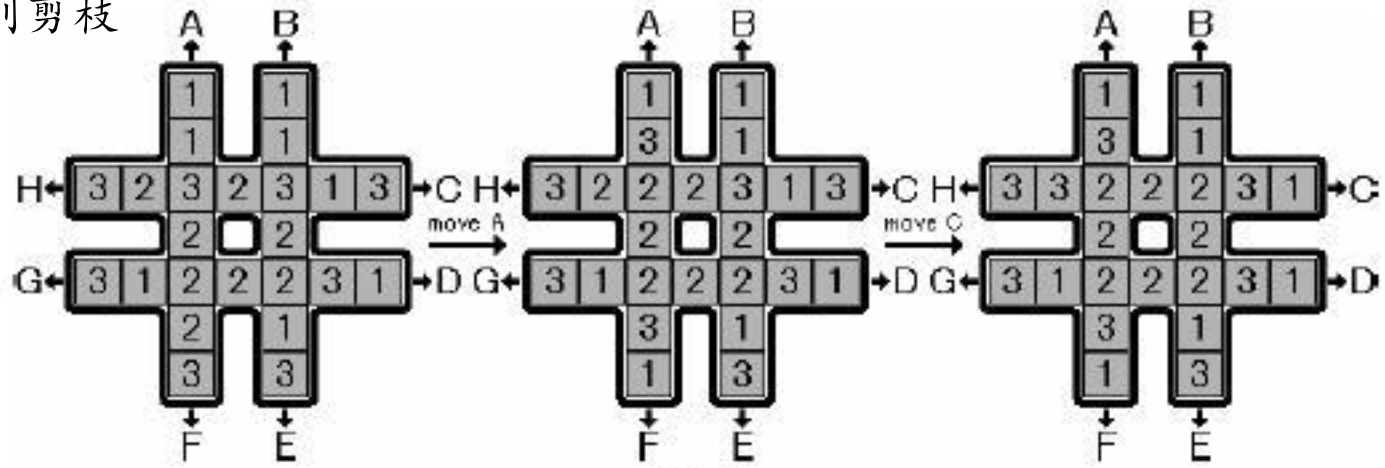


Fig.1

IDA*算法

IDA*：迭代加深的A*算法

- IDA*的基本思路是：首先将初始状态结点的估价函数H值设为阈值maxH，然后进行深度优先搜索，搜索过程中忽略所有H值大于maxH的结点；如果没有找到解，则加大阈值maxH，再重复上述搜索，直到找到一个解。在保证H值的计算满足A*算法的要求(相容，递增）下，可以证明找到的这个解一定是最优解。在程序实现上，IDA* 要比 A* 方便，因为不需要保存结点，不需要判重复，也不需要根据 H值对结点排序，占用空间小。

- 在一般的问题中是这样使用IDA*算法的，当前局面的估价函数值+当前的搜索深度 > 预定义的最大搜索深度时，就停止继续往下搜索。

适用题目： poj2286 the rotation game

IDA*：迭代加深的A*算法

```
1) 初始化深度限制 c= 1
2)  将初始节点入栈; c' = INFINIT
    while (栈不为空)  {
       弹出栈顶元素n
       if (n == goal)
            结束，返回初始节点到n的路径
       else
            for n 的每个子节点 n' {
               if f(n') <= c
                   n'入栈
                 else
                   c=min(c',f(n'))
            }
     }
     if 栈为空且 c' == INFINIT  停止并退出
     if 栈为空且 c' != INFINIT  c= c'; goto 2)
```

c'是本次搜索中发现的f值大于c的节点中，最小的f值

IDA*与A*

- IDA*与A*算法相比，主要的优点是对于内存的需求
  - A*算法需要指数级数量的存储空间，因为没有深度方面的限制
  - 而IDA*算法只有当节点n的所有子节点n’的f(n’)小于限制值c时才扩展它，这样就可以节省大量的内存。
- 另外，IDA*不需要对扩展出的节点按启发值排序。

北京大学 PEKING UNIVERSITY

Alpha-Beta剪枝

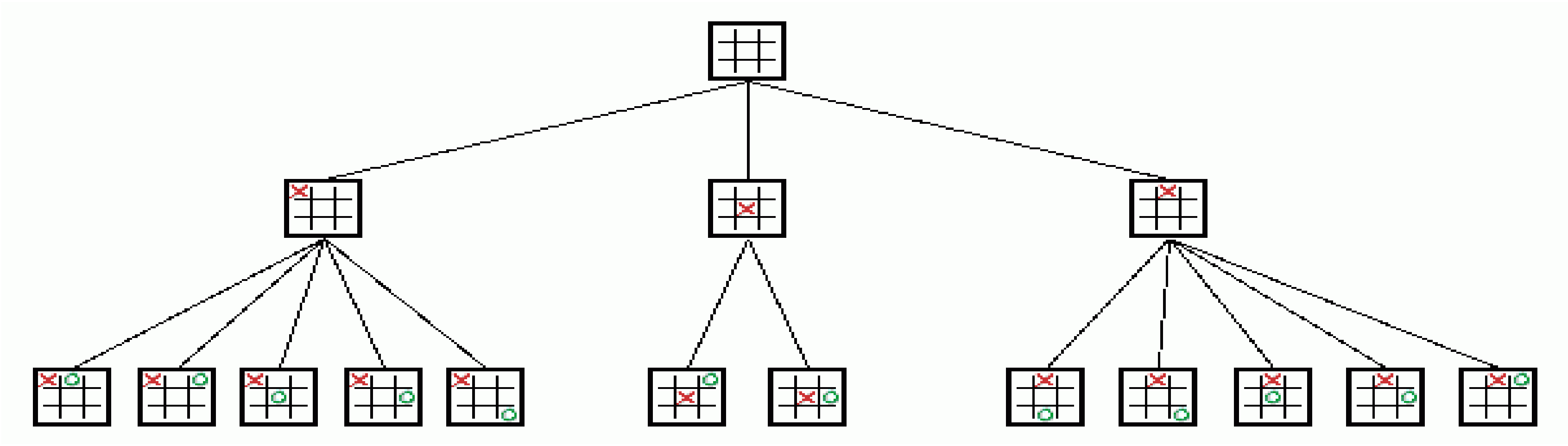
极大极小搜索法

在博弈搜索中，比如：围棋，五子棋，象棋等，结果有三种可能：胜利、失败和平局。

理论上可以穷举所有的走法，这就需要生成整棵博弈树。实际上不可行。因此搜索时可以限定博弈树的深度，到达该深度则不再往下搜，相当于只往前看 n 步。

极大极小搜索法

抢先连成三点一线的一字棋的三层博弈树：



极大极小搜索法

假设：MAX和MIN对弈，轮到MAX走棋了，那么我们会遍历MAX的每一个可能走棋方法，然后对于前面MAX的每一个走棋方法，遍历MIN的每一个走棋方法，然后接着遍历MAX的每一个走棋方法，……直到分出胜负或者达到了搜索深度的限制。若达到搜索深度限制时尚未分出胜负，则根据当前局面的形式，给出一个得分，计算得分的方法被称为估价函数，不同游戏的估价函数如何设计和具体游戏相关。

在搜索树中，轮到MAX走棋的节点即为极大节点，轮到MIN走棋的节点为极小节点。

极大极小搜索法

1） 确定估价值函数，来计算每个棋局节点的估价值。对MAX方有利，估价值为正，对MAX方越有利，估价值越大。对MIN方有利，估价值为负，对MIN方越有利，估价值越小。

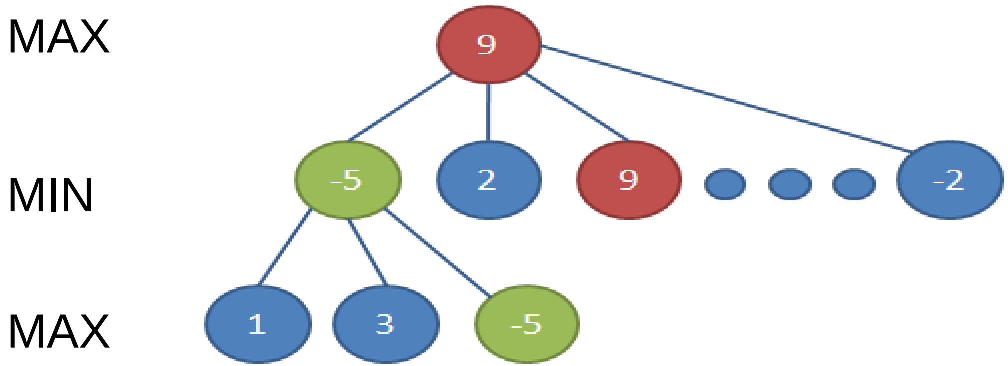
2） 从当前棋局的节点要决定下一步如何走时，以当前棋局节点为根，生成一棵深度为n的搜索树。不妨总是假设当前棋局节点是MAX节点。

3） 用局面估价函数计算出每个叶子节点的估价值

4） 若某个非叶子节点是极大节点，则其估价值为其子节点中估价值最大的那个节点的估价值
若某个非叶子节点是极小节点，则其估价值为其子节点中估价值最小的那个节点的估价值

极大极小搜索法

5) 选当前棋局节点的估价值最大的那个子节点，作为此步行棋的走法。



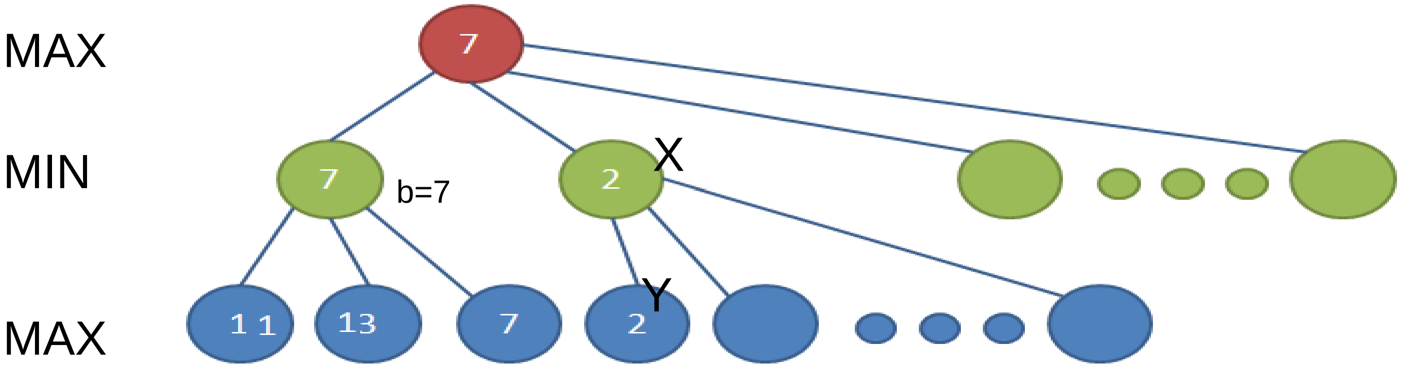
```
function minimax(node, depth) // 指定当前节点和还要搜索的深度
    // 如果胜负已分或者深度为零，使用评估函数返回局面得分
    if node is a terminal node or depth = 0
        return the heuristic value of node
    // 如果轮到对手走棋，即node是极小节点，选择一个得分最小的走法
    if the adversary is to play at node
        let α := +∞
        foreach child of node
            α := min(α, minimax(child, depth-1))
    // 如果轮到自己走棋，是极大节点，选择一个得分最大的走法
    else {we are to play at node}
        let α := -∞
        foreach child of node
            α := max(α, minimax(child, depth-1))
    return α;
```

具体的做法：

```
int MinMax(int depth)   { // 函数的评估都是以MAX方的角度来评估的
  if (SideToMove() == MAX_SIDE)
        return Max(depth);
   else
        return Min(depth);
}
int Max(int depth) {
  int best = -INFINITY;
  if (depth <= 0)   return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
     MakeNextMove();
     val = Min(depth - 1);
     UnmakeMove();
     if (val > best)   best = val;
  }
  return best;   }
```

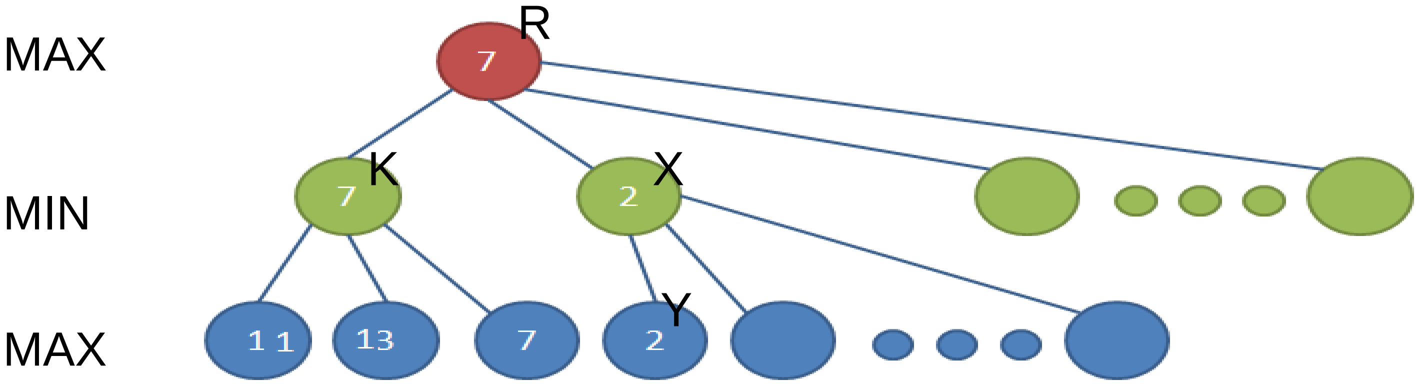
```
int Min(int depth) {
  int best = INFINITY;  // 注意这里不同于“最大”算法
  if (depth <= 0)
        return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Max(depth - 1);
    UnmakeMove();
    if (val < best)     // 注意这里不同于“最大”算法
        best = val;
  }
  return best;
}
```

Alpha－beta剪枝



若结点x是Min节点，其兄弟节点(父节点相同的节点）中，已经求到的最大估价值是b（有些兄弟节点的估价值,可能还没有算出来），那么在对x的子节点进行考查的过程中，如果一旦发现某子节点的估价值 <=b,则不必再考查后面的x的子节点了。

alpha剪枝



当搜索节点X时，若已求得某子节点Y的值为2，因为X是一个极小节点，那么X节点得到的值肯定不大于2。因此X节点的子节点即使都搜索了，X节点值也不会超过2。而节点K的值为7，由于R是一个Max节点，那么R的取值已经可以肯定不会选X的取值了。因此X节点的Y后面子节点可以忽略，即图中第三层没有数字的节点可被忽略。此即为alpha剪枝 ---- 因被剪掉的节点是极大节点。相应的也有beta剪枝，即被剪掉的节点是极小节点。

beta剪枝

若结点x是Max节点，其兄弟节点(父节点相同的节点）中，已经求到的最小估价值是a（有些兄弟节点的估价值,可能还没有算出来）
那么在对x的子节点进行考查的过程中，如果一旦发现某子节点的估价值 >= a,则不必再考查后面的x的子节点了。

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        α := -INF; //负无穷大
        for each child of node
            α := max(α, alphabeta(child, depth - 1, α, β,
                                          not(maximizingPlayer)))
            if β ≤ α  //β是node的兄弟节点到目前为止的最小估价值
                break       // Beta cut-off,剪掉剩下的极小节点
        return α
    else
        β := INF; //无穷大
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β,
                                          not(maximizingPlayer)))
            if β ≤ α  //α是node的兄弟节点到目前为止的最大估价值
                break       // Alpha cut-off, 剪掉剩下的极大节点
        return β
初始调用: alphabeta(origin, depth, -infinity, +infinity, TRUE)
```

注解

●红色字母处，随便写什么值都可以

●在搜到底的情况下，infinity不一定是无穷大 infinity应该是主角赢的那个状态(胜负已分的状态）的估价值，而-infinity应该是主角输的那个状态(胜负已分的状态）的估价值。

Alpha －beta剪枝例题 POJ 1568

给定一个4 * 4 的四子连珠棋盘的局面，现在轮到x先走，问x下在哪里可以必胜

....
.xo.
.ox.
....

Alpha -beta剪枝例题 POJ 1568

在穷尽搜索，而非搜几层就停止的情况下，估价函数只有三种取值，正(inf)，0,负(-inf)。分别代表己方胜，平，负。

必胜：

无论对手(B,o)怎么走，己方(A,x)最后都能找到获胜办法。不是自己无论怎么走，都能获胜。

此步行棋，若能找到一个走法，其对应的节点估价值为inf ,则此为一个必胜的走法。

```
#include<iostream>
#include<cstdio>
#include<map>
#include<cstring>
#include<cmath>
#include<vector>
#include<queue>
#include<algorithm>
#include<set>
using namespace std;
//程序改编自网上博客源代码
#define inf 1<<30  //inf取1也可以
char str[5][5];
int X,Y,chess;
bool check(int x,int y){  //….略   }
//判断一个局面是否是棋局结束的局面

int MinSearch(int x,int y,int alpha);
int MaxSearch(int x,int y,int beta);
```

```
int MinSearch(int x,int y, int alpha)
{ /*本节点是A刚下完, 刚刚下在了(x,y)该轮到B下的节点
MinSearch的返回值 是B最佳走法的估价值。具体办法，枚举本步所有B走法，对每种走法用
MaxSearch求其估价值，返回估价值最小的那个走法(B的选择）的估价值。
参数alpha，表示本步（本节点）的兄弟节点里面，已经算出来的最大估价值。
如果枚举走法的过程中，发现有 MaxSearch的值如ans已经小于等于alpha,那么立即停止
枚举，返回 ans,表示本节点的估价值就是ans
如果 ans < alpha，那么本节点就不会被自己选择 */
        int ans = inf;
        if( check(x,y)) //自己刚走了一步，如果棋局分出胜负，那么定是自己胜利
                return inf;
        if( chess == 16 )
                return 0; //平局
        for( int i = 0;i < 4; ++i)
                for( int j = 0; j < 4; ++ j ) {
                        if( str[i][j] == '.' ) {
                                str[i][j] = 'o'; ++ chess;
                                int tmp = MaxSearch(i,j,ans); //此处的
ans是beta 值，即在本节点的各个子节点中，目前已经找到的最小的 估价值
                                str[i][j] = '.'; -- chess;
                                ans = min(ans,tmp);
```

```
                              if( ans <= alpha ) {
                                        return ans;
                              }
                    }
          }
     return ans;
}
int MaxSearch(int x,int y, int beta)
{/*本节点是B刚下完,下在(x,y),该轮到A下的节点
在MaxSearch中，要返回本步A最佳走法的估价值。具体办法，枚举本步所有走法，对每种走法用 MinSearch求其估价值，返回估价值最大的那个走法的估价值。
beta表示本步的兄弟节点中，已经算出来的最小的估价值。
如果枚举走法的过程中，发现有 MinSearch的值如ans已经大于等于 beta,那么立即停止枚举，返回 ans,表示本节点的估价值就是ans
如果 ans > beta，那么本节点就不会被B选择 */
     int ans = -inf; //本节点展开的各子节点中，最大的估价函数值
     if( check(x,y))
          return -inf;
```

```
        if( chess == 16 )
                return 0;
        for( int i = 0;i < 4; ++ i)
                for( int j = 0; j < 4; ++ j ) {
                        if( str[i][j] == '.' ) {
                                str[i][j] = 'x'; ++chess;
                                int tmp  = MinSearch(i,j,ans );
//ans是 beta值，即在本节点的各个子节点中，目前找到的最大的估价值
                                str[i][j] = '.'; --chess;
                                ans = max(tmp,ans);
                                if( ans >= beta )
                                        return ans;

                        }
                }
        return ans;
}
```

```
bool Solve()
{
	int alpha = -inf;
	for(int i = 0;i < 4; ++ i)
		for( int j = 0; j < 4; ++j ) {
			if( str[i][j] == '.' ) {
				str[i][j] = 'x'; ++ chess;
				int tmp = MinSearch(i,j,alpha);
				str[i][j] = '.'; -- chess;
				if( tmp == inf ) {
					X = i;
					Y = j;
					return true;
				}
			}
		}
	return false;
}
```

```
int main(){
        char ch[5];
        while(scanf("%s",ch)!=EOF&&ch[0]!='$'){
                chess=0;
                for(int i=0;i<4;i++){
                        scanf("%s",str[i]);
                        for(int j=0;j<4;j++)
                                chess+=str[i][j]!='.';
                }
                if(chess<=4){//这一步直接从2S+到0ms
                        printf("#####\n");
                        continue;
                }
                if(Solve()) printf("(%d,%d)\n",X,Y);
                else printf(“#####\n”); //无必胜走法
        }
        return 0;
}
```

例：2012金华邀请赛Problem F. Chess

给定4*4棋盘上四子棋的局面（至少已经有6个子），现在轮到Bob下。

Bob可能执 X 也可能执 O,

问Bob是否能做到必胜(想赢就赢),

或问Bob是否能做到想败就一定败

或问Bob是否能做到想和就一定和

例：2012金华邀请赛Problem F. Chess

误区：采用和POJ1568一样的估价函数定义来做此题。

O必能败，等价于X必胜吗？

O必能败 等价于 X 不是必败，且不是必和 吗？

O必能和 等价于 X 不是必胜，也不是必败 吗？

例：2012金华邀请赛Problem F. Chess

误区:采用和POJ1568一样的估价函数定义来做此题。

O必能败，等价于 X必胜吗？否

O必能败 等价于 X 不是必败，且不是必和 吗？ 否

O必能和 等价于 X 不是必胜，也不是必败 吗？否

例：2012金华邀请赛Problem F. Chess

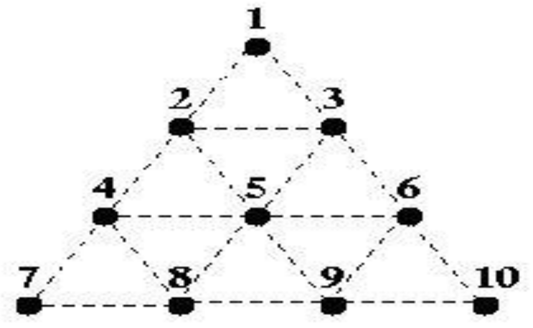
误区:采用和POJ1568一样的估价函数定义来做此题。

O 必能如何，无法等价于 X 必能或不必能怎么样的组合

所以，应根据Bob是X还是O,以及Bob想怎么样，改变估价函数的定义。比如，若Bob是O,且Bob想输，那么和棋终局或是Bob赢的终局，估价函数值都是 –inf，Bob输的终局，估价函数值都是 inf

例：三角形大战(POJ1085)

- 两个游戏者轮流填充左边的虚线三角形。每次只能填充一条短边。
- 若某游戏者填充一条短边之后组成了一个小三角形，则该游戏者“拥有”这个三角形，并且可以继续填充。
- 当所有边都被填充之后，拥有三角形数目多的游戏者获胜。
- 给定局面，问谁能赢



- 博弈的思路
  - alpha-beta剪枝
  - 估价函数：A拥有的三角形数 – B拥有的三角形数