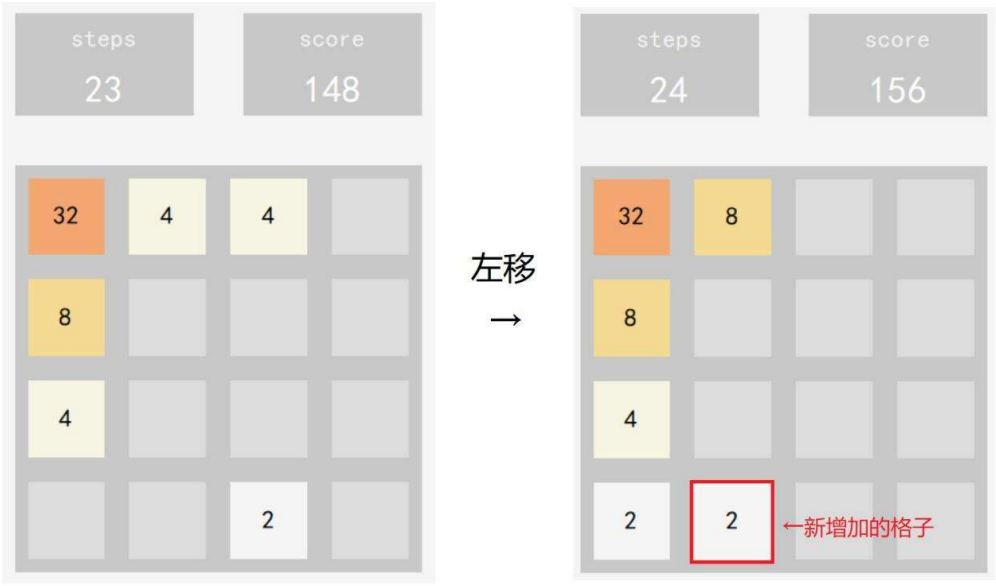
2048 游戏算法的进一步研究及 C++语言实现

一、 2048 游戏简介

“2048”是一款流行的数字游戏。整个游戏在一个有 4×4 格子（block）的棋盘（board）中进行，每个格子上的数字必定为 2 的 n 次方或 0（0 不显示）。玩家通过棋盘上数字的移动（move）进行游戏。当进行移动时，棋盘上的所有非 0 数字会向同一方向移动，并且，如果移动方向上的两个相邻数字相同，则会发生合并（merge），此时合并后的数字大小翻一倍。只要每次做出移动操作后棋盘上非零数的位置或大小发生变化，则该移动是有效的（valid），此时系统会从剩余的空格子中以均匀概率选中一个，并以 9：1 的概率填入 2 或 4。当没有空格子且棋盘上每两个数都不相等时，所有的移动操作均是无效的，此时游戏结束。游戏的总分为总共合并的数字之和。



2048 游戏规则简单，但想达到数字 2048，也就是将棋盘中最大的数合并至少 10 次，是相对困难的。正因为该游戏的规则简单，所以不失为人工智能研究的好课题。本文将介绍针对该游戏的解法开展的一系列研究。

二、 2048 游戏建模

2.1 棋盘状态的表示

2048 游戏的棋盘很容易用一个四阶矩阵（4×4 二维数组）表示。所有的移动操作实际上是对数组中元素的操作。棋盘上所有的数字只能是 0 或 2 的 n 次方。所以，为了节省储存空间，可以将棋盘在程序内部以指数形式表示（1，2，3…表示 2，4，8…，0 表示空），只需进行按位位移操作即可方便地得到实际数值。另外，对于一局游戏而言，还应有分数（score）和步数（steps）两个值以表示游戏进度。

2.2 状态转移与步循环

2048 游戏的每一步分为两个过程：

1）棋盘上数字的移动（及合并）；

2）新数生成。

因此，这两个过程将棋盘分成三个状态：始态（initial state，用 s 表示）、过渡态（transition state，用 s'表示）和终态（terminal state，用 s"表示）。在每一步中，棋盘从始态通过状态转移函数 T(s,a)以概率 1 转移至一个确定的、可预测的过渡态，然后再从过渡态以某一概率 P（P 已知且确定）过渡到一个可预测的终态。注意，这里所有可能出现的终态都是可预测的。

2.3 Board 类的建立与原游戏的 C++实现

基于上述讨论，可以建立 Board 类实现棋盘的表示和对棋盘的操作，进而实现游戏。所有以连字符开头的变量和函数均属于私有或受保护类型，不能直接从类的外部调用。所有以大写字母开头的函数在为公开类型，类外可见。

首先，Board 类应具有如下字段：

```
int8 _board[4][4];//棋盘
short _steps = 0;//步数
int  _score = 0;//分数
```

另外，为了实现新数的添加，需要获得棋盘上所有空格子的坐标，故须有以下两个字段：

```
int8 _emptyblocknum = 16;//空格子个数，初始值为 16
COOR* _emptyblocklist;//空格子坐标链头指针。其中 COOR 为结构体，储存行列坐标。
```

_emptyblocklist 可以被看作为一个查找表，_emptyblocknum 指示表的长度。通过这两个信息可以访问到棋盘上所有的空格子。

对于棋盘的基本操作有：

1）初始化（Initialize）：每次游戏开始前，会清空整个棋盘，并且随机生成两个新数。该过程称为初始化。

2）移动（Move）及其子函数（_left，_right，_up，_down）：尝试对棋盘上的数字进行移动。如果为有效移动，则会更新分数、步数和空格坐标表。

3）添加新数（AddNum）：在有效移动之后按照上述规则添加新数。

4）判断游戏是否结束（GameOver）：如果游戏结束，则返回 true，否则返回 false。

5）输出（Print）：以图形界面的形式输出棋盘、分数和步数。

至此，Board 类已基本实现，可以编写程序实现游戏。

```
Board board;
while (true)
{
  board.Clear();//清空棋盘，将步数、分数复位
  board.Initialize();//初始化，添加两个数
  board.Print();//输出棋盘
  while (!board.GameOver())
  {
      char ch = _getche();//从键盘获得指令
      DIR dir = 0;
      switch (ch)//a,d,w,s 控制移动
      {
      case 'a':dir = LEFT; break;
      case 'd':dir = RIGHT; break;
      case 'w':dir = UP; break;
      case 's':dir = DOWN; break;
      default: break;
      }
      if (board.Move(dir))board.AddNum();//若为有效操作，则添加新数
      else puts("无效操作!");
      board.Print();//输出棋盘
  }
}
```

三、 2048 算法先行研究

3.1 游戏基本策略

在 2048 游戏中，如何保持棋盘不陷入游戏结束状态从而进行更多走步，获得更高的分数是游戏策略的关键。游戏结束状态是棋盘被占满无法移动以及相邻格子互异无法合并的状态，故要远离该状态，可以从两个角度出发：一是棋盘非空格子数尽量少，保持一定的自由空间；二是让棋盘中的数字形成某种排列方式，使数字排列趋向有序，便于之后的合并。可基于这两点设计 AI。

3.2 评估 AI 性能指标

对于一个游戏的 AI“玩家”，评估该 AI 指标有三个：

1）游戏水平，可以用分数、步数、最大格子三个指标来刻画。分数和最大格子之间存在相对固定的关系，达到某个格子所需要合并的数字之和即为此时的分数。比如，若要获得 2048（2^{11}）需要合并两个 1024（分数增加 $2*2^{10}$），获得 2 个 1024 又需要合并 4 个 512（分数增加 $2*2*2^{9}$）……如此类推，如果添加新数时全部出现的是 2，则全部的 4 都需要从 2 的合并开始，总分为 $2*2^{10}+4*2^{9}+2^{3}*2^{8}+\cdots+2^{10}*2=10*2^{11}=20480$ 分，而对于全部出现 4 的情况，为 $9*2^{11}=18432$ 分。实际游戏中，2 和 4 按照 9：1 的概率出现，所以平均来说，出现 2048 格子时的最低分数（由于还可能有别的格子在合成 2048 时未被用上）应当为 20480*0.9+18432*0.1=20275.2。以此类推，合成其他格子时的分数如下表所示。

最大格子	全部出现 2	全部出现 4	9：1
4	4	0	3.6
8	16	8	15.2
16	48	32	46.4
32	128	96	124.8
64	320	256	313.6
128	768	640	755.2
256	1792	1536	1766.4
512	4096	3584	4044.8
1024	9216	8192	9113.6
2048	20480	18432	20275.2
4096	45056	40960	44646.4
8192	98304	90112	97484.8
16384	212992	196608	211353.6
32768	458752	425984	455475.2
65536	983040	917504	976486.4
131072	2097152	1966080	2084044.8

2）空间复杂度，即 AI 运行需要占用多少存储空间；

3）时间复杂度，体现在 AI 做出决策的速度。

3.3 利用 S 形链策略设计的 0 深度搜索算法

基于以上讨论，游戏中需要保持数字的有序性以更好地实现数字的合并，进而与新添加数保持平衡，维持空格子个数。又因为数字的连续合并需要形成一条链，这条链上从大到小每个数都是下一个数的两倍。基于此可以设计一种算法使棋盘上数字从大到小排成一条 S 形链（如图），只需不断合成链尾端的数字即可递进地合成链头部的最大数。这种排列方式保持了棋盘上数字的高稳定性和有序性，也成为达到可能的最大数字 131072 的唯一方式。

基于此策略，设计一种算法，该算法不对棋盘进行搜索，而直接分析当前棋盘布局，以合成-断链修复（Merge or Broken-chain Fixing）原则做出决策，即首先获得数字按 S 形形成链的情况，如果在某处出现了“断链”，即数字的排列不符合沿链的单调性规律时，则进行修复，通过数字的移动、合并等手段重新形成一条更长的 S 形链。当没有出现断链情况时，则进行链尾端的数字合并。

经过不断优化，该算法的胜率可以达到 50%（见下表），并且速度可达 50000 步/秒。

（版本：V3.4.2.4，编译时间：2019 年 6 月 16 日）

最大数	局数	比例
16	3	0.00%
32	57	0.01%
64	761	0.08%
128	5618	0.58%
256	33643	3.45%
512	131446	13.49%
1024	311121	31.93%
2048	362606	37.22%
4096	125963	12.93%
8192	3131	0.32%
总局数	974349	
平均分	24505	
平均步数	1262.5	
胜率	50.46%	

该算法在速度上具有优势，但由于仅对当前局面进行分析且分析很不全面，所以胜率较低。接下来将对搜索类算法进行研究。

四、 运用蒙特卡罗模拟和动态规划寻优实现 AI

4.1 概述

可以设计评估函数，以上文所述的空格数和有序性评价某个局面的好坏。因此，可以建造一棵搜索树，通过一定深度的走步模拟，基于评估函数评估最终局面寻优，进而找出当前局面的最佳移动方向。

但是，玩家做出移动之后，虽然过渡态仅有 1 个，但依据空格子数，终态的数目会很多，以至于树的深度增加时，建造树和计算节点评估值的时间和空间复杂度都会大幅增加，故为了得到近似最优解，同时保证智能体的运行速度，采用蒙特卡罗随机模拟方法，即每次局面向某方向移动之后，系统随机选择格子产生一个新数到达终态，而不是产生所有终态的结点。通过多次模拟也可得到节点的近似平均评估值。

4.2 搜索树的建立

根据上述思想，构造数据结构——树，这棵搜索树的度最大值为 4，每个节点可以通过 1 个根指针和四个孩子指针与根节点和四个方向移动后的终态节点连接。其中，如果不是有效移动，则对该节点进行先剪枝。

建立节点类 TreeNode，每个节点含有一个棋盘状态，因此，它继承 Board 类；除了根指针和孩子指针外，树节点还应包含节点深度_depth 和节点评估值_eval_score。其中，节点评估值在节点建立时即通过评估函数_eval()被计算。

```
class TreeNode :public Board
{
    public:
        TreeNode(TreeNode& origin,int8 dir); //节点的构造函数，包含根节点的引用和移动方向。
        ~TreeNode() {};
    private:
        int8 _depth = 0;//节点所在深度
        TreeNode* _root = nullptr;//根节点指针
        TreeNode* _child[4] = {nullptr};//孩子节点指针数组
        float _eval_score = 0.0F;//评估值

        void _eval();//评估函数
};
```

节点构造函数如下所示。建立新节点时，首先将新节点的深度增加 1，将指针连接到根节点，然后根据移动方向 dir 进行移动。若为有效操作，则添加新数并进行评估；反之，则说明该节点不能再向下扩展，应被 Tree 剪枝，故修改 depth 为-1 以向树构造函数传递销毁该节点的信息。

```
TreeNode::TreeNode(TreeNode & origin, int8 dir) : Board(origin)
{
    this->_depth = origin._depth + 1;
    _root = &origin;
    if (Move(dir)) { AddNum(); _eval(); }
    else _depth = -1;
}
```

基于节点的树类 Tree 负责树的建立、遍历寻优与销毁。建立树 Tree 时，需要提供根节点和最大深度。

```
class Tree
{
    public:
        Tree(int8 depth, TreeNode* root);//构造函数，调用 CreateTree 函数
        ~Tree();//析构函数，调用 DestroyTree 函数
        void CreateTree(TreeNode * root);
        void DestroyTree(TreeNode * root);
    private:
        int8 _max_depth;//最大深度
        TreeNode * _treeroot;//树的根节点
        float _findmax(TreeNode * root);//根据根节点遍历寻优
};
```

具体的 CreateTree 函数与 DestroyTree 函数如下所示。根据“资源获取即初始化（RAII）”思想，在构造函数中申请内存空间，在析构函数中释放。

在树的建造中，从根节点建立四个方向移动后终态的孩子节点，如果孩子节点返回无效操作，即刻删除该节点，进行剪枝。

```
void Tree::CreateTree(TreeNode* root)
{
    if (root == nullptr || root->_depth == this->_max_depth)return;

    for (loop_control i = 0; i < 4; i++)
    {
        root->_child[i] = new TreeNode(*root, i + 1);
        if (root->_child[i]->_depth == -1) { delete root->_child[i]; root->_child[i] = nullptr; }
        else CreateTree(root->_child[i]);
    }
}
void Tree::DestroyTree(TreeNode* root)
{/ /注意：初始节点 root 不会被销毁。
    if (root == nullptr)return;

    for (loop_control i = 0; i < 4; i++)
    {
        DestroyTree(root->_child[i]);
        delete root->_child[i];
    }
}
```

4.3 状态评估函数

基于上述分析，对于棋盘的状态评估应从两个方面入手：一是棋盘上空格的多少，二是棋盘上数字的有序性。前者的量化较为简单，可以方便地由棋盘目前空格个数除以最大空格个数（16）得到。

对于有序性的量化，经实际试验，按照 S 型链策略编写出的状态评估函数很难反映棋盘的实际有序性情况，故目前采用一种简单策略：单调性策略对棋盘进行评估。

单调性策略是指棋盘上所有数字满足对角线方向的单调性，即从角到内部，数字依次减小（如图），这种排列方式同样能保证棋盘的有序性。

对单调性策略进行简单分析，即可大致得出单调性的量化方法：

首先，我们知道，当一个数越来越大的时候，就会变得更难被合成。比如如果由 4 合成 8，只需要另外一个 4 或者两个 2；但是如果由 64 将 128 合成，那就需要先合成另外一个 64，而合成另外一个 64 需要再合成两个 32，合成 32 又需要合成两个 16……如此可见一斑。也就是说，大数的合并频率要比小数指数级减少。这意味着大数在整个棋盘中应占据一个较为稳定的位置，并且比它小的其他数应处于次稳定的位置，才能达到最佳效果。因此，整个棋盘最稳定的位置在角上，最大的数应当处于某一个角，其他数从大到小按辐射状分布。

同时，为了合并一个数，比它小而尽量大的数必须要在其旁边，该数才更有希望被合并。因此，在处理有序性问题时，两个数字的相对大小同样重要。并且，较大数的有序性相比于较小数的有序性更重要，因为较大数的位置更难改变。

程序的设计：

空格多少和单调性均以一个 0~1 范围的浮点数值给出。

对于单调性，程序会逐行、逐列扫描棋盘，如果两个相邻非零数相等，或从大到小的方向符合指定方向，则为 1，如果小于，则为-1。对每个值进行加权，权重为两个数对数值之和。最后求出 4 行/列的加权平均值。所有行的加权平均值和所有列的加权平均值均分别取绝对值后按权重 1：1 相加。

最后，将空格子比例与单调性结果在按照 1：1 权重加权平均，得到一个 0~1 范围的浮点数。这个数即为该局面的评估值。数值越大，则棋盘布局越好。

4.4 程序运行效果

由以上内容可知，AI 执行的是全局搜索策略。全局搜索策略即扩展所有可扩展的节点，即扩展所有还可进行有效操作的节点，直至游戏结束，或达到设定最大搜索深度。以上代码即为进行全局搜索策略的部分程序实现。为了对整个程序进行总控，还需要建立智能主体类 Agent，操控游戏，进行树的建造、节点寻优等操作，Agent 的具体实现在此略去。

经试验，在搜索深度为 4，模拟次数为 64 时，本算法的胜率可达到约 94%（见下表），速度约为 100 步/秒。

（版本：V4.0.0.6，编译时间：2019 年 7 月 26 日）

最大数	局数	比例
512	5	0.40%
1024	72	5.71%
2048	516	40.89%
4096	640	50.71%
8192	29	2.30%
	1262	**93.90%**
	平均步数	2514.6
	平均分	52734

五、 运用时序差分增强学习实现 AI

5.1 概述

由于 2048 的游戏过程很容易被建模为马尔可夫决策过程，又因为决策过程所需信息完备，所以使用时序差分增强学习不失为解决问题的绝佳途径。下面将从马尔可夫决策过程在 2048 中的特化开始，通过时序差分算法对游戏开展研究。

5.2 马尔可夫决策过程

一般来讲，马尔可夫决策过程可以用一个五元组(S,A,P,R, γ)来表示。下面结合 2048 游戏对这 5 个信息进行特化。

定义集合 S 为游戏中所有可能的棋盘状态的集合，$s_t \in S$ 表示智能主体在步数 t 的状态。

对于 4×4 棋盘而言，可能出现的最大数字为 131072（2^{17}），即每个格子有 19 种状态，整个棋盘总的状态数可达 $19^{16}(\approx 2.88\times 10^{20})$，故储存所有棋盘状态是不现实的，需要想别的办法。

定义集合 A 是决策过程中的动作集合，即左移、右移、上移和下移四个移动操作。对于某个特定状态 s 而言，存在对该状态的有效操作集合 $a(s)\in A$。

定义函数 $P_1=P_1(s,a,s')\equiv 1$ 及 $P_2=P_2(s',s'')$为状态之间的转移概率。P_1 为始态通过移动操作至过渡态的转移概率，由于始态和移动操作确定后确定唯一的过渡态，因此 P_1 始终为 1。P_2 是过渡态通过添加新数转移至终态的转移概率。设 s'有 i 个空格子，则有

$$P_2(s',s'')=\begin{cases}\frac{0.9}{i}, & \text{当第 } n(1\le n\le i)\text{个格子处出现 2}\\ \frac{0.1}{i}, & \text{当第 } n(1\le n\le i)\text{个格子处出现 4}\end{cases}$$

定义值 R=R(s,a,s',s'')是从始态采取从某一动作到达终态后的奖励值。鉴于得到更高的分数是游戏的最终目的且分数的变化值容易得到，奖励值被定义为

$$R=\begin{cases}\text{score}(s'')-\text{score}(s), & \text{当! GameOver}(s'')\\ -V(s'') & \text{当 GameOver}(s'')\end{cases}$$

注意，分数的增加只发生在始态至过渡态的转变过程中，故有第一个等式；如果游戏结束，则该步以后的累积回报将为 0，故有第二个等式。

定义值 γ 为折扣因子，用于减弱后面状态的回报对当前状态衡量的影响。由于 2048 游戏要求智能体更有“远见”，又因为在 TD（0）中只进行单步更新，故设定 γ 为 1.

5.3 数据关系的元组表示与值函数定义

定义状态值函数 $V_\pi(s)$表示从状态 s 出发，使用策略 π 所带来的期望累积奖赏。

按照贪婪策略的定义式，定义策略 $\pi(s)$：

$$\pi(s)=argmax_a[R(s,a,s',s'')+P_1(s,a,s')P_2(s',s'')V_\pi(s'')]$$

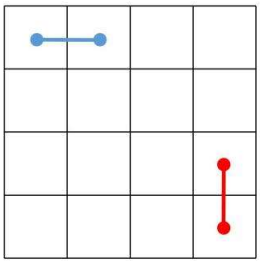
那么接下来问题来了，值函数 V(s)该如何得到和更新呢？

如前文所述，存储棋盘上所有的状态并更新其值函数是不现实的，所以必须寻求其他方法以减少数据总量至可行水平，这意味着智能主体可以完整地观测到棋盘的数字排列方式，但无法完整地以值的形式存储这些排列方式，也就是说，智能体理论上虽然能“知道”棋盘的局面，但在实际上只能学习并利用该局面的一部分。

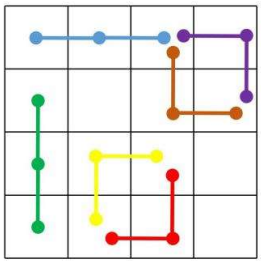
值函数可以看作是对当前棋盘状态的评估，而分立的格子不具有评估价值，因为分立的格子不能表示棋盘局面的好坏，只有多个格子及格子上的数字组成数据关系，才能作为排列方式的一部分被评估。并且，数据关系越多，这些关系就越能拟合整个棋盘的排列方式，智能体也就更加“博学”了。

我们将每一个数据关系表示为一个数值变量，则棋盘的状态值函数值可以简单地近似为这些变量的加和。

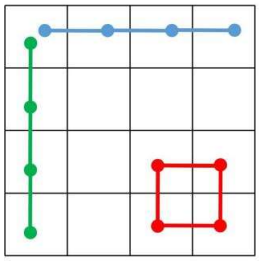
综上所述，我们要寻找棋盘上的数据关系，而数据关系可以用集合、元组来表示。如果仅考虑横纵向相邻格子间的关系，则对于 2 元（维）组、3 元组和 4 元组，分别有 24 组、52 组、89 组元组，如图所示。



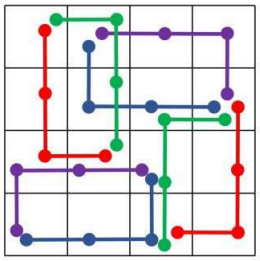
2元组：共24组
直线形：2种空间构型，每种构型12组；



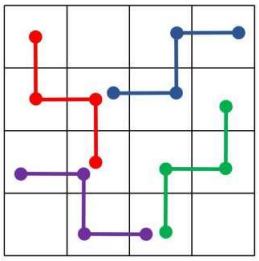
3元组：共52组
直线形：2种空间构型，每种构型8组；
L形：4种空间构型，每种构型9组；



4元组：共89组
直线形：2种空间构型，每种构型4组；
正方形：1种空间构型，每种构型9组；



L形：8种空间构型，每种构型6组



S形：4种空间构型，每种构型6组

按照每个格子 16 种状态（因实际游戏中极难出现 32768 及以上的数字），每个元组的值以 4 字节浮点数储存，则所占用空间如表所示。

元组维度	元组组数	元组个数	占用空间（MB）
2	24	6144	0.02344
3	52	212992	0.8125
4	89	4259840	22.25
5	>89	>93323264	>356
（对比）16	1	2^{64}	2^{46}

可以看到，随着元组维度增长，元组的个数呈指数级爆炸性增长，在 5 元组时已经至少有亿级数据需要存储，所需空间也已达到数百 MB。虽然当前的小型计算机还能满足这样的存储需求，但对于 AI 而言，以空间换智力这种做法也应有所限制。所以本文只研究 4 元组及以下的存储量。

这些数据显然应以查找表的方式存储。某元组的值在整个查找表的位置

$$L(n, v_1 \dots v_i) = n \cdot 16^i + \sum_1^i v_i \cdot 16^{i-1}$$

其中 n 为元组的编号，vi 表示该元组第 i 个元素的值。

5.4 时序差分学习

时序差分学习的主要思想是进行值函数近似，不断减小时间差分，即当前状态值与新的基于下一步的状态值的新值之差。在 2048 游戏中，基于上述马尔可夫决策过程，可以定义时间差分：

$$\Delta = R(s, a, s', s'') + V(s'') - V(s)$$

则值函数的更新公式为

$$V(s) \leftarrow V(s) + \alpha \cdot \Delta$$

其中 α 为更新率，设定为$\alpha = 0.1$.

算法如右框所示。

1. 初始化所有$V(s)$，给定参数α
2. Repeat：
   初始化棋盘，并根据贪婪策略选择 a
   Repeat（对于每一步）：
   1) 根据贪婪策略在初态 s 选择动作 a 得到回报 r 和终态 s”
      根据贪婪策略得到动作 a’
   2) $V(s) \leftarrow V(s) + \alpha \cdot \Delta$
   3) s=s”,a=a’
   Until GameOver(s)
   Until 达到学习次数
3. 输出最终值函数

5.5 智能体类 Agent 的实现及程序运行效果

为了实现时序差分学习，需要建立智能体 Agent 类，以实现对环境(Environment)及其反馈(Reward)的感知、对状态(State)的分析、执行操作(Action)以及更新值函数进行学习等操作。

Agent 类的实现如下：

```
class Agent
{
    public:
        Agent(float update_rate);//构造函数
        ~Agent(){delete _lookUpTable;}//析构函数，释放查找表
        void Learn(int learn_time);//学习函数

    private:
        DIR _chooseNextMove(Board& present_state);//选择当前状态最佳动作
        void _updateWeight(Board& last_state, Board& present_state);//更新状态值函数
        float _getStateValue(Board& board);//获取某局面状态值函数
        long _getLocation(Board& board, int num);//获取某局面状态值在内存中的位置
        void _writefile();//将查找表写入文件

        float _update_rate = 0.0f;//更新率
        float* _lookUpTable = nullptr;//查找表指针
};
```

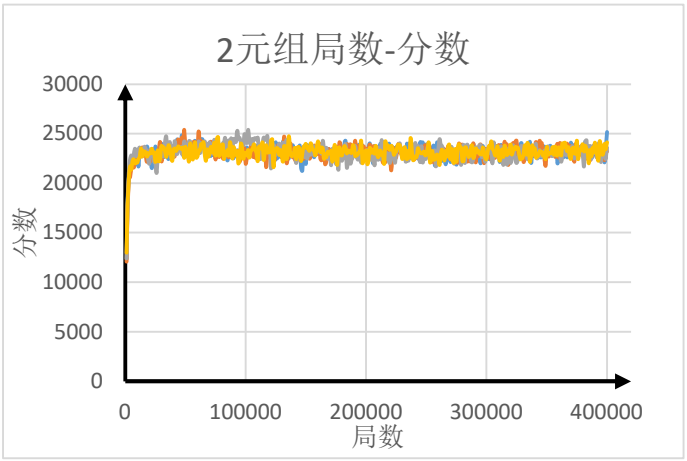
智能体在进行学习（值函数更新）时，需要给定更新率α与学习局数 learn_time。开始学习时，进行值函数值的初始化（全部设为 0），并且每隔一段时间，将值函数表写入磁盘保存。为了检验学习效果，每局游戏结束，将该局游戏棋盘最大数、步数与分数输出保存至磁盘。

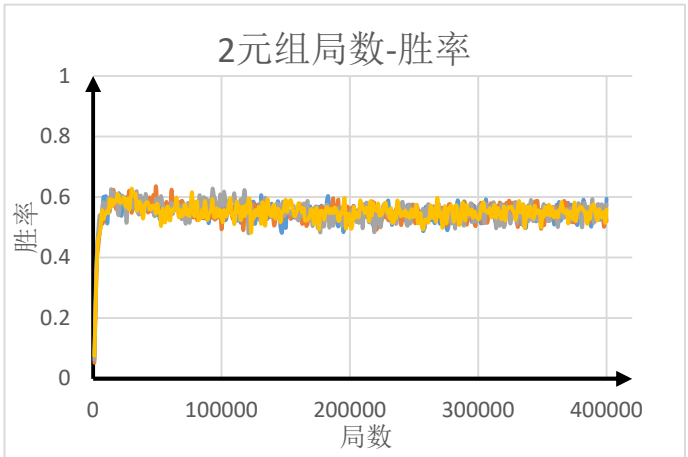
学习函数的实现如下所示。

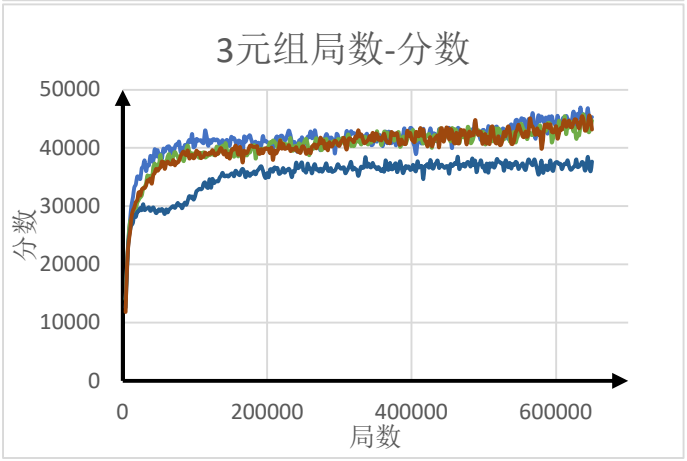
```
void Agent::Learn(int learn_time)
{
    Board gameboard, laststate;//建立两个棋盘，其中一个保存上一步棋盘状态
    //进行循环直至学习局数达到设定值
    for (int i = 0; i < learn_time; i++)
    {
        gameboard.Clear();//清空棋盘
        gameboard.Initialize();//初始化棋盘

        while (!gameboard.GameOver())
        {
            laststate = gameboard;//保存上一步棋盘状态
            gameboard.Move(_chooseNextMove(gameboard));//按照贪心策略选择移动方向
            gameboard.AddNum();//添加新数
            _updateWeight(laststate, gameboard);//计算时序差分并进行值函数更新
        }
        gameboard.SaveToFile();//输出棋盘数据

        if (!(i % 9999))_writefile();//将查找表保存至文件
    }
}
```

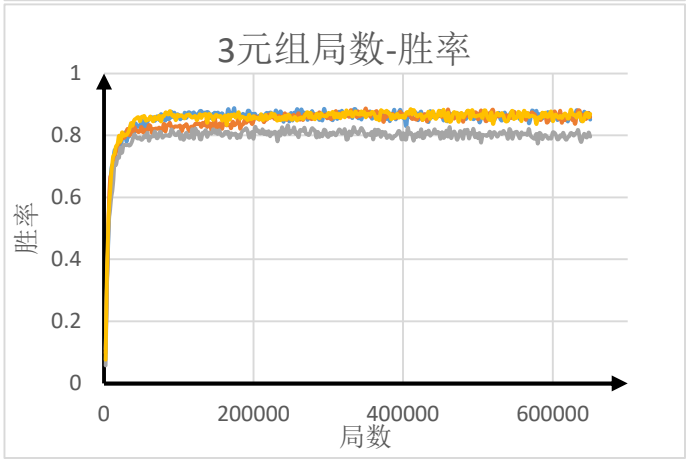
对 2 元组、3 元组、4 元组分别进行测试，对每元组进行 4 次学习，次数介于 40 万局~160 万局之间，学习结果如下图与下表所示：

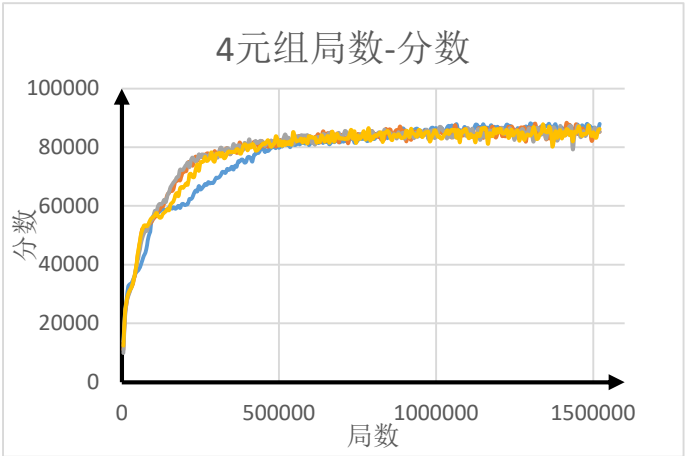
（版本：V5.0.1.10，编译时间：2019 年 10 月 28 日）

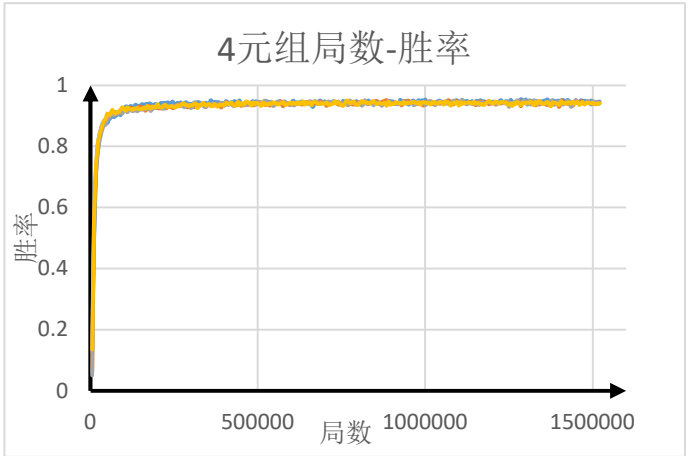






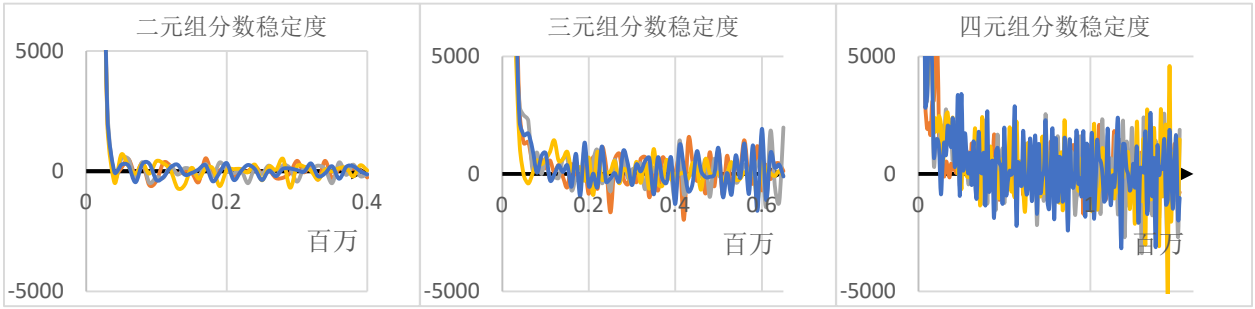






由上图可知，胜率较分数的收敛速度快，这是因为智能体在达到 2048 后，仍可以获得更高的分数。

为了将学习速度可视化，以每 1 万局平均分数的变化作图：



可见，随着值函数数据量的增大，值函数收敛的速度逐渐减慢，不稳定度逐渐增大。2 元组在 20 万局之后基本稳定，而三元组则在 40 万局之后基本稳定，四元组则在 100 万局后基本稳定

为了获取值函数收敛后 AI 的运行效果，取平均分数稳定后 20 万局的平均结果(置信度为 95%)：

元组元	实验次数	学习局数	胜率	分数
2	4	400000	0.5456±0.0032	23129± 100
3	4	650000	0.8477±0.0488	41577±4874
4	4	1550000	0.9436±0.0042	85471±1409

最大数/ 元组组数	2	3	4
128	100.0%	100.0%	100.0%
256	99.8%	99.8%	99.9%
512	98.6%	98.9%	99.5%
1024	90.3%	96.0%	97.8%
2048	54.6%	84.8%	94.4%
4096	4.4%	42.7%	76.5%
8192	0.0%	0.4%	41.2%

可见，在 4 元组时，智能体的胜率不但已经达到了 94.4%左右，并且还有 41%左右机率达到 8192，已经达到了高级玩家的水平。

5.6 资格迹

笔者也尝试过将资格迹（TD(λ)）应用于值函数更新中，具体来说是构造循环队列以存储一定长度的棋盘状态序列，然后基于奖励 R 对和迹衰减系数 λ 更新各状态值函数，但实现后，其运行结果不如 TD(0)。具体原因还未深入研究。对于资格迹及其他复杂的基于值函数的学习方法将在今后研究。

六、 结语

本文从 2048 游戏的建模及 C++语言实现出发，研究游戏 AI 设计思想，并分别通过三种不同方法实现了游戏 AI。其中第一种方法（先行研究内容）使用纯粹模拟人类思考方式的策略，不进行任何搜索，运行速度较快，但程序繁杂冗长，且运行结果不理想；第二种方法则基于树数据结构与评估函数，运用蒙特卡罗方法进行动态规划以寻找近似最优解，取得了较好的胜率，但由于需要大量时间进行搜索，运行速度较慢；第三种方法则运用了机器学习中常见的时序差分方法，不使用任何人为评估策略，仅通过值函数更新迭代实现自主学习。通过该方法，不但获得了 94%的胜率，而且程序运行速度达 10 万步/秒，要比前述方法快得多。因此，本研究基本达到了预期目标。

本研究存在数个不尽人意之处：一是在第四章运用蒙特卡罗和动态规划寻优时，对于状态评估函数的编制过于草率，影响了运行效果；二是在第五章运用时序差分算法时，由于计算机算力等原因，没有对算法中各个参数的取值开展大量试验，比如更新率 α 的选择。三是对于时序差分、资格迹及其他更高效的算法缺乏深入研究，仅仅从表面层面上将公式应用于游戏中，并没有太多创新点。总的来说，研究取得了一定成果，但也向笔者提出了很多新的、值得研究的问题。下一步工作将以继续提高 AI 胜率与运行速度为目标，对算法继续进行改进。