

魔塔游戏AI设计的基本思路与实现

队名：作队

成员：

姓名	学号
张孝帅	1500010716
马赞彭	1400012630
叶天	1500012747
叶志晟	1500012804

魔塔游戏AI设计的基本要求

魔塔游戏是一个较为著名的策略类RPG小游戏，通过操纵一名勇士，拾取给定地图上的包括血瓶、钥匙等在内的特殊道具，主动与怪物进行战斗和使用钥匙进行开门，最终目的是战胜整座塔的boss。

经过分析，我们觉得需要额外注意处理的地方有如下几点：

- 钥匙是具有很大作用的道具，当没有并且已经无法继续获得足够钥匙时游戏可能会无法继续
- 不同的门需要对应颜色的钥匙方可开启，需要对不同的钥匙分开记录
- 勇士可以通过选择一些增益效果来弥补经过这条路所受到的损失
- 完全连通的部分内部的周游是没有代价的，所以可以将部分类似的块合并处理以简化路径选择过程

根据评判标准，可能存在路线无法战胜boss从而选择较为优秀方法的情况，所以我们考虑使用将地图抽象为类似树结构的方式进行搜索，在限制的时间与空间内停止搜索返回当前的最优解。

类似于以前编写过的黑白棋、Pacman游戏AI，在搜索上继承了深度优先搜索并回溯，在同一层上利用广度优先搜索来对图进行重构，更加具体的内容将会在后文详细介绍。

我们还在游戏过程中总结出这样一些规律。

例如，我们在很大几率上遇到了某个连通块，它被墙壁包裹着，只有一个门（可能包含门附近的怪物），但是该连通块内部又有着较为丰富的增益效果。这种情况下我们应当进行评估，然后大多情况下选择进入该块的决定。在实际AI设计中，我们也希望这种思想能够在AI中实现，所以在设计评估函数时我们对这种情况进行了特殊的处理：通过内部增益块的权重的合理设置来抵消由于进入这个过程所引发的减益效果以及进入一个连通分量减少区域带来的负面影响。

简单来说，魔塔游戏AI的设计确实是一项艰巨而富有挑战性的任务，我们在进行建模、设计、具体实现以及参数调整过程中，也感受到了不同阶段的各种困难，在努力解决中也收获了很多。同时，非常感谢助教可视化程序的提供，这为我们在人脑选择可能较优路径与AI的转化之间提供了非常高效而且方便的工具，极大的加速了AI的设计与调试这一阶段的进度。

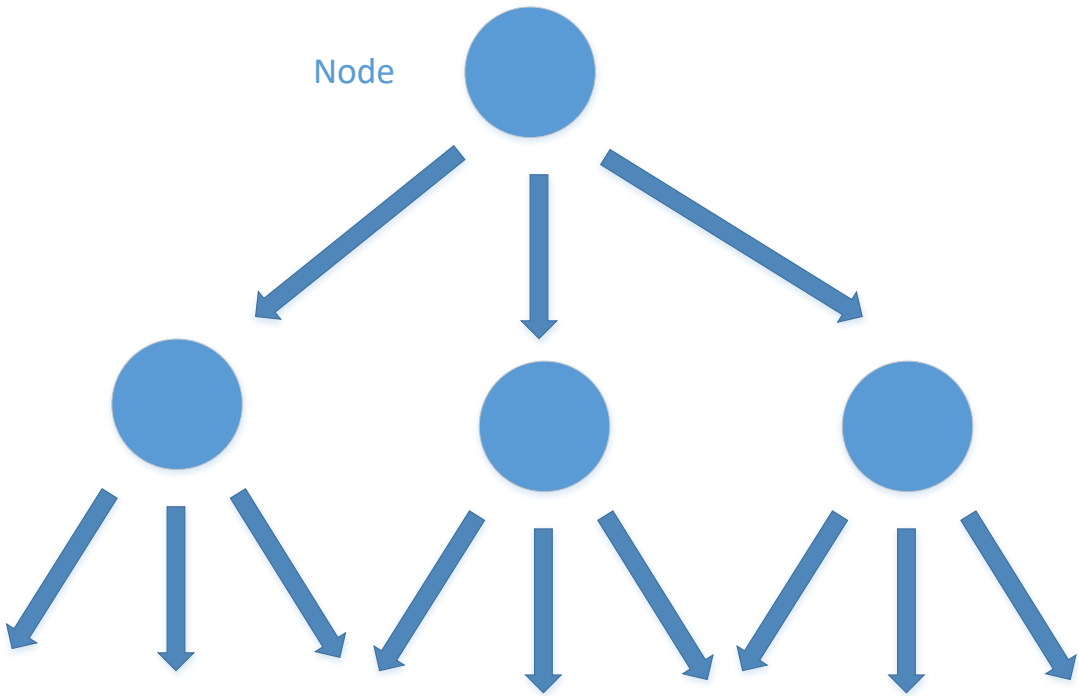
接下来我们将根据设计的时间顺序，就具体过程中遇到的问题与可能的解决方案展开叙述。

魔塔游戏AI设计的过程

魔塔游戏AI设计的建模

首先最先完成的部分是魔塔地图部分的表达，这一部分是最为简单的部分。我们直接将勇士、怪物、塔与塔中的物体分别抽象为Player、Monster、Tower以及枚举类MapObj，并封装了getDamage函数，也为了输入的整洁性对流进行了重载，这是最开始对魔塔地图的表达。

对于图的表达是很轻松的完成了，但是最为关键的搜索部分应该采用怎样的数据结构的讨论花费了我们不少时间，我们在将搜索结构抽象为图还是树产生了比较大的分歧，最终我们选择了图作为每层搜索中的基础数据结构类型，在一层一层向下搜索与回溯过程中，利用树的性质，较为方便整洁的组织起了每一层之间的关系，大致结构如图：



我们考虑每一个Node，由于是搜索的过程，每一个Node都应该能够等价的表达当前整个图的状态，我们使用了这些成员来表达：

```

/* 魔塔重构图节点结构 */
struct GraphNode
{
    bool valid;           /* 访问该节点后将valid设为false */
    Position pos;         /* 该节点的坐标 */
    MapObj type;          /* 该节点类型（门或怪物） */
    vector<GraphNode*> next; /* 子节点列表 */
    vector<MapObj> obj;    /* 节点物品列表 */
    int blockCount;       /* 该节点增加的连通块计数 */
};

/* 状态转移结构 */
struct Status
{
    GraphNode* head;
    PlayerInfo player;
};

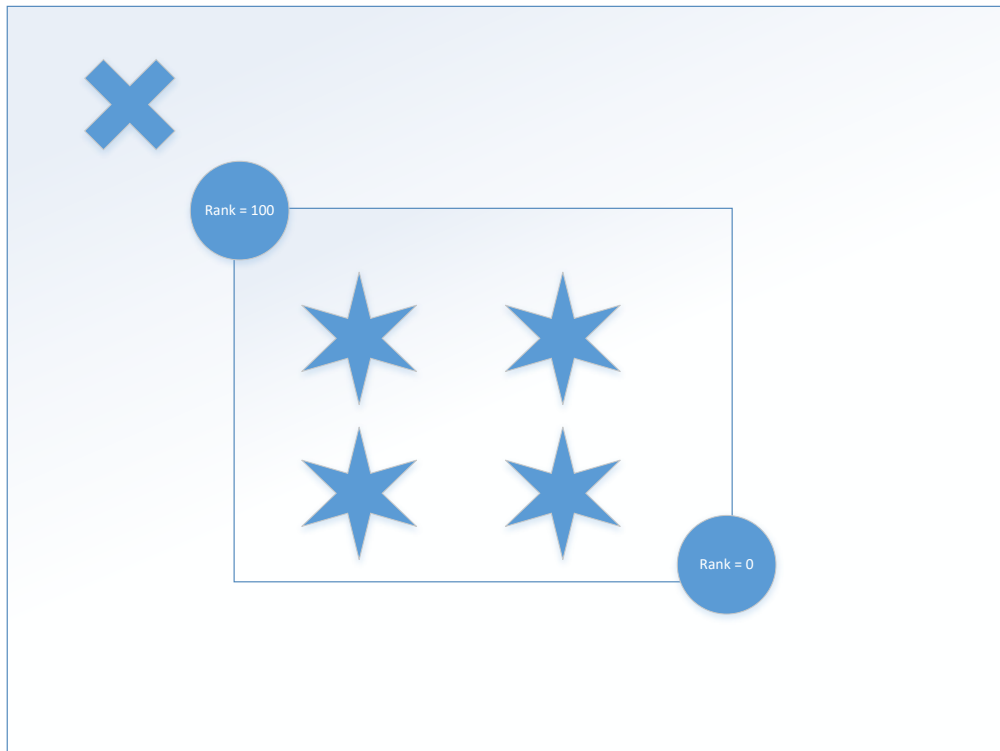
```

每个status利用一个Status的实例化对象等价表达，每个状态下，我们对局面所呈现的图进行节点重构，每一个静态状态地图中的每一个门或怪物节点都是树的组成部分。关于为什么只对门和怪物建树我们也经过了一番讨论，主要理由是大多数情况下，门和怪物节点才应该是决定连通分支数目的因素，将他们作为子节点插入到当前节点中，有利于使搜索过程更加高效。

AI搜索算法的建立与完善

一开始我们先完成了一个较为简单的基于bfs的算法，采用队列，对当前节点所在整个封闭区域内进行搜索，以此来找出当前节点的每个next指针的位置，这里的next既包含GraphNode也包含MapObj，可以同时同步完成节点的添加过程，利用vector也是基于其高效的插入特性。

这个方案看似非常符合常理，但是在完成后我们突然想到了这样的情况，如图所示：



图中十字叉位置表达当前搜索到的节点，圆形表示一个怪物，其上rank值表示其生命值，也就是勇士攻击它而获得的损益效果，考虑离勇士很近的怪物拥有很高的生命值的情况，中间的星型表示许多增益效果。

如果采用上文所描述的广度优先搜索的方式进行搜索，我们会自然的将中间的增益效果归为高rank的怪物所有，所以我们会评估增益效果与这个高rank的损益，从而做出要么不取要么通过高rank怪物得到增益，而显然地，我们本可以经过那个稍远的怪物获得这些增益，这样的代价明显是更小的，但是上文描述的广度优先搜索无法对这种情况做出很好的计算结果，而在实际中这种情况出现的概率并不能达到被忽略的地步，所以我们只好做出修改核心搜索算法的选择。

我们尝试使用图染色的基本思路来处理这里的问题，因为在这个情景中，连通这一条件与增益的获得有着必然的联系。我们最终针对单层局面的搜索函数如下：

```

static void colorize(const Tower& mogleTower, int x, int y,
                    const int color, int been[MAP_LENGTH][MAP_WIDTH])
{
    if (been[x][y] != 0)
        return;
    auto& map = mogleTower.mapContent;
    been[x][y] = color;
    for (int i = 0; i < 4; i++)
    {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (!isInRange(nx, ny))
            continue;
        if (map[nx][ny] == wall)
            continue;
        if (isMonster(mogleTower, nx, ny))
            continue;
        if (isDoor(mogleTower, nx, ny))
            continue;
        colorize(mogleTower, nx, ny, color, been);
    }
}

static void traverseMap(const Tower& mogleTower,
                       GraphNode* cur, int been[MAP_LENGTH][MAP_WIDTH])
{
    int color = 0;
    auto& map = mogleTower.mapContent;
    for (int i = 0; i < MAP_LENGTH; ++i)
        for (int j = 0; j < MAP_WIDTH; ++j)
        {
            if (map[i][j] == wall)
                continue;
            if (been[i][j] != 0)
                continue;
            if (isMonster(mogleTower, i, j) || isDoor(mogleTower, i, j))
            {
                been[i][j] = ++color;
                continue;
            }
            colorize(mogleTower, i, j, ++color, been);
        }
}

```

通过两种不同类型的染色形式，对当前的局面进行了较为准确而快速的量化，最终对于单层中的连通分量，我们取得了较好的计算方式。

通过上述操作，我们得到了染色后的地图，也就是代码中的been数组，然后根据been数组建立上文提到的后继节点关系，具体代码实现在buildGraph函数中，唯一需要指出的是，为了搜索的方便，将容器由vector改为set，具体代码因较长不在此列出。

到这里我们得到了每个status对应的等价图关系，这为我们深度优先搜索以及回溯提供了基础。

深度优先搜索算法的具体实现

根据深度优先搜索算法的实现过程，大致可以分成这样几个部分：

```
if(depth == MAX_DEPTH)
    return eval;

backup();
for(every_next_status)
{
    try();
    if(curVal > maxVal)
        maxVal = curVal;
    restore();
}
```

在我们进行上述构造并搜索时，在可接受的时间范围内对第一个输入大概只能搜索到4-5层，虽然这样可以解决前3个例子，但是最终只能达到4-5层并不是一个非常好的深度。为了找到代码中最为耗时的部分，我们使用**Visual Studio**的调试工具对代码进行了性能分析，最终发现**backup**与**restore**部分占用了过多的时间，究其原因，我们推测是在复制的时候对于指针的处理过于复杂。

为了能够让搜索算法的效果有质的提升，必须要对指针复制中大量的资源占用做出优化，我们无奈之中只能对整个图节点结构进行重构，重构的目的是优化指针操作，我们改用记录**index**值的方式，并因此修改了几乎整个项目，这次重构是整个过程中最为耗时的部分。

```

/* 魔塔重构图节点结构 */
class GraphNode
{
private:
    int index;           /* 该节点索引值（最初被染的颜色） */
    Position pos;        /* 该节点的坐标 */
    MapObj type;         /* 该节点类型 */

public:
    bool empty;          /* 访问该节点后将empty设为true */
    int blockCount;       /* 该节点增加的连通块计数 */
    set<int> adj;         /* 邻接节点索引列表 */
    vector<MapObj> obj;   /* 节点物品列表 */
    Status* fatherStat;

    //GraphNode() { empty = true; }
    GraphNode(Status* father = nullptr): empty(true), fatherStat(father) {}
    GraphNode(Status* father, int _idx, int _x, int _y, MapObj _type):
        fatherStat(father), index(_idx), pos(_x, _y), type(_type), empty(false), blockCount(1) {}
    MapObj getType()const { return type; }
    int getIndex()const { return index; }
    const Position& getPos()const { return pos; }
    bool operator==(const GraphNode& o)const;
};

/* 状态转移结构 */
struct Status
{
    int curIdx;
    PlayerInfo player;

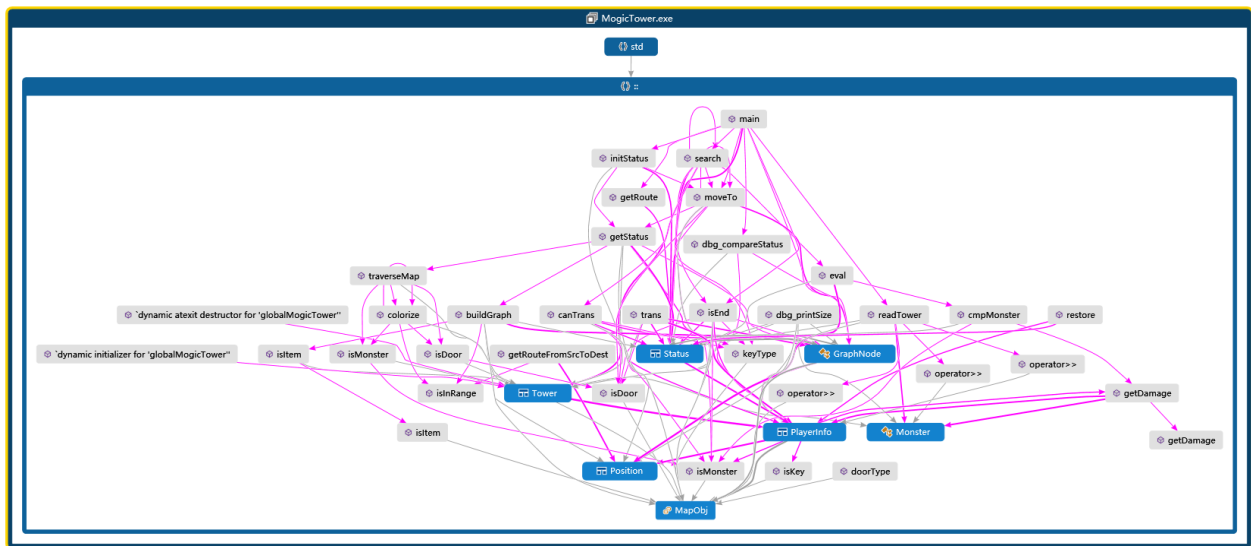
    vector<GraphNode> nodeContainer;

    Status(): curIdx(0), player(), nodeContainer() {}
    Status(const Status& other);
    const Status& operator=(const Status& other);
    GraphNode& getNode(int index = 0) { return index ? nodeContainer[index] :
nodeContainer[curIdx]; }
    const GraphNode& getNode(int index = 0)const { return index ? nodeContainer[index] :
nodeContainer[curIdx]; }
    GraphNode* getNodePtr(int index = 0) { return index ? &nodeContainer[index] :
&nodeContainer[curIdx]; }
};

```

重构后性能有了巨大的提升，若以第一个输入为例，每次搜索8层可以在1秒内出解，达到了预期要求。

最终我们主函数逻辑可作如下图，从图来看还是很错综复杂的，应该有优化的空间：



AI设计过程所遇到困难及解决方案

状态的表达与快速操作

在一开始构建`status`类的目的是用来完整的等价表达当前的状态，以期能够在尽量减少图的重构操作的同时保证搜索的有效与准确，该如何完整的表达状态是我们遇到的第一个问题。本质上来说，这跟上学期程序设计实习中提到的深拷贝与浅拷贝有类似之处。经过了上述的思考与讨论过程，我们最终选用了存`index`的方案，达到了速度与检索方便程度的统一。

评估函数与终止判断

评估函数是对抗类AI设计的核心内容，也是竞争力的体现，我们对于每一个独立的`status`进行了评估，评估考虑的对象以当前状态的人物为主，考虑了包括生命值、攻击力、防御力、钥匙等在内的许多元素。计划中加入一些能表达此处连通情况的权值，但是不能保证一定能对估值有正影响，所以暂时没有考虑加入进来。

每次判断是否结束时，我们当前采用的方案是判断在当前连通分支中，是否有可以继续占据的怪物或者门，但是这样的判断是有问题的。最关键的问题是，如果当前可以通过占据一个怪物来拓展连通分支，并且这一条新的连通分支背后可能意味着更多的收益，从而使得最终收益增大，这是无法通过判断结束来解决的。同时，判断是否结束的函数是每一层搜索的每一步都需要调用一次，所以时间消耗不允许过大。最终我们仅仅在其中实现遍历所有邻接节点，速度有所提升。

记录并输出路径

这一部分的调试也占用了很多时间，原因在于在某一连通块之中如何进行遍历，因为我们所记录的信息并不足以进行一次有序的遍历。对于每个status，我们只记录了与其连通的门或者怪物，而对于连通区域内部所有增益效果块均可以无损耗获取。所以我们先写了一个求两点之间路径的函数，通过宽搜实现。接着枚举连通块内增益点，按距离长短依次访问，并标记，最终得到一条路径。

AI的重构

由于之前AI对于过于复杂的图也没有采取剪枝的操作，所以尤其是对于类似后两张图这种较为复杂的图来说并没有得到较为优秀的解，我们又对AI进行了重构。

首先为了提升运算速度，减少搜索时间与等待时间，我们添加了多线程计算，可以极大地提高CPU的利用率，从而大幅度缩短运行时间。

为了避免显式根据input内容选择搜索深度，我们加入了AI搜索深度自动控制的方法，根据局面初始化默认深度（评价邻接状况），每一次搜索使用迭代加深策略，后续可用哈希等方法利用之前搜索结果。

同时为了使得运行过程中等待过程中输出人性化，我们考虑在#define DEBUG时输出搜索过程，即你能够看到每次选择了那个点作为下一次到达的目标，同时在屏幕上得到到该目标的路径，所以等待过程也是十分有趣的。我们也建议使用Visual Studio编译的时候使用本地windows调试器观察CPU占用状态。

总而言之，新的AI.cpp会让您耳目一新，同时也达到了远超之前的效果。

AI性能分析





































样例号	初始最大邻接数 ma	初始平均邻接数 aac	初始节点计数 nc	适当搜索层数 md	最终结果	耗时 (windows cl15)	耗时(ubuntu g++6)
1	4	2	33	11	KO, HP = 3	188.922s	105.05s
2	3	1.95455	44	14	KO, HP = 9	135.946s	89.265s
3	4	1.95918	49	8	KO, HP = 395	106.257s	104.177s
4	5	2.20408	49	7	KO, HP = 1034	210.835s	201.736s
5	5	2.2	60	6	KO, HP = 129	238.418s	296.799s
6	6	2.25	48	10	KO, HP = 54	209.707s	213.246s
7	5	2.19672	61	7	剩余2, HP = 105	284.413s	228.43s

AI设计过程中的亮点

版本控制工具的使用

AI设计大作业的完成是一项相互合作，各取所长的过程，同时由于客观时间原因，不太可能抽出很多时间大家坐在一起同时完成某项功能，只能讲其按模块分开完成，这对各部分与接口之间的统一性提出了要求。我们先对整体的几个表达图与搜索过程的关键定义与函数进行了讨论，得出了较为统一的意见，在后来的实现过程中尽量减少对其的更改，在无法避免的情况下再进行大型重构。

为了便于版本控制与协作，我们利用了Github作为同步与版本控制工具，下图是某个集中开发的周末的commit的记录截图：

	增加一个节点索引参数（颜色） BuriedJet committed 28 minutes ago		62fc476	
	基本完成Player类（acquire拾取函数和fight对战函数），完成trans状态转移函数的部分前提函数，备份初始图。初始图的第一... BuriedJet committed an hour ago		95d264e	
	加入buildGraph的部分描述 yzs981130 committed 17 hours ago		45b8f8e	
	README完成到搜索算法部分 yzs981130 committed 17 hours ago		4b29688	
	从been数组建立图，头结点head；GraphNode类的next成员类型改为set<GraphNode *> yetiancn committed 17 hours ago		7ee8928	
	更换一种建立图的方法 BuriedJet committed 19 hours ago		00fe70e	
	合并和Jet提交的冲突 mzpyeah committed 21 hours ago		fcd87a1	
	Marco's extraordinary commitments -- mzpyeah committed 21 hours ago		8d7a224	
	Nothing BuriedJet committed 21 hours ago		9a29de8	
	Merge branch 'master' of https://github.com/BuriedJet/MogicTower BuriedJet committed 22 hours ago		b5f4327	
	尝试实现search基本框架 BuriedJet committed 22 hours ago		d3de87c	
	在node中增加player指针，增加对当前局面player本身性质的评估函数 yzs981130 committed 23 hours ago		f488808	

通过git的使用，方便了组同学内部的交流，也便于出现巨大问题时迅速恢复。

对工程项目文件组织形式的优化

为了更好的体现C++面向对象的特点，增加工程项目文件的可读性，我们对于项目文件的组织进行了调整。

我们首先将所有基础类的定义从主函数中抽离出去，并以*.h的形式声明，在对应的*.cpp中实现函数的定义。接着对于一些常用的函数进行了重新包装以提高重用性。

姓名	学号	对高效的C++特性的利用	占比
----	----	--------------	----

我们也力图通过尝试C++的特性来实现部分内容的简化。尽管这些内容可能在课堂上不会作为要点教授，但是我们觉得这些的学习与使用，才是本门课程中实习二字的意义所在。

具体而言，在我们的项目文件中，我们使用了：多线程、auto、基于范围的for、大量的STL等等。

严谨的调试

为了提高调试的效率，尽可能缩短达到理想效果的时间，我们单独定义了一系列供调试使用的宏、函数，加入了一系列异常处理，从而更好的处理了一些可能出现的偶然错误。

其他说明

参考资料

除了魔塔本身游戏的基本规则外，没有参考任何资料。

对魔塔游戏的理解全部来源于我们手动玩魔塔游戏的经验，游戏中的算法均为独立构思，未参考任何代码。

使用方法

- 将要输入的图以input.txt为文件名放在当前目录下，由于使用了大量的STL，编译时请选择x64 release模式，并建议开启全部优化，已经将这些推荐开启的项目写入Visual Studio工程文件中，默认打开即启用。注意：务必使用release模式编译，程序会将输出写入当前目录下的output.txt中。
- 为方便助教评测，我们也编写了CMakeLists.txt，在linux平台下直接cmake即可。
- 每组出解的时间不会超过5分钟，在输出结束前无提示，输出完成后会自动退出。
- 如有问题烦请联系我们，十分感谢！

组内分工

姓名	学号	大致完成内容	占比
张孝帅	1500010716	算法设计、深度搜索部分主要函数	31%
马赞彭	1400012630	估值函数以及调参	23%
叶天	1500012747	路径搜索函数	23%

姓名	学号	大致完成内容	占比
叶志晟	1500012804	Readme编写、基本IO与模型	23%

最后再次感谢助教提供的可供可视化调试的魔塔程序！