智能体大赛报告

2017011307 张晨

2018年5月29日

目录

1 概述

本次大赛共分为贺岁赛与挑战赛两部分比赛，贺岁赛为为期5周(3月4日至4月8日)的线上对战，挑战赛首先进行了为期4周(4月9日至5月6日)的线上预赛，之后于5月13日进行现场决赛。我取得了贺岁赛第5名，挑战赛第13名，总第8名的成绩。

2 贺岁赛 Troy Uncalm

2.1 游戏规则

Troy Uncalm 为玩家提供了一个回合制战场，选手需通过布设战斗单位，占领魔法阵采集能量，与敌方战斗单位交战来摧毁敌方大本营，获得游戏胜利。

游戏提供建造、移动、攻击、潜行四种基本操作，每回合内，玩家可在自己基地中建造单位、移动已建造单位、占领能源、攻击敌方单位，最终率先攻破敌方基地的一方为游戏胜者。

- 战斗单位分类

  共4个兵种：火族、冰族、风族、弓箭手

  克制关系：火克冰，冰克风，风克火，火、冰、风均克制弓箭手，受克制则受到伤害*2 相较其他兵种，弓箭手射程大，攻击力大，但每回合移动距离少，血量少，且由于克制原因受到单次攻击时的伤害大

- 单位特殊机制
  - 真假单位

    除生成普通单位外，还可生成每个兵种的幻象单位。幻象单位造价低，每回合移动距离高，但不可进行攻击，且受到攻击后会被直接消灭。幻象单位在距离敌方一格范围内时会被侦测到。

  - 潜行设置

单位可以进入潜行状态，使其对敌方不可见。每个潜行单位每回合都需消耗一些能量。潜行单位在距敌方一格范围内会被侦测到

- 行动力机制

  每个单位每回合有一定行动点数（幻象4，风火冰3，弓箭手2），每移动一格需消耗一点行动点数，攻击消耗所有剩余点数

- 攻击力与HP挂钩

  战斗单位实际攻击力为攻击力最大值乘以当前HP比率，即HP越高，攻击力越大

- 反击机制

  单位攻击敌方单位时会自动触发反击，反击伤害与攻击伤害结算方式相同

2.2 游戏策略

2.2.1 攻击策略

我采取了快攻打法，在前两轮制造出共计6个风火冰族单位，让它们以最快的速度攻占敌方大本营。

- 可行性分析

  1. 游戏对于“摧毁敌方大本营”的规则为：玩家自己的回合结束时，每有一个单位在敌方基地中，便可对敌方基地造成一点伤害。累计五点伤害则获胜。故本方单位只要进入敌方基地，便可对敌方基地造成伤害，防守方若想保护好自己的基地，必须要确保消灭在自方基地周边很大范围内的单位，以保证没有单位能进入基地。但这是十分困难的，主要原因如下：

     （1）由于快攻打法很快，防守方难以在自家基地附近布置足量的用以防守的兵。

     （2）即便全部使用满血单位去攻击我的快攻兵，且属性恰好克制，或使用弓箭手攻击快攻兵，也需要2次攻击才能击毙，若没有满足上述条件，所需攻击数还要增加。

     （3）风火冰族单位的移动速度较快，进入对守方基地有威胁的区域到进入守方基地所需回合数很少

  2. 阵地战需要在开局攻占能源，以保证未来的经济，故用以防守的兵会比较少。排名靠前的玩家除我之外无一例外地选择了阵地战，故初期的防守会比较弱。
  3. 由于攻击策略丰富多样，难以兼容地设计针对每一种攻击策略的防御方式。而排名靠前的快攻玩家极少，特意设计策略去防御的性价比很低。

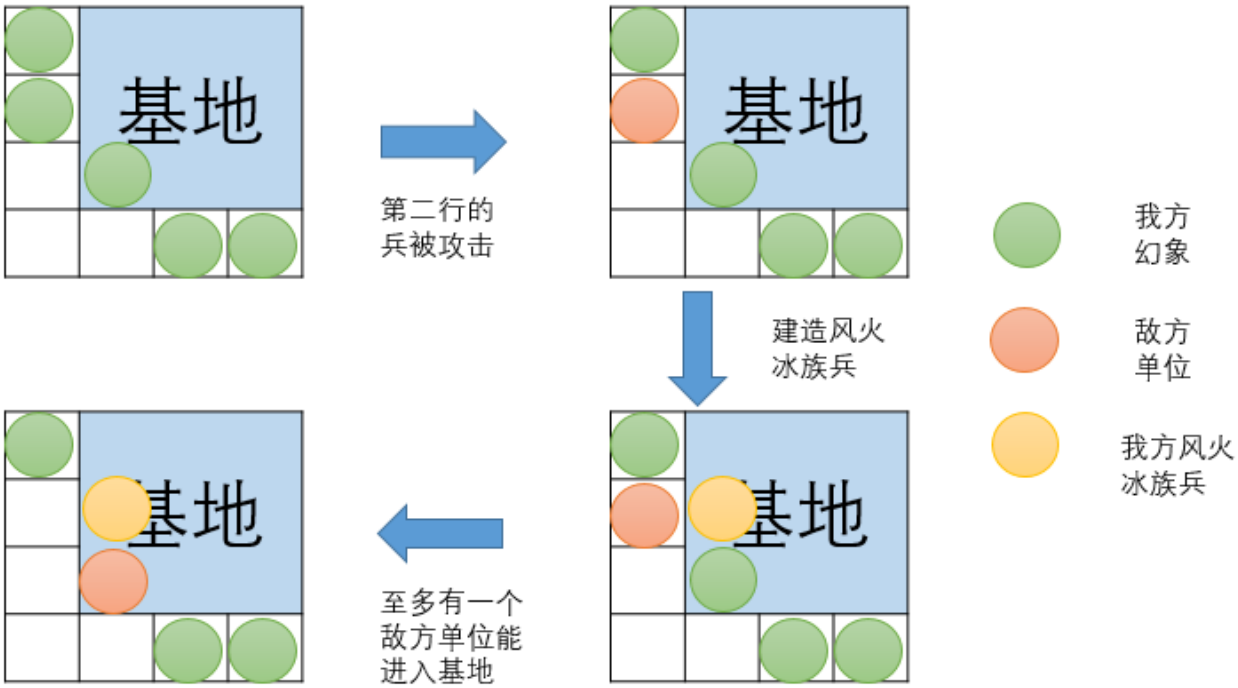
- 具体实现方法

  分别对每个快攻兵进行决策

  1. 如果此回合可以进入基地（无论是直接进入还是通过攻击敌方单位），那么就进入基地
  2. 否则，采取贪心策略，每一步都尽可能地向基地移动（如果能一次攻击击毙前进的最优路径上的单位，那就击毙它开路），在此基础上尽量向敌方单位分布较少的位置移动
  3. 如果被一排兵挡在了基地之外，那么就着重攻击其中的一个兵开路

2.2.2 防御策略

我采用的是快攻策略，没有打持久战的能力，故防御以拖延对方，为自己快攻争取时间为主。我的防御策略是尽快使用幻象在基地外围一圈，在幻象被消灭后尽量使用普通兵补上空位



- 可行性分析
  1. 在2.2.1节中已经分析了防快攻的困难性，在我开局派出去两个攻击兵，且不探矿的情况下，消灭敌方单位是不可能的，拖延时间是唯一方法。
  2. 该拖延时间的方式，能为快攻兵争取到接近两轮的时间，实践中，这套防御体系绝大多数情况下可支撑至游戏胜利或快攻兵被敌方消灭
- 实现细节
  1. 由于中间基地（大小为$3*5$）比两边基地（大小为$3*3$)大，因此围起来所需的兵比较多，需要将部分两边基地产生的单位移动到中间基地
  2. 先产生距敌方基地较近的位置处的攻击兵，再产生距敌方基地较远的攻击兵
  3. 尽管进行了上述2个优化，若对方使用能高速移动的幻象发起快攻，我仍无法完全防御。但可以利用幻象单位移动速度快于普通单位的性质识别幻象，并要求负责快攻的单位消灭第一波幻象。
  4. 尽管从两边基地调到中间基地的是幻象，移动过程中仍按照普通兵3格/回合的速度移动，以隐藏自己是幻象的事实。

2.3 代码设计

2.3.1 整体设计

使用$SoliderSaver$类将单位分为$attacker,defender,supporter,rival$四类，其中$rival$为敌方单位，剩余三种类型为我方单位。

之后，使用命名空间$AttackArranger,DefendArranger,SupportArranger$分别对三种我方单位进行处理

2.3.2 SoldierSaver类

```
class SoldierSaver
{
public:
```

```
    SoldierSaver();
    enum Type
    {
        Attacker,Defender,Rival,Supporter,Size
    };
    unsigned int size(Type a) const{ return soldier[a].size(); }
    Point s(Type a,int b) const{ return soldier[a][b]; }
private:
    int a[ms][ms];
    vector<Point> soldier[Size];
};
```

该类实现了将单位分为attacker,defender,rival,supporter四类的功能

- attacker：我方快攻单位
- defender：我方防御单位
- supporter：从两边基地移动至中间基地的防御单位
- rival：敌方单位

  枚举"Type"中还有一个元素为"Size"，它的枚举值恰为单位的类型总数，这使得若想要增加一种单位类型，只需要在Size之前添加这个单位类型的名称。

2.3.3 AttackArranger命名空间

该命名空间内的变量定义及函数声明如下

```
namespace AttackArranger
{
    int dist[ms][ms];
    int danger[ms][ms];
    int lx[5]={-1,1,0,0,0};
    int ly[5]={0,0,-1,1,0};

    int attackHP(const sd &f,const sd &g);
    bool winInOneAttack(int fx,int fy,int gx,int gy);
    bool better(Point x,Point y);
    bool baseBetter(Point a, Point b);
    bool gotoBase(const Point &p,int* d);
    bool meetFake(const Point &p, int *d);

    void valueInitialize();
    void go(const Point &p,int *d);
    void moveForward(const Point &p);
    void moveForward();
    void buildSoldier();
    void solveAttack();
}
```

- $dist[ms][ms]$存储了从每个位置进入敌方基地所需走的最短距离
- $danger[ms][ms]$存储了能攻击到每个位置上的单位的敌方单位的数量

- $lx[5]$ $ly[5]$ 两个数组用以实现向上下左右移动及原地不动
- $attackHP$计算了士兵f攻击士兵g时，会对g造成的伤害值
- $winInOneAttack$判断了位于(fx,fy)的单位能否通过一次攻击击毙位于(gx,gy)的单位
- $better$判断了快攻单位移动到x，y哪个位置更好
- $baseBetter$ 判断在进入基地时应该优先选的点，如果a比b优，返回1，若b比a优，返回0。具体判断方式为，在敌方基地中的点比不在敌方基地中的点优，同在敌方基地中的点，走的深入的更优
- $gotoBase$判断了位于p的单位能否进入敌方基地，如果能够进入，通过d数组返回最优线路
- $meetFake$处理了需要快攻单位攻击敌方第一波幻象快攻兵的情况。它判断了位于p的单位是否需要攻击快攻兵，如果需要，通过d数组返回路线
- go通过与逻辑交互，向逻辑发送沿d数组运行的指令。d数组存储了第i步的方向（代表第i步的移动方向为$(|lx[d[i]]|,|ly[d[i]]|)$,从0开始编号,d>0表示移动，d<0表示攻击，策略中不存在后退一步攻击）
- $moveForward(constPoint\&p)$为产生位于p的单位此回合操作的函数
- $moveForward()$分别调用了每个单位的$moveForward(constPoint\&p)$
- $buildSoldier()$完成了快攻单位的建造
- $solveAttack()$为主函数需要调用的函数，其调用了$moveForward()$与$buildSoldier()$两个函数

2.3.4 DefendArranger命名空间

该命名空间下的变量声明及函数定义如下

```
namespace DefendArranger
{
    static const int top=5,below=5,middle=9;
    const int topy[top]={22,24,23,22,22};
    const int topx[top]={2,2,2,1,0};
    const int belowy[below]={2,0,1,2,2};
    const int belowx[top]={2,2,2,1,0};
    const int midy[middle]={12,14,10,13,11,14,10,14,10};
    const int midx[middle]={2,2,2,2,2,1,1,0,0};

    int moveX(int x,int y);
    bool exist(int x,int y);
    int getBestType(int x);

    void buildDefender();
    void adjust();
    void solveDefend();
}
```

- $topx$ $topy$ $belowx$ $belowy$ $midx$ $midy$六个数组存储了每个基地中防御兵的建造顺序
- $moveX$ 判断了位于(x,y)的单位是否需要外移以扩大防御范围，返回该单位应该位于的位置的x坐标
- $exist$判断了于位置(x,y)建造的单位还是否存活
- $getBestType$返回克制类型x的类型，在补兵时调用以补最优类型的兵
- $buildDefender$完成了防御单位的建造
- $adjust$完成了防御单位的外扩移动
- $solveDefend$为主函数所需要调用的函数，其调用了$buildDefender()$与$adjust()$两个函数

2.3.5 SupportArranger命名空间

该命名空间下只有一个函数，用以控制supporter 的移动

```
namespace SupportArranger
{
    void solveSupport();
}
```

2.3.6 mapSaver命名空间

此命名空间存储了一些局面信息，用以判断己方是否曾经受到过敌方攻击（用以区别防御兵消失是由于从两边基地向中间基地调度还是由于敌方攻击）

```
namespace mapSaver
{
bool last[ms][ms];
bool now[ms][ms];
bool beenAttacked = 0;
void save(bool s[ms][ms])
bool attacked()
}
```

- $last[ms][ms]$ $now[ms][ms]$ 分别存储了完成上一回合操作后的局面与此回合的输入局面
- $beenAttacked$存储了在之前的回合中己方是否被攻击过
- $save$将当前局面信息存储到$s[ms][ms]$数组中，存储操作后局面与存储输入局面分别调用$mapSaver::save(mapSaver::last)$;与$mapSaver::save(mapSaver::now)$;
- $attacked$通过对比$last$ now的信息更新$beenAttacked$，并返回己方是否曾经被攻击过

2.3.6 用以与逻辑交互的代码

```
Point convert(int &x, int &y)
{
    if (Logic::Instance()->GetId() == 1)
    {
        x = 24 - x;
        y = 24 - y;
    }
    return Point(x, y);
}

bool build(int x, int y, int type, bool real)
{
    convert(x, y);
    return Logic::Instance()->Build(x, y, type, real);
}

bool attack(int fx, int fy, int tx, int ty)
{
    convert(fx, fy);
    convert(tx, ty);
    return Logic::Instance()->Attack(fx, fy, tx, ty);
```

```
}

bool move(int fx, int fy, int tx, int ty)
{
    convert(fx, fy);
    convert(tx, ty);
    return Logic::Instance()->Move(fx, fy, tx, ty);
}

int hp(int id)
{
    return Logic::Instance()->GetHP(id);
}

Soldier Map(int x, int y)
{
    convert(x, y);
    return Logic::Instance()->GetMap(x, y);
}

bool base(int x, int y, int id)
{
    convert(x, y);
    return Logic::Instance()->InBase(x, y, id);
}
```

左下角为(0,0)，右上角为(24,24)。由于从左向右移动与从右向左移动的策略几乎相同，故首先利用convert函数将从左向右与从右向左统一，再运行其他代码

2.3.6 主函数

```
void playerAI()
{
    ID = Logic::Instance()->GetId();
    roundID++;
    mapSaver::save(mapSaver::now);
    ssr = new SoldierSaver;
    AttackArranger::solveAttack();
    SupportArranger::solveSupport();
    DefendArranger::solveDefend();
    mapSaver::save(mapSaver::last);
    delete ssr;
}
```

使用$SoliderSaver$将兵分为四类，再使用三个$Arranger$处理属于己方的三类兵，在该回合开始与结束后分别在$mapSaver$中存储局面信息

3 挑战赛 Stealth

3.1 游戏规则

Stealth中，每个玩家都作为一个刺客隐藏在一个村庄中，玩家需要模仿村民的行为以隐藏自己，并设法找到并干掉其他刺客，每局玩家数为4~6，标准局为6人

- 村民

  地图上会有随机生成的NPC，只会随机游走，它们被称为村民，攻击它们可以补血，杀掉它们可以得到额外的金钱

- 刺客

  刺客由玩家扮演，无法从外观上分辨它们和普通村民，因而玩家需要通过对视野内单位的观察，找出行迹可疑的刺客

- 道具

  玩家可使用两种道具：炸弹和守卫。炸弹是威力巨大的远程投掷道具，可对地图的任何一处造成毁灭性打击（单位的HP为100，炸弹的伤害为89点HP）

- 昼夜

  游戏中有昼夜交替。白天，玩家事业覆盖整个视野，夜晚只能看到周围很小的一个区域

- 灼烧伤害

  白天玩家的HP会持续流失，因而需要主动攻击他人来回复HP

3.2 游戏策略

此游戏中，玩家只要想要进行攻击，必然要暴露自己。因此，我采取了比较激进的打法：在没有发现其他玩家时，模拟村民的运动，在发现其他玩家后，直接去对它进行攻击。

- 判断其他玩家的方法

1. 若一个单位的运动路线不符合村民的运动路线，则它是玩家(对应 $Unit :: SetPlayerReason :: strangeVelocity$)
2. 若一个单位扔出了炸弹，则它是玩家(对应$Unit :: SetPlayerReason :: bomb$)
3. 若一个单位周边出现了尸体，且满足能确定这周围没有炸弹爆炸与该单位是尸体周边唯一一个单位两个条件，则该单位是玩家(对应$Unit :: SetPlayerReason :: corpse$)
4. 若我的hp与上回合的差别达到某个阈值（阈值高于灼烧伤害），且周围只有一个单位，则那个单位是玩家(对应$Unit :: SetPlayerReason :: attackme$)

- 策略的实现细节及分析如下：

1. 在黑夜，尝试击杀距离自己最近的村民，用以补HP并积累金钱

   由于黑夜中玩家视野极小，击杀村民不会导致自己被发现，比较安全

2. 如果发现一个到达自己附近的炸弹，并且自己能够躲过这个炸弹，则尝试躲开，若自己无法躲开，则忽略此炸弹

   炸弹伤害极大，躲开它一定是最优的，但若无法躲开，就继续模仿村民运动，以让自己不被发现，从而不被发现以求延长存活时间

3. 如果发现了某个单位是玩家，那么向它投掷炸弹，并追着它打

   投掷炸弹会导致自己暴露，但这是最高效的击杀方法，因为如果不使用炸弹，则只能追着其他玩家打，奇怪的移动路线会使自己更长时间的暴露

   实际测试发现，炸弹+追着打的胜率要高于只使用炸弹，故采用前一种方法

4. 如果不满足以上几点条件，则模仿村民运动，并随机攻击自己周围的单位补血

5. 只要有钱，就买炸弹

   死亡会导致金钱减少，但不会影响已购买物品。剩余金钱数是排名时较为靠后的关键字，可不考虑

3.3 代码设计

3.3.1 总体设计

首先，需要实现"寻找出玩家"的功能，总体设计如下：

$SitutationSaver$命名空间负责存储每一轮的局面，并利用每一轮的局面更新每个单位是否为玩家的信息。执行更新时，分别调用各个$solver$命名空间中的$update$函数，利用每个$solver$更新玩家信息。

总共定义了四个$Solver$:$UnitSolver$，$BombSolver$,$CorpseSolver$,$AttackedSolver$，分别实现3.2节提到的四种发现玩家的方法。

具体实现时，需要从逻辑的$PUnitInfo$ $PBombInfo$派生$Unit$ $Bomb$两个类，以存储更多的局面信息

之后，需要实现自己的策略。

逻辑中给出了一个$Strategy$基类，我通过继承这个基类实现了多种策略。Strategy基类的一个非常重要的设计是，为该类定义了enable()与disable()两个函数，使用户可以通过禁用掉一些策略来保证不会同时使用相互影响的策略，但需注意在每回合结束后，需要把禁用掉的策略复原

3.3.2 对逻辑中类的继承

3.3.2.1 Unit类

$Unit$类继承自逻辑中的$PUnitInfo$类，实现了对单位更为复杂的处理，主要的成员如下：

```
class Unit: public PUnitInfo
{
public:
    Unit(const PUnitInfo& unit);
    Unit(const PUnitInfo& unit, const Unit& pre);

    enum SetPlayerReason { corpse, bomb, attackme, strangeVelocity, size};

    void setPlayer(SetPlayerReason a);
    void setAttacked();

    bool isPlayer() const;
    bool attacked() const;

    bool operator == (const Unit& unit)
    {
        return this->id == unit.id;
    }
    friend ostream &operator << (ostream &os, const Unit& unit);

protected:
    void checkRoute(const PUnitInfo&);
    void checkBomb(const PUnitInfo&);
    void checkAttack(const PUnitInfo&);
    void updateVStatus(const PUnitInfo& unit, const Unit& pre);

    bool m_isPlayer, m_attacked;
    int m_vStatus;
    vector<Vec2> m_trace;
};
```

- 成员变量
  - $m_isPlayer$存储了单位是否被判定为玩家
  - $m_attacked$存储了单位是否被我攻击过 (在攻击村民补血的过程中需尽量保证每个村民只攻击一次，以减少打死村民带来的被发现的风险，因此需要存储此信息)
  - $m_vStatus$存储了当前的移动状态(-2表示不详，-1表示运动，正数表示已经保持静止的轮数，用以判断该单位是否按照村民的移动规则进行移动)
  - m_trace存储了该单位的历史移动轨迹，可用以判断该单位是否按照村民移动规则进行移动
  - $SetPlayerReason$ 枚举定义了判断一个单位为玩家的四种理由，具体含义见3.2节
- 成员函数
  - 第一个构造函数实现了构造一个视野内新出现的单位

- 第二个构造函数通过综合同一个单位在之前轮和当前轮的信息，构造出一个存储该单位在当前轮信息的Unit
- 以下成员函数提供了改变及访问私有成员的接口

```
void setPlayer(SetPlayerReason a);
void setAttacked();

bool isPlayer() const;
bool attacked() const;
```

- 以下成员函数辅助实现了第二个构造函数

```
void checkRoute(const PUnitInfo&);
void checkBomb(const PUnitInfo&);
void checkAttack(const PUnitInfo&);
void updateVStatus(const PUnitInfo& unit, const Unit& pre);
```

- 运算符重载
  - 重载==号，判断两单位是否为同一个单位
  - 重载<<号，以实现输出单位信息的功能

3.3.2.2 Bomb类

$Bomb$类继承自逻辑中的$PBombInfo$类，实现了在仅知道每回合视野内炸弹位置的情况下，计算每个炸弹的起始位置与结束位置，判断方法为：

若按照炸弹的速度逆向追溯，恰好与某个单位位置重合，则炸弹为该单位发出的，该单位位置为炸弹的起始位置，结束位置可通过开始位置与速度推出。

若无法找到重合的单位，则它的起点可能是回溯路线上任何一个玩家可以到达的位置，但由于炸弹的移动时间总共为3轮，起点的可能位置不多，且每向后一轮，可能位置都会减少，若可能位置减少至1，则该可能位置便为炸弹起始位置，可根据起始位置与速度推出结束位置。

由于自己扔出的炸弹不会对自己造成伤害，故还需存储每个炸弹是否是自己扔出的。

主要的成员如下：

```
class Bomb : public PBombInfo
{
public:
    Bomb(const PBombInfo& bomb);
    Bomb(const Vec2 &source_, const Vec2 &target_);
    friend Bomb newRound(Bomb);

    void setCertain(const Vec2 &target);

    bool certain() const;
    bool canUpdate() const;
    bool isFriendly() const;

    bool moveRounds() const;
```

```
    vector<Vec2> possibleTargets() const;

    bool operator == (const PBombInfo &bomb);
    friend ostream &operator << (ostream &os, const Bomb& bomb);

protected:
    void findPossibleTarget();

    vector<Vec2> possibleTarget;
    Vec2 source;
    int moveRound;
    bool friendly;
};
```

- 成员变量
  - $possibleTarget$ 存储了所有该炸弹可能爆炸的位置
  - 若能确定起点位置，则$source$存储了这个唯一起点
  - $moveRound$记录了已经检测到这个炸弹的轮数，可供排除部分可能爆炸的位置时使用
  - $friendly$记录了这个炸弹是否是我扔出的炸弹
- 成员函数
  - 第一个构造函数构造了一个本回合首次发现的炸弹
  - 第二个构造函数通过同一个炸弹在之前几轮的信息及本回合局面信息，构造出存储该炸弹在本回合状态的bomb
  - $newRound$实现了炸弹从上一轮到这一轮的状态更新
  - $setCertain$实现了确认炸弹唯一终点后对类的更新
  - $certain$返回这个炸弹的终点是否唯一确定
  - $canUpdate$返回这个炸弹是否可能在下一轮仍未爆炸
  - 以下函数提供了访问保护成员的接口

    ```
    bool isFriendly() const;
    bool moveRounds() const;
    vector<Vec2> possibleTargets() const;
    ```

  - $findPossibleTarget$用以根据已有信息，计算可能的爆炸位置
- 运算符重载
  - 重载==用以判断两个炸弹是否为同一炸弹，判断方法为：若两个炸弹在同一轮位于相同位置，则为同一个炸弹
  - 重载<<以实现输出炸弹信息的功能

3.3.3 SituationSaver命名空间

该命名空间存储了每一轮的局面信息，并提供了一些分析局面的函数

```
namespace SituationSaver
{
vector<PlayerSight> sights;

bool dead(int roundID=curSight.round);
bool insight(const Vec2 &p,int roundID=curSight.round);
int IDofPosition(const Vec2 &p,int roundID);

Vec2 getTarget(const Vec2 &p, const Vec2 &v);
void update(const PlayerSight& sight);
}
```

- $vector < PlayerSight > sights$ 存储了每一轮的局面信息
- $dead$判断在某一轮玩家是否死亡
- $insight$判断在某一轮，位置p是否在我的视野范围内
- $IDofPosition$返回在某一轮位置p上的单位的id，如果不存在返回-1
- $getTarget$防止向逻辑传输“向一个不可达位置移动”的指令
- $update$在每回合最开始时调用，更新$SituationSaver$

3.3.4 Solver命名空间

定义了四种solver，分别处理3.2节的四种判定为玩家的情况

3.3.4.1 UnitSolver命名空间

该命名空间更新了在视野内的单位的信息，并处理了第一种情况：若一个单位的运动路线不符合村民的运动路线，则它是玩家，主要成员如下：

```
namespace UnitSolver
{
vector<Unit> units;

int unitsFind(int id);
void setPlayer(int id,Unit::SetPlayerReason);

void update();
}
```

3.3.4.2 BombSolver命名空间

该命名空间更新了视野内炸弹信息，并处理了第二种情况：若一个单位周边出现了尸体，且满足能确定这周围没有炸弹且该单位是尸体周边唯一一个单位两个条件，则该单位是玩家，主要成员如下：

```
namespace BombSolver
{
vector<Bomb> bombs;
vector<Bomb> preBombs;

int bombFind(const PBombInfo &bomb);
bool inBombRange(const Vec2 &pos);

void update();
}
```

3.3.4.3 CorpseSolver命名空间

该命名空间处理了第三种情况：若一个单位周边出现了尸体，且满足能确定这周围没有炸弹爆炸与该单位是尸体周边唯一一个单位两个条件，则该单位是玩家，主要成员如下：

```
namespace CorpseSolver
{
vector<Vec2> newCorpses;
vector<Vec2> corpses;

int findCorpse(const Vec2 &p);
void detectPlayer(const Vec2 &corpse);

void update();
}
```

3.3.4.4 AttackedSolver命名空间

该命名空间处理了第四种情况：若我的hp与上回合的差别达到某个阈值（阈值高于灼烧伤害），且周围只有一个单位，则那个单位是玩家，主要成员如下：

```
namespace AttackedSolver
{
void update();
}
```

3.3.5 Strategy类及其派生类

利用Strategy类的派生类生成我的策略，策略共8种。

3.3.5.1 Strategy类

这是主办方的提供的一个基类。

该类定义了enable()与disable()两个函数，使用户可以通过禁用掉一些策略来保证不会同时使用相互影响的策略。

该类定义了纯虚函数$generateActions$，用户通过重载这个函数来实现不同的策略。

该类的主要成员如下：

```
class Strategy
{
    friend class ActionMaker;
public:
    Strategy();
    virtual ~Strategy() = default;

    virtual void generateActions(const PlayerSight&, Actions*) = 0;

    void Disable();
    void Enable();

protected:
    bool disabled;
};
```

3.3.5.2 Strategy类的派生类

我通过继承Strategy类，定义了8种策略，策略如下：

```
// attack units in sight randomly, guarantee that attack one at most one time
class RandomAttackStrategy;
// attack players using bomb
class BombAttackStrategy;
// attack players using suck
class SuckAttackStrategy;
// move like a villager
class VillagerMoveStrategy;
// move to a player
class ToPlayerMoveStrategy;
// move to escape from bombs
class BombEscapeMoveStrategy;
// buy a bomb
class BombBuyStrategy;
// attack villager to make fortune
class ProfitAttackStrategy;
```

- $RandomAttackStrategy$ 定义了随机攻击周围村民来补充HP的策略，该策略保证每个村民至多被攻击一次
- $BombAttackStrategy$实现了向某个位置扔炸弹的功能
- $SuckAttackStrategy$实现了利用普通攻击攻击特定玩家的功能
- $VillagerMoveStrategy$实现了模仿村民运动的功能
- $ToPlayerMoveStrategy$实现了向某个玩家进行移动的功能
- $BombEscapeMoveStrategy$实现了逃离炸弹的功能
- $BombBuyStrategy$实现了有钱就买炸弹的功能

- $ProfitAttackStrategy$实现了在夜晚击毙村民以获取金钱的功能

3.3.6 主函数

```
extern "C"
{
    AI_API void playerAI(const PlayerSight sight, Actions* actions)
    {
        ZC_Strategy::SituationSaver::update(sight);
        ZC_Strategy::ActionGenerater::generateActions(sight,actions);
        ZC_Strategy::ActionGenerater::enableActions();
    }

}
```

- $ZC\_Strategy::SituationSaver::update(sight)$;更新了局面信息，并利用局面信息识别玩家
- $ZC\_Strategy::ActionGenerater::generateActions(sight,actions)$;产生我本回合的策略
- $ZC\_Strategy::ActionGenerater::enableActions()$;恢复被禁用掉的策略