涉密论文 ☐ 公开论文 ☐

浙大宁波理工学院

NINGBOTECH UNIVERSITY

毕业论文（设计）

题　　目 基于分层有限状态机的 RTS 游戏 AI 设计与实践

姓　　名 潘肖磊

学　　号 3170407010

专业班级 软件工程 171

指导教师 郭新友

学　　院 数据与工程学院

日　　期 2021/4/29

浙大宁波理工学院本科毕业论文（设计）承诺书

1.本人郑重承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2.本人在毕业论文(设计)中引用他人的观点和参考资料均加以注释和说明。

3.与我一同工作过的同学对本研究所做的任何贡献均已在论文中做了明确的说明并表示谢意。

4.本人承诺在毕业论文（设计）工作过程中没有抄袭他人研究成果和伪造数据等行为。

5.若本人在毕业论文（设计）中有任何侵犯知识产权的行为，由本人承担相应的法律责任。

6.本人完全了解浙大宁波理工学院有权保留并向有关部门或机构送交本论文（设计）复印件和电子文档，允许本论文（设计）被查阅和借阅。本人授权浙大宁波理工学院可以将本论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或者扫描等复制手段保存和汇编本论文（设计）。

作者签名：　　日期：　　年　　月　　日

致谢

大学四年很短，编码人生很长。为了感受到时间的存在，我需要意识到自己踩过校园中的某段台阶两千多次；意识到自己对河边橙黄的路灯与无人夜路上扑面而来的蛛网习以为常；意识到自己已经在通用楼的某个教室日复一日，写下了一行又一行的代码。

对于这段可能是学生时代终点的大学生活，我想我得感谢时常陪伴我奋斗在程序设计竞赛路上朋友们；感谢在大学期间指导以及对我的毕业论文提供帮助的郭新友老师；感谢给我提供物质和精神支持的家人；感谢浙大宁波理工学院以及诸多校内教师、同学、校工。

最后，我还想感谢 Mail.Ru 和 Codeforces 提供的平台支持。它们对我的学习起到了相当大的作用，让我能够借助便捷易用的网络平台和高水平的对手了解并参与到游戏 AI 的设计与实践中。既让我获得了写这篇论文的契机，也让我在一段时间内收获了许多乐趣。

摘要

本文概述了如何将分层有限状态自动机（Hierarchical Finite State Machine，HFSM）用于即时战略类游戏（Real-Time Strategy ,RTS）的 AI 系统。

具体地来讲，本文将介绍什么是 HFSM，以及如何根据 RTS 游戏的特点及其对程序设计的影响，来设计基于 HFSM 的 RTS 游戏 AI 程序。首先，本文将阐述如何围绕 CodeCraft（一个 Russian AI cup 公布的 RTS 游戏）进行具体实践，以 Russian AI cup 为网络测试平台，使用 C++实现一个 CodeCraft 的 AI，并设计自动机的状态以及状态转移以满足策略需求。然后，本文会展示一些在 AI 程序设计中会涉及到的具体子问题和它们的解决方案与对应算法，描述它们对策略的影响以及如何使 AI 在竞争中获取优势。最后，本文将以概括性的内容做收尾，包括文章中涉及到内容的总结和一些对未来的展望。

关键词：分层有限状态自动机；即时战略游戏；游戏 AI

Abstract

This article outlines how to apply Hierarchical Hinite State Machine (HFSM) to the AI system of real time strategy (RTS).

Specifically, this article will introduce what HFSM is, and how to design RTS game AI program based on HFSM according to the characteristics of RTS game and its influence on programming. First of all, this paper focuses on CodeCraft (a RTS game published by Russian AI cup). Taking Russian AI cup as the network test platform, it describes how to use C++ to realize codecraft's AI, and designs the state and state transition of automata to meet the needs of strategy and make AI gain advantages in competition. Then, this paper will introduce some specific subproblems involved in AI programming, as well as their solutions and corresponding algorithms. Finally, this paper will conclude with a summary of the content and some prospects for the future.

Keyword： Hierarchical Finite State Machine；Real-Time Strategy；Game AI

目录

1 绪论

1.1 课题背景与意义

随着几十年来人工智能技术的发展，AI 不仅在传统游戏中不断有所突破，更在本身就扎根于计算机的电子游戏中占据了重要的地位。随着电子游戏的发展，玩家数量的逐年增加,大众对内容可玩性的要求也随之增长。相应地，游戏中的 AI 行为也开始复杂多变，对 AI 角色行为进行的管控成为了游戏开发中的要点[7]。不论是单机游戏中的 NPC（Non-Player Character，非玩家角色），还是作为对手与玩家进行博弈，都离不开各类相关计算机及算法技术的支撑。

即时战略类（Real-Time Strategy ,RTS）游戏是策略类游戏（Simulation Game，SLG）一种衍生类型。RTS 继承了策略类游戏注重探索、扩张、开发、消灭的特点，同时将操作从回合制变为了实时控制。在 RTS 游戏中，玩家往往需要去控制单位探索地图获得视野、采集资源、发展经济、建造建筑、训练单位、研究科技，以取得积分或消灭对手的各种单位为目标。

RTS 游戏往往具更高的复杂性，在有着巨大状态空间的基础上还有着大量的决策选项。国际象棋的状态空间规模大约为10^{50}，德州扑克为10^{80}，围棋为10^{170}。而即使是最简单的 RTS 游戏,状态数都比上述游戏高得多，比如本课题涉及到的 CodeCraft2020 规模大约在10^{6400}至10^{12800}之间，而且每次的综合决策远比上述游戏的单一决策复杂。

如今的人工智能面对的一大挑战便是复杂多变的动态环境。同时在这种环境下我们往往无法获得完整的环境变量数据，人工智能需要在巨大的不确定性下做出决策，在例如量化交易、智能交通、天气预报、金融风控等方面有着广泛的实践。RTS 游戏包含许多有趣且复杂的子问题，这些子问题不仅与 AI 研究的其他领域密切相关，而且还与现实世界中的问题密切相关。例如：在工厂流水线操作需要优化顺序和产能平衡；模拟军事推演甚至现实军事冲突中的部队部署和调度；无人车导航需要实时的寻路和决策。

尽管 RTS 游戏在许多方面仍然不够复杂和真实，但仍能看作是对某些现实事物进行抽象而得到的模型。而这种对现实事物核心特质的简化，可以帮助我们更好地理解和解决问题。所以对 RTS 游戏 AI 的研究不仅仅在游戏领域能给玩家更

好的体验，也能使其他相关的计算机领域甚至非计算机领域受益。

1.2 课题工作

Russian AI cup 是一个 AI 编程比赛，每年都会吸引全球的爱好者和程序员参加并相互对战。2020 年的 Russian AI cup 的赛题是 CodeCraft，一个为比赛量身制定的有着简化模型的 RTS 游戏。本课题借助 CodeCraft 进行 AI 编程的实践，编写程序并提交自己的策略代码参加比赛。

使用简单而单纯的方法去用一系列复杂而缺少条理的逻辑条件判断难以优雅地实现一些更智能化且复杂的策略，所以本课题引入在游戏 AI 领域有着广泛应用的分层有限状态自动机来整合自己的算法以及策略。本文的提供的方法最后在 Russian AI cup 2020 排名为约前 20%。

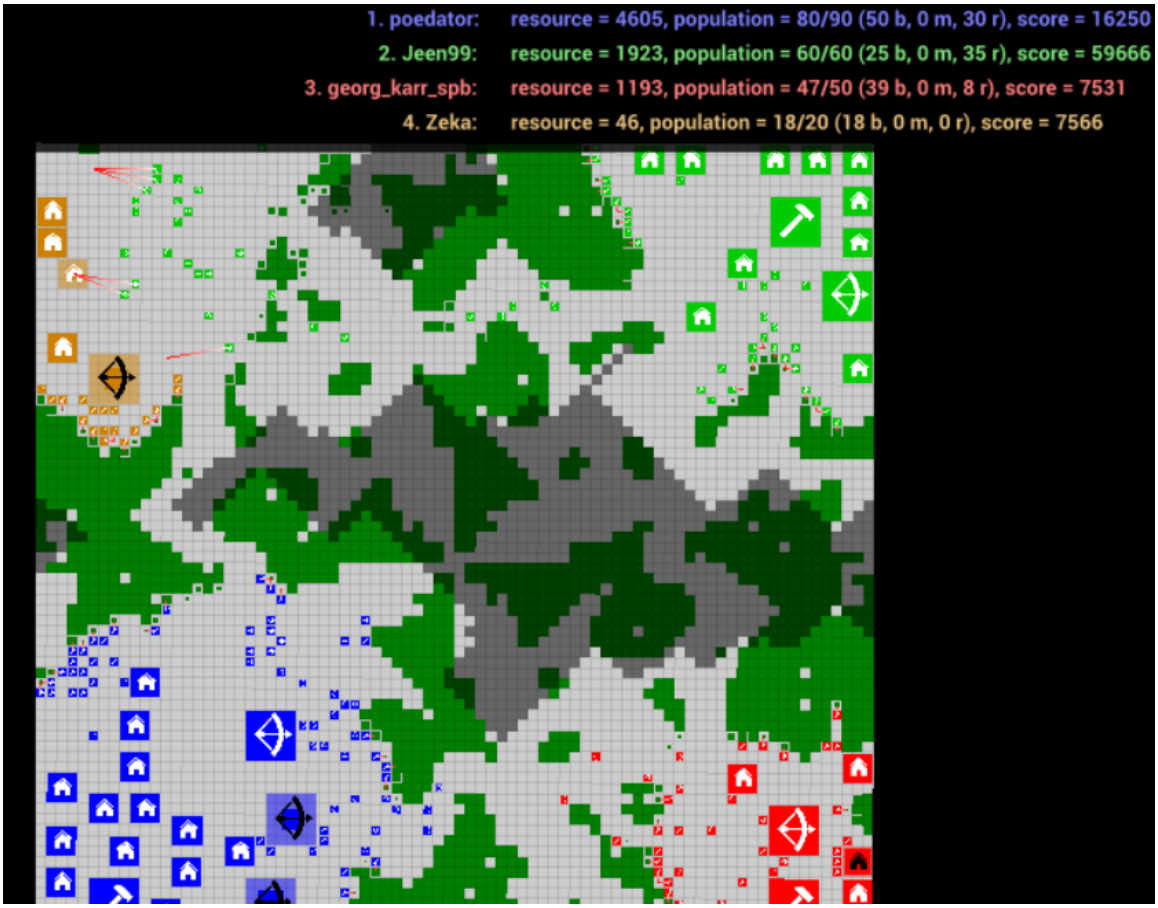


图 1　CodeCraft

本课题主要使用 C++语言，介绍如何通过 HFSM 构建一个具有一定强度的 CodeCraft 的 AI，并会介绍一些相关算法与策略、数据收集与分析过程或方法。同时由于 CodeCraft 在简化模型的前提下包含了大多经典 RTS 游戏的共性，因此

本课题会介绍一些具体战术，分析玩家之间的博弈关系，并通过一些具体的实际例子来展示组织各种策略的细节。

2　分层有限状态机

2.1　介绍

分层有限状态机（Hierarchical Finite State Machine，HFSM），是从有限状态机（Finite State Machine，FSM）衍生出的一种状态机。

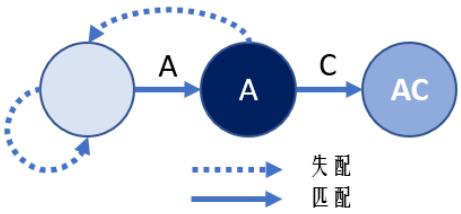


图 2　AC 自动机

许多控制系统的状态都可以用有限状态机来描述，在各类机械控制程序中都有它的身影。在计算机领域许多有限状态系统也常用它来表示，如大名鼎鼎的 TCP/IP 协议的三次握手四次挥手就可以状态机来表示。从状态机发展而来的自动机理论在字符串匹配方面也有广泛的应用，例如 AC 自动机（ACAM，Aho-Corasick Automaton）、回文自动机（PAM，Palindrome Automaton）、后缀自动机（SAM，Suffix Automaton）、非确定有限状态自动机（NFA，Nondeterministic Finite Automaton）、确定有限状态自动机（DFA, Deterministic Finite Automaton）。自动机编程的技术常用在如形式语言分析等以自动机原理为基础的算法中 [13]，早在 1963 年，Peter Naur 就在论文中将自动机编程描述为具有通用性的一种软件技术[15]，早期提到自动机编程的还有 Walter Johnson 在 1968 年发表的《Automatic generation of efficient lexical processors using finite state techniques》等论文[14]。

状态机主要由状态和状态转移两部分构成。简单地来说，状态机类似于一种特殊的图(Graph)。图中点(Node)可以认为是物体目前所属的状态(State)，边(Edge)则可以认为是环境给予物体的信息，物体根据不同的环境输入信息来决定状态的转换情况。每进行一次状态转移，物体就在图上沿着某条边移动一次。

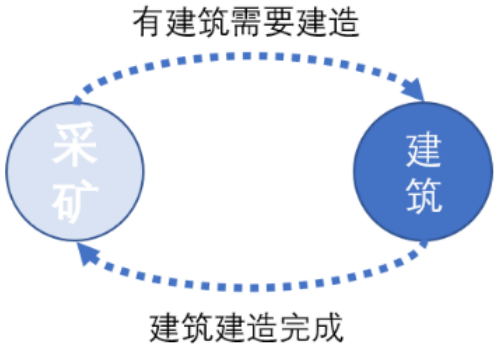


图 3　状态转移

如果物体的状态繁多，且状态转换情况复杂，那么构建状态机就变得非常繁琐。举个例子，一个人有喝水、走路、看手机三种动作，他可能边喝水边看手机边走路，或是一边看手机一边走路但不喝水，如此类推就有2^3种状态，如果行为更多，状态增加也是指数级的。所以在 FSM 的基础上又发展出了 HFSM，旨在减少 FSM 中的一些冗余内容，减少状态机设计和代码实现的工作量。原理是将状态分层分类，从而降低各种环境变量对状态影响的耦合性，而不是对每个细分状态都分类出不同状态。

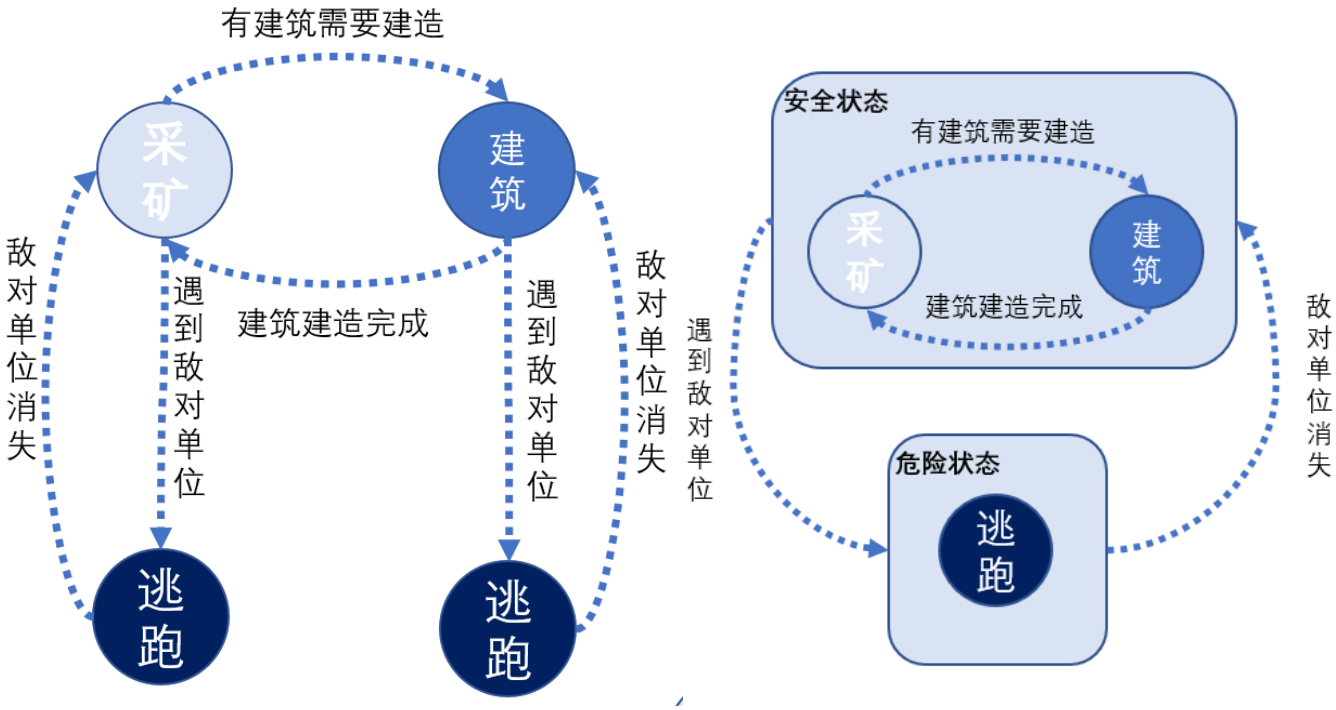


图 4　通过分层减少冗余

2.2 状态机实现

一般来说，由于状态机本身可以认为是一种特殊的图，因此可以考虑如何存储图和如何实现转移即可。例如某个节点 x，FSM 需要实现 next(x,in)，in 为环境输入，在输入后变为 x=next(x,in)。例如现在有一个字符串自动机存储：

```
map<char, int> mp[maxn];
```

有在节点 1 如果输入 a 则 next 指向节点 2 的表示方法：

```
mp[1]['a'] = 2;
```

此时可以用如下方式进行转移：

```
typedef int node;
node x = 1;
x = mp[x]['a']; //x = 2 after input
```

这样就完成了一个最基本的状态机状态转移的过程。不过一般来说还需要对这种过程进行封装来方便组织逻辑结合进状态机。

例如一个后缀自动机声明如下：

```
struct SuffixAutomaton
{
    int next[maxn * 2][ALP], p; //next 指针和节点数
    int fail[maxn * 2]; //后缀链接
    int len[maxn * 2];  //等价集合的最长子串长度
    int last;           //最后添加的最长子串所指向的节点
    void init();        //二段构造初始化
    int newnode(int l); //新建节点
    void add(int c);    //添加字符
    void f();           //...其他功能函数
} sam;
```

这种经过简单封装的自动机已经能满足基本需求，并且它的状态机形态根据不同的字符串输入会各有不同。

一般来说，如果需要更复杂的功能，就需要进一步对节点进行面向对象封装。

```
class Node
{
    Info info;              //节点信息
    Node next(Input input); //状态转移
};
```

在面相对象封装后，可以使得不同节点之间的功能进一步差异化（通过将它们分为不同的 Class），并且许多功能也可以用继承与重载操作来进行各种类型节点之间的功能复用和状态细分。

状态机的形态往往由两种方式之一来决定：

1、通过读入信息来即时地构建状态机，对于专门处理各种匹配问题的状态机来说这种输入一般是字符串或者是正则表达式。或者一些其他类型的状态机实现可以根据各类格式的配置文件读入来构建所需的形态。

2、采取硬编码的方式把自动机形态固定。好处是在一些场景下这种根据不同配置读入临时构造自动机的方式不是很有必要，硬编码可以避免许多麻烦（如配置格式的设计、encode/decode 的实现等等）。

本文主要采用第二种方法，因为在 AI 设计中一个新节点往往意味着新的逻辑和新的代码。像字符串领域的应用一样在有限的字符集（Alphabet）下根据模式串（Pattern）构建自动机的情况则比较适合第一种。

总而言之，FSM 的原理是比较简单的，在这种基础原理下发展出了很多有意义的自动机理论和应用。而 HFSM 对 FSM 的区别主要体现在状态机的设计优化，这种优化在硬编码的构建方式下极大地减少了设计状态机的工作量。

3 CodeCraft

3.1 规则介绍

CodeCraft 2020 是一种 RTS 游戏，玩家必须控制多个实体，通过收集资源、建立基地、攻击敌人等手段来获取分数最终谋求胜利。选手们创建控制 CodeCraft 中玩家的人工智能（策略），并通过网络平台与其他选手的策略竞争，以此来测试编程水平。

每个玩家（Player）的目标是获得比对手更多的分数。当达到最大 tick（计时单位,可以简单理解为帧）数或剩余玩家数小于 2 时游戏结束。在进行 1v1（决赛规则）比赛时，如果仅剩下一个玩家，他将获得足够的额外分数以赢得比赛。

游戏是在一个矩形网格上进行的，该矩形网格分为多个图块。 所有游戏中实体均为正方形，并位于一些整数坐标处。在计算距离时，计算到达目的地所需经过的图块数（Manhattan Distance，曼哈顿距离）。如图 5 中a, b两实体的曼哈顿

距离为$abs(a.x-b.x)+abs(a.y-b.y)=6$ 。

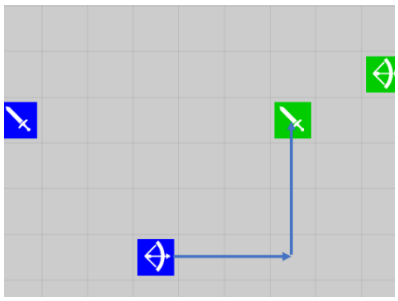


图 5　曼哈顿距离

游戏中的时间是离散的，以 tick 为单位。在每个 tick 开始时，游戏会将所有状态数据传输给参与者的策略，然后从参与者的策略中接收动作，并根据这些动作和游戏规则来更新游戏状态。然后在每个 tick 重复该过程。在 tick 达到上限时会结束游戏，如果所有策略程序都意外崩溃，也可以提早终止游戏。

崩溃的程序不再能够控制玩家的行为。在以下情况下，该策略被认为是“崩溃”的 ：

1、策略程序意外终止，或者程序与游戏服务器之间的交互协议发生错误。

2、程序响应超出了为其分配的时间限制之一。时间限制分为在每个 tick 需要在一秒内收到回应的实时响应限制，和整场游戏内总响应时间不超过四十秒的总 CPU 时间限制。

3.2　实体

实体的行为由其属性定义。最重要的属性之一定义了实体的大小。游戏中的所有实体均为正方形，边长等于定义的值。

一些实体可以移动（这些实体称为单位）。可移动实体的大小始终为 1。如果某个图块没有被其他实体占用，它们可以在一刻之间移动到相邻的瓷砖。

某些实体可以发起攻击，所有实体都有生命值并且可以被摧毁。如果实体的生命值低于或等于零，则将其从游戏中移除。可攻击实体的攻击范围有限，这是与执行攻击的目标之间的最大距离。实体每一次攻击，都会从目标中减去一定数量的生命值。

某些实体可以维修其他实体，但只有与之相邻的实体可以被维修。维修是指使实体恢复一定的生命值。维修时，目标的生命值不能超过其属性中指定的最大生命值。只有存活的实体（具有正的生命值）才能得到维修。

部分可攻击实体还可以从目标收集资源。对于每次造成伤害，都会向拥有该实体的玩家添加固定数量的资源（在实体的属性中指定）。

一些实体可以使用收集的资源建造新实体。新实体的类型受其属性中列出的功能的限制。要建造新实体，玩家必须花费指定数量的资源。对于单位（可移动实体），所需资源数量等于该单位类型的属性中指定的值加上玩家已有同类单位的数量。对于其他实体，所需资源等于其属性中指定的值。玩家还应该选择一个位置，该位置不被其他实体占用，并且位于建造者附近。新实体的初始生命值等于该实体的最大生命值，或者等于建造者实体的属性中指定的特定值。

实体在刚被建造的时候是非活动的，这意味着它不能执行任何操作。要激活实体，它必须首先达到其最大生命值。因此，如果刚被建造的实体生命值没有达到上限，则需要首先对其进行修复。

此外，建立新实体还有另一个限制。除资源外，还有另一个称为“人口”的参数。一些实体提供人口，而其他实体使用人口。为了建立新实体，所有当前玩家活动实体提供的人口总和应大于或等于所有当前玩家实体（包括新建的实体）所使用的人口总和。

实体的最后一个属性是它的视野范围。如果启用了战争迷雾，则玩家只能看到那些位于距自己控制的某个实体指定的距离不远的实体。如图 6，深灰色区域为不可见范围。

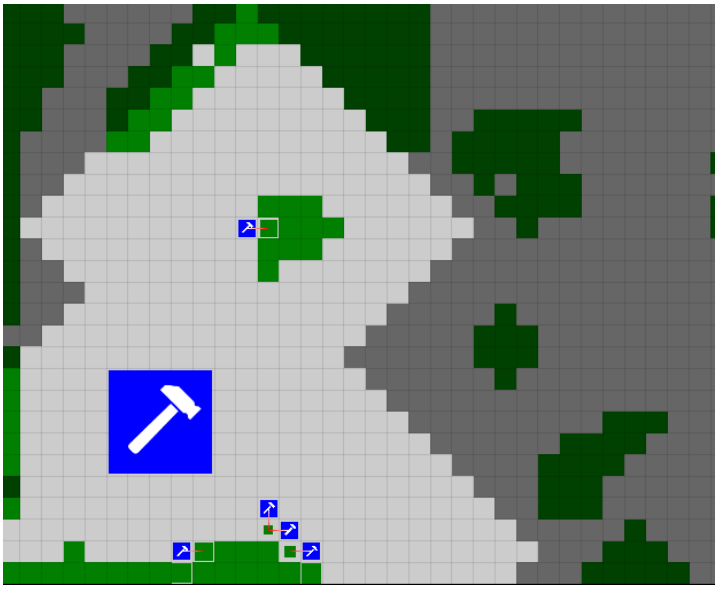


图 6　战争迷雾（深灰色部分）

游戏中有固定数量的实体类型，并且相同类型的实体具有相同的属性。这是实体类型的完整列表：

资源（Resource）：这是唯一不受任何玩家控制的实体类型。 它的大小为 1，只有被建造单位攻击才能被收集。

建造单位(BuilderUnit)：其主要目的是收集资源并建造建筑物，也可以维修其他单位。

近战单位(MeleeUnit)：攻击范围为 1 的基本攻击单位。

远程单位(RangedUnit)：远程攻击单位，生命值和造成的伤害少于近战单位，但攻击距离更大。

建造/近战/远程基地(BuilderBase/ MeleeBase/RangedBase)：这些是用来生产相应类型新单位的建筑物，可由建造单位建造，也可以扩大单位数量限制。

墙(Wall)：可以用来阻挡敌人道路的小型建筑物。

住房(House)：扩大单位数量限制的的建筑物。

炮塔(Turret)：可以攻击敌人的建筑物。由于它不能移动，因此最好用于防御。

所有实体都有自己的各类属性用于描述它自身的功能和特性，例如近战单位的属性如下：


```
{
    "MeleeUnit": {
        "size": 1,
        "build_score": 20,
        "destroy_score": 200,
        "can_move": true,
        "population_provide": 0,
        "population_use": 1,
        "max_health": 50,
        "initial_cost": 20,
        "sight_range": 10,
        "resource_per_health": 0,
        "build": {},
        "attack": {
            "range": 1,
            "damage": 5,
            "collect_resource": false
```

```
        },
        "repair": {}
    }
}
```

4 HFSM 应用

4.1 算法实现

本课题的内容用 C++作为实现语言。许多状态的分层操作和转移操作主要借助大量面相对象方法。

首先可以确定所有状态（state）的基类：

```
class HfsmNode
{
public:
    Entity selfInfo;//单位自身的信息
    virtual NodeType get_type();//获取该单位的状态类型
    virtual std::shared_ptr<HfsmNode> get_next_state();
    virtual EntityAction get_action();
};
```

在每个 tick（程序中的最小时间单位）程序需要对每个单位（Entity）执行一次 get_next_state，这是最主要的函数。它根据环境信息和本单位在前一个 tick 所属的状态来获取自身应该转换成的状态（Next State）。换句话说，它使得单位在状态机上进行了一次转移。

```
HfsmData::hfsmStates[entity.id]->selfInfo = entity;
auto state = HfsmData::hfsmStates[entity.id]->get_next_state();
if (state != nullptr)//如果返回为空则保持原有状态
    HfsmData::hfsmStates[entity.id] = state;
```

同时在每个 tick 的最后，需要借助单位的状态来确定它执行的具体动作。动作主要从环境信息和本单位在当前 tick 的状态求得。

```
entityAction = HfsmData::hfsmStates[entity.id]->get_action();
```

以处于逃离状态的单位为例，首先需要定义逃离状态，继承自所有状态的基类 HfsmNode。

```
class EscaperNode : public HfsmNode
```

然后需要重写 EscaperNode 的 get_next_state()方法，使其与父类在具体行为表现上有所区别。

```
    virtual std::shared_ptr<HfsmNode> get_next_state()
    {
        for (size_t i = 0; i < HfsmData::playerView.entities.size(); i++
+)
        {
            const Entity &entity = HfsmData::playerView.entities[i];
            if (entity.playerId == nullptr || *entity.playerId == HfsmD
ata::playerView.myId)
                continue;
            if (noEscape.count(entity.entityType))
                continue;
            if (HfsmData::playerView.entityProperties.at(entity.entityT
ype).attack == nullptr)
                continue;
            if (HfsmData::playerView.entityProperties.at(entity.entityT
ype).attack->attackRange + 2 >=
                HfsmData::dis(selfInfo, entity))
                return nullptr;
        }
        return init_node(selfInfo);
    }
```

在这个方法里具体定义了逃跑状态的单位如何根据环境输入来判断是保持还是脱离逃跑状态。

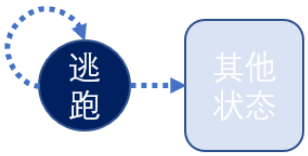


图 7　逃跑状态

然后还需要定义状态自身的行为，所以需要重写 get_action()方法。

```
    virtual EntityAction get_action()
    {
        std::shared_ptr<MoveAction> moveAction = nullptr;
        if (selfInfo.position.x > selfInfo.position.y)
            moveAction = std::shared_ptr<MoveAction>(new MoveAction(
                Vec2Int(0, selfInfo.position.y), true, true));
```

```
        else
            moveAction = std::shared_ptr<MoveAction>(new MoveAction(
                Vec2Int(selfInfo.position.x, 0), true, true));
        std::shared_ptr<BuildAction> buildAction = nullptr;
        std::shared_ptr<AttackAction> attackAction = nullptr;
        std::shared_ptr<RepairAction> repairAction = nullptr;
        return EntityAction(moveAction, buildAction, attackAction, repa
irAction);
    }
```

到这里为止已经建立了一个 CodeCraftAI 的基本框架，在这个基础上只要继续设计和添加其他类与状态转移逻辑即可。

4.2 HFSM 设计

一般的 AI 程序往往包含了诸多的算法，HFSM 这类控制算法的出现就是为了更好的将各类不同的算法以及策略整合在一起。一般来说，整体的策略是由状态机的状态和状态转移决定的，所以设计一个合理的 HFSM 尤为重要。

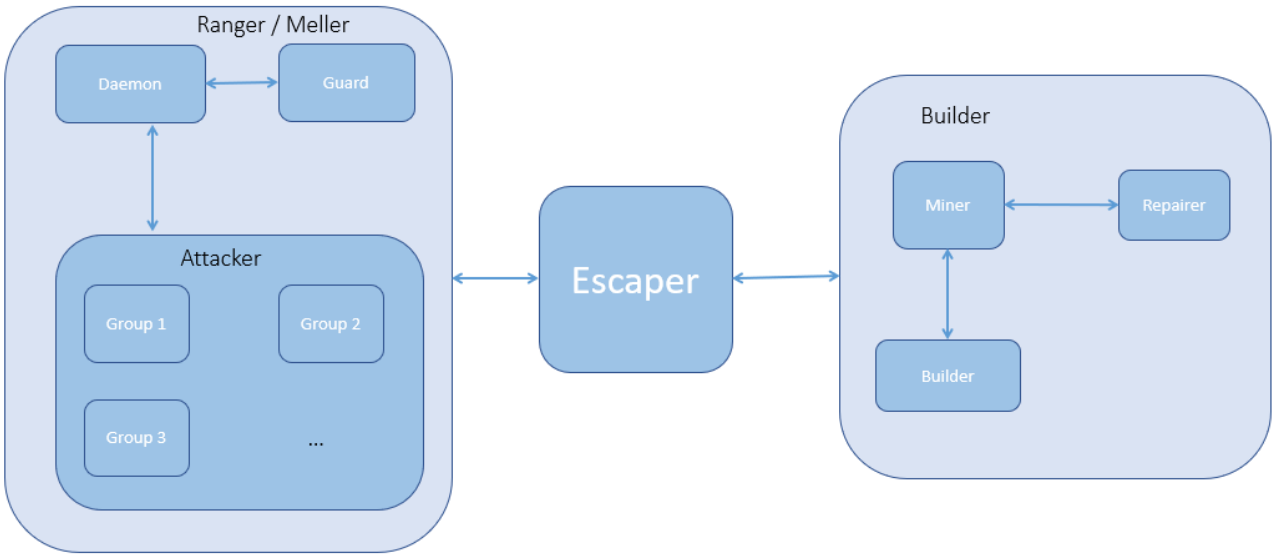


图 8　HFSM 设计（1）

以 Builder 这种单位为例，它们在一般情况下状态为 Miner(采集资源)，而在遇到敌人后会进入 Escaper(逃离)。而什么时候会主动从 Miner 转变成 Repairer（维修）或 Builder（建造）呢？这就涉及到一个 RTS 游戏与其他类型 AI 的不同之处。

RTS 中 Player 往往需要操控复数的单位，单位之间的配合不能仅靠每个单位简单的自发行为，还需要一个总体的调度流程去调动单位让它们配合完成某些任务。在这里本文将这个流程称为 Convene。

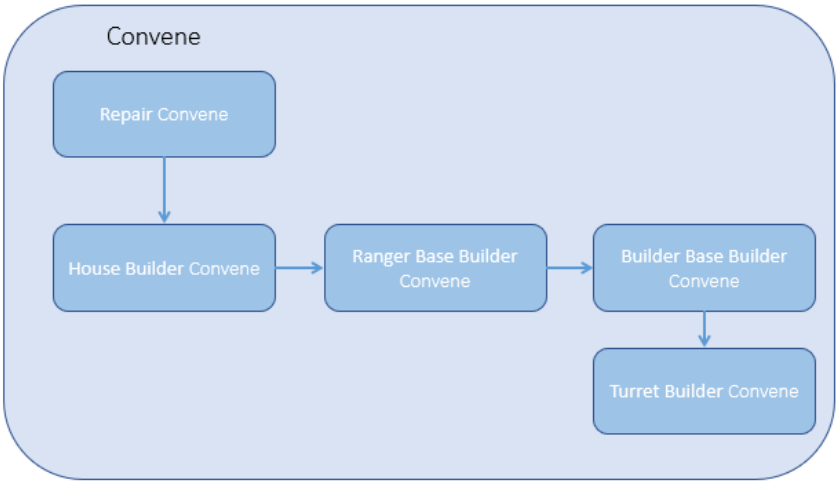


图 9　Convene 流程（1）

如图所示，对于 Builder 这种单位的 Convene，首先会征集单位去维修建筑，然后再根据优先级和策略去征集单位去建造各类建筑。在这里 Convene 作为生产者，提出任务，然后单位作为消费者，去从队列中取出任务完成。

Convene 使得 Builder 的建造建筑行为符合整体规划，而不是各自根据环境建造建筑。同时可以用同样的方式对军事单位进行编队控制。

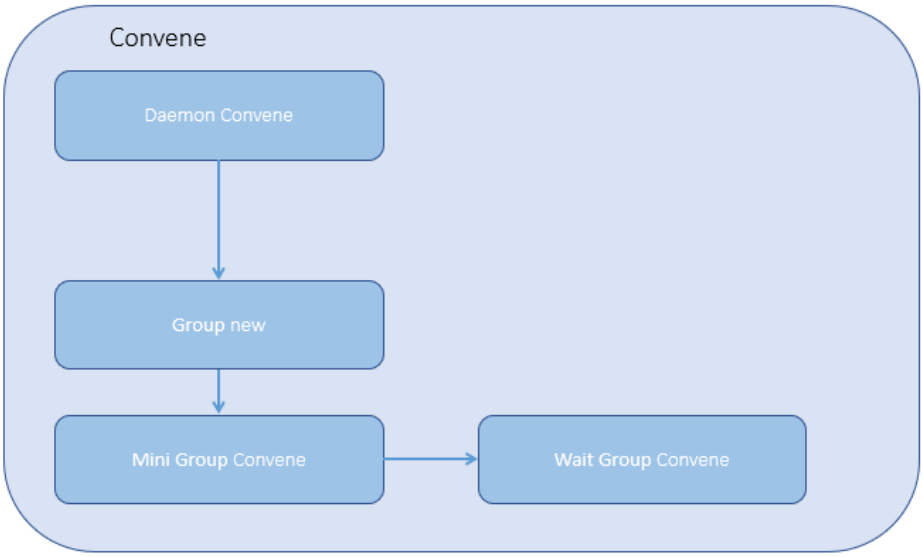


图 10　Convene 流程（2）

首先创建出一些 Group，对于每个 Group 让它发布自己的 Convene。比如现有一个 Group A，A 先通过 Convene 征集了一些单位作为自己的成员。然后会将编队中的成员集中到一起（Together）。然后再根据更高层的 Convene 去选择自己

是防守还是进攻，或者是在战损后与其他编队合并。

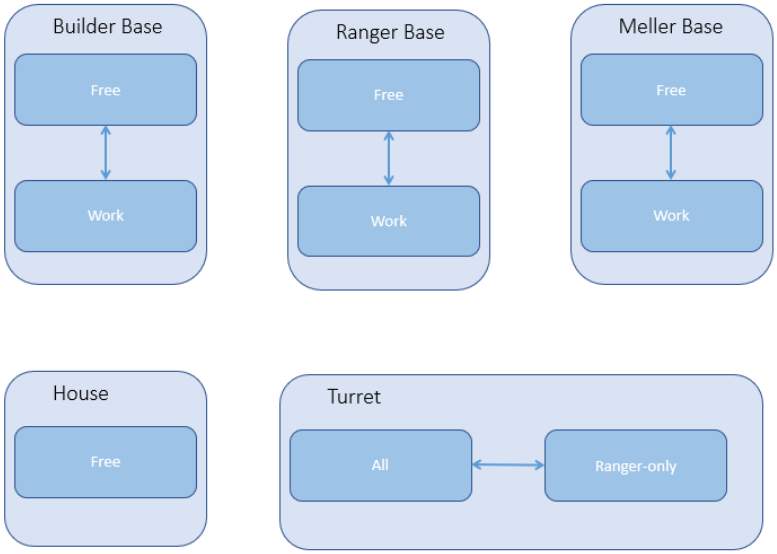


图 11　HFSM 设计（2）

建造建筑和建筑生产单位都由一个统一的 Convene 进行规划，根据需求来调动 Builder 和各类的 Base。所以建筑实体的状态就简单了许多，只需要根据分为工作与否即可。而炮塔比较特殊，它同时也是一种军事实体，根据 Convene 来选择攻击对象。

同时由于 CodeCraft 的 API 将行为分成了 BuildAction、RepairAction、AttackAction、Repair Action，所以可以根据这些分类来将每个 Action 的具体行为做一些粗略的划分。

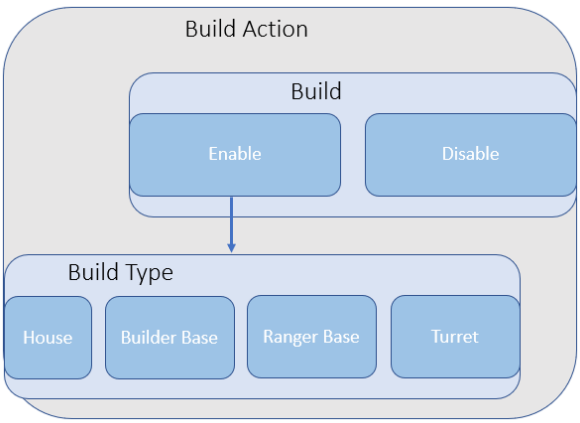


图 12　HFSM 设计（3）

这样能进一步提高代码的复用程度，许多重复的逻辑可以通过类的继承关系来共用。

```
virtual std::shared_ptr<MoveAction> get_moveAction();
```

```
virtual std::shared_ptr<BuildAction> get_buildAction();
virtual std::shared_ptr<RepairAction> get_repairAction();
virtual std::shared_ptr<AttackAction> get_attackAction();
virtual EntityAction get_action()
{
    return EntityAction(
        get_moveAction(), get_buildAction(), get_repairAction(), get_at
tackAction());
}
```

5 策略分析及设计

5.1 单位寻路

高效的寻路实现在 RTS 游戏的 AI 中是一个非常基础的部分，所有的单位在设定目标后都需要进行寻路规划。寻路能力对策略实现的影响巨大，而且在寻路问题方面已经有许多广为人知的成熟算法，对于各类变形问题也有许多可靠的解决方案。

相对传统的最短路问题，在 CodeCraft 中寻路有以下特性：

1、因为地图由二维图块构成，如果一个无向图$G<V,E>$的所有点和边可以在二维平面上不相交地表现出来，则可以称G为平面图。

2、因为地图是二维平面图，如果不考虑中间障碍，显然最短移动距离为两点间的曼哈顿距离。

3、只有相邻图块可以移动，且在图中所有边距离都为 1。

4、由于每个 tick 只有 1 秒的时限，而所需寻路的单位可能很多，近似于多源最短路问题，在时间复杂度上有所要求。

4、地图由 80*80 个图块构成，所以点的数量为 6400，同时边的数量大约为 6400*4。

首先定义包含地图中的所有点为点集V，包含所有边的边集为E，己方单位的点集为M。

相对于可以用$O(V^3)$复杂度求得全源最短路的 Floyd-Warshall 算法来说，6400 个点基本无法在 1s 内求解。根据实际情况考虑，可以认为地图为稀疏图，一方玩家单位数量基本不会超过 200，因此可以考虑用堆优化的 Dijkstra 算法以$O(Elog(E)M)$的复杂度求解多源最短路。

图 13 最短路矩阵

进一步的，由于所有的边距都为 1，所以事实上可以直接用队列替代堆，因为在 FIFO(First Input First Output)下必然满足先入队的权值小。从而可以直接转化为多源 BFS（Breadth First Search，广度优先搜索），以$O(ME)$的复杂度求解所有单位对目标的最短路径。

5.2 在 FFA（Free For All）模式下的博弈分析

FFA 模式为不分敌我的多方混战模式。在 CodeCraft 中的 4 Player FFA 模式下，分析 4 个 Player 之间的博弈关系很有必要。

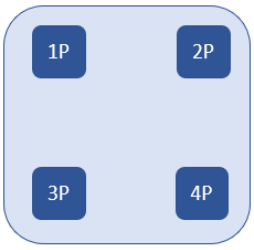


图 14 4 位玩家的初始位置分布

通过对许多其他选手的策略的观察与分析，可以粗略地分几种策略类型：

1、均衡策略：均衡地发展经济和军事，并均匀的对周边其他玩家进攻。

2、快攻策略：放弃一些经济发展，选择在早期做更积极的进攻。

3、单一进攻策略：相比均衡策略，选择周边的一个玩家进攻。

4、防守策略：建立防御工事，进攻性较弱。

虽然许多选手的策略单纯靠观察很难分析它们的行为逻辑，但是大体还是可以归类于这些策略中的一种或者多种混合。

均衡策略是被使用的最多的策略，选手之间主要差距在于各类细节操作（Micro Control）的优化水平，比如寻路决定了 Builder 的资源采集效率和建造

建筑效率，军事单位的攻击目标选择决定了战损比。这种均衡的策略能够应对各种场景，并通过在细节上积累的优势取胜。

而快攻策略很大程度上在经济建设方面减少了工作量，有时能轻松取得胜利。但是这种策略很依赖于随机地形的优劣（如果临近玩家间地形比较开阔则会对快攻有优势），所以在高胜率策略中基本不会见到快攻。

单一进攻策略事实上比较符合人类玩家的直觉，人类玩家在多线同时操作时无法像 AI 一样能够做出比较好的操作，而且聚拢力量先让一个敌人失去战斗力似乎也能产生很大优势，而且单一的目标也为编码提供了便利。但是事实上这样的策略却不一定适合 AI 程序，因为一旦展开与相邻一家展开消耗，就容易被另一家趁虚而入，对于这种时机(Timing)的把握是非常困难的。

由于 Player 初始位置之间的曼哈顿距离不尽相同。以$1P$为例,显然其与$4P$的距离比其他两方更远，有：

$$distance(1P, 2P) = distance(1P, 3P) = \sqrt{2} * distance(1P, 4P)$$

如果$1P$一直采取防守的策略，而$2P$、$3P$、$4P$都采取比较均衡的策略。$4P$会相比$1P$承受更多的$2P$、$3P$的压力，导致最先被消灭。因为$2P$与$3P$距离很远，所以之后$1P$就会同时承受$2P$、$3P$的主要进攻。最后的积分排名较大可能是$2P = 3P > 1P > 4P$。而在 CodeCraft 的 4 人 FFA 规则中，单场获取积分根据场内分数排名分别是 8/4/2/1，避免在单场中落到 3、4 名对积分排名更重要。所以过于消极的防守策略也效果不佳。

5.3 微操（Micro Control）

程序比起人类而言，一大优势计算能力和操作速度。人类受限于每次操作都需要经过一个动手的物理过程，即使最顶尖的玩家，APM（Actions Per Minute，每分钟操作数）也大约在 200-500 之间，而程序却可以很轻松的使 APM 达到 1000 以上。这方面的实现水平很大程度影响了 AI 程序进行游戏给人的观感，一些高水平的实现（例如星际争霸中的悍马 2000）在游戏中的表现让人叹为观止。

图 15　星际争霸 2

显而易见的是，在操作复数单位的时候，一些最优的操作是非常细致的，同时需要大量计算和大量操作。其中一些操作会为选手带来巨大优势，以《星际争霸 2》为例，其中有一种单位叫追猎者，它除了可以进行远程攻击外，还有一个技能可以进行空间跳跃，传送到不远处的位置。如果能在对战中将靠前的生命值较低的追猎者即时跳跃到后排，就可以让伤害分摊到其他单位，从而获得优势。

而在 CodeCraft 中，远程单位（RangerUnit）就有着较好的微操余地，比如我们可以通过局部的计算得到单位攻击目标的最优解。例如敌我双方此时都有 2 个远程单位分别即为我方的 Pa,Pb,和对方的 Da,Db，而远程单位的 max_health 为 10，攻击伤害为 5。显然让 Pa,Pb 在第一个 tick 同时将攻击目标设置为 Da,或者同时设置为 Db 是最优解。

Tick/ Unit	Pa	Pb	Da	Db
0	10/10	10/10	10/10	10/10
1	5/10	5/10	10/10	0/10
2	5/10	0/10	0/10	0/10
3	5/10	0/10	0/10	0/10

图 16　单位生命值随时间的变化

如图 16 所示，如果我方采用最优策略，可能在同样兵力下最终会多存活一个单位从而积累优势。

在 CodeCraft 中一个 tick 内单位只能选择移动或者攻击，不能同时进行两种操作。这就意味着不仅需要在攻击目标间做决策，也需要在攻击还是移动间做权衡。

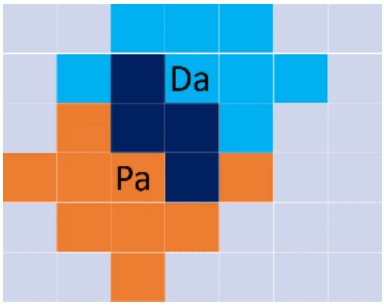


图 17　攻击范围示例

图 17 中橙色为 Pa 的可攻击范围，浅蓝色为 Da 的可攻击范围，深色为两者的交叉部分。此时如果 Pa 主动靠近 Da，就可能会受到先手攻击，从而在 1 对 1 的情况下让 Da 以 5/10 的优势存活。所以此时的最优策略应该是后撤或者原地等待。

还有一种叫放风筝的策略。远程攻击单位在遇到近战单位时，在距离较远处可以进攻，等到双方接近后开始远离对方，以此来避免受到对方攻击，同时又有机会消灭对方。

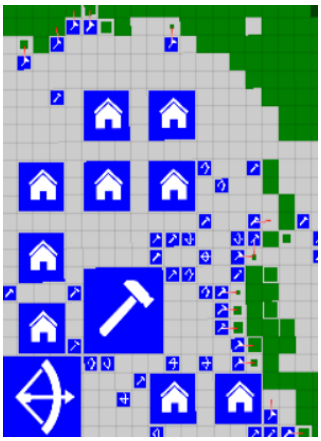
这些操作或策略情况复杂，且各种状态会相互覆盖，需要复杂的分支逻辑判断。但是具体实现中可以使用 HFSM 将其状态分层，对情况做粗略的分类后再在状态机中添加具体的实现逻辑即可，大大减少了编码和维护难度。

5.4　建筑规划

在建设基地方面，除了在合适的时间建造建筑之外，选择合理的位置建造建筑也十分重要。在部分游戏中（如帝国时代 2），建筑相对不容易被破坏，所以高水平玩家会选择在外层建造建筑来保护内部从事生产的脆弱单位。

图 18　帝国时代 2

具体的来说，一个建筑具有坐标和边长两种位置属性。坐标表示它的左下角第一个图块所在的位置(X,Y)，边长L表示它所占空间是以左下角为顶点的一个四边长度为L的矩形。合理的规划应该使得$||X|+|Y||$尽可能小，且不会阻碍单位移动。进一步的，还可以让它们的位置与建筑工尽可能的接近，这样就能提高建筑效率。



可以考虑这样一个方式，让建筑等距地排列，中间留出空隙以供其他单位移动：设间距为k，让边长为L的所有建筑的(x,y)坐标满足$x \equiv C_1 \ (mod\ k+L)$，$y \equiv C_2 \ (mod\ k+L)$，$C_1$、$C_2$为可选常数。同时也可以对$||x|+|y||$做约束，这样就可以轻松地求解出可行的坐标集合$S$。然后调度建筑工单位以消费者的方式获取$S$中的元素去指定坐标建造建筑即可。

图 19　CodeCraft

5.5　数据爬取和分析

在改进程序和策略的过程中，知道当前改动对比之前的版本（Baseline）是否起到正面影响是很重要的。所以需要编写一个 Python 爬虫从 russianaicup.ru 网站收集比赛数据。

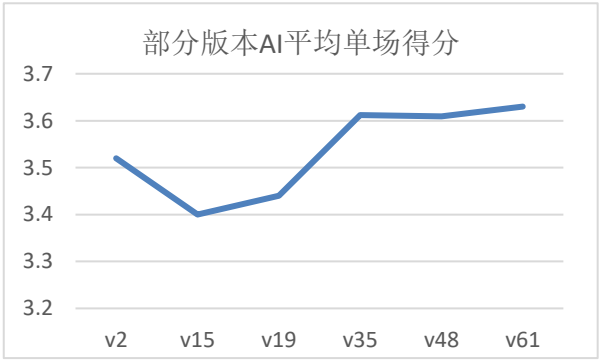


图 20　数据统计

经过一些观察发现，每个比赛有 game_id 作为标识符，但是这个标识符却不能直接用 requests 获取，只能用 selenium 库模拟 chrome 浏览器的操作来获得所有比赛记录页面代码，然后通过正则匹配获得 game_id。在获取了 game_id 之后就可以用 requests 发送 get 请求，获得页面后用 BeautifulSoup 调用 lxml 来解析页面文本获得 csrf_token。之后就可以用已经获得的 game_id 和 csrf_token 两个字段构造 post 请求来获得具体的比赛数据。

这里以平均单场得分作为衡量策略强度的指标。在 CodeCraft 中，单场为 1、2、3、4 位的得分分别为 8/4/2/1。然后对比新策略与 Baseline 的平均得分来决定是否替换 Baseline。不过这种衡量方式有较大缺陷，一是在场次少的情况下平均得分的参考价值不大，二是其他选手也会一直改进自己的策略。另一种衡量方式是在本地进行新策略与 Baseline 的对战测试，但是针对自己的旧版本程序有效的策略不一定具有普适性，在没有其他选手的源代码的情况下也不能模拟与各类其他选手对战的情况，反而可能会在一些子问题上陷入类似过拟合的情况。

6 总结与展望

RTS 游戏的 AI 程序既涉及到许多具体的工程实践，也涉及到各方面的算法优化。一方面由于它天然的复杂性使得复杂的逻辑需要被用 HFSM 有序合理的组织，另一方面各类子问题需要诸如图论、搜索、贪心、数论等算法与知识才能较好的解决。

从 2020 年 12 月到 2021 年 2 月，从一开始官方提供的简单样例程序经过几十个版本的迭代变成了上千行的 HFSM 代码。本文实现的程序在 russianaicup 平台上已经与来自世界各地的其他 AI 进行了总计上万场对局，最终版本的程序进行了 821 场正式对局（即积分排位），单场排名 1、2、3、4 位的次数和百分比分别为 155(19%)、343(42%)、229(28%)、94(11%)，总积分排名为 169/838。

虽然 HFSM、行为树等算法在整合逻辑和策略上已经有较大作用，但是具体的决策仍然需要借助其他各种算法和人类经验来解决和调优。这方面强化学习（Reinforcement Learning, RL）等新兴的机器学习算法应该有较大的发挥空间，希望参加来年的 Russian AI cup 时能有机会引入这类算法，并通过往年的经验在新的赛题中获取更好成绩。

7 参考文献

[1] D Silver, A Huang, CJ Maddison, A Guez，L Sifre，GVD Driessche，J Schrittwieser，I Antonoglou, V Panneershelvam, M Lanctot. Mastering the game of Go with deep neural networks and tree search[J]. Nature, 2016, 529: 484~489

[2] 潘才钦. 基于 OpenGL 的游戏 AI 行为树编辑器的设计与实现[D]. 厦门, 厦门大学, 2015

[3] 胡俊. 游戏开发中的人工智能研究与应用[D]. 成都，电子科技大学，2007

[4] S Ontanon，G Synnaeve，A Uriarte，F Richoux. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft[J]. IEEE Xplore ,2013, vol. 5: 293~311

[5] 万潭凯. 神经网络在即时战略游戏中的应用[D]. 福州，福州大学，2014

[6] 潘伟民，彭成. 教学游戏 AIFSM 实现策略研究[J]. 新疆师范大学学报, 2008，vol. 4: 24

[7] 石俊杰. 基于有限状态机的游戏角色控制系统设计与实现[D].武汉，华中科技大学，2018

[8] Mat Buckland. Progamming Game AI by Example 游戏人工智能编程案例精粹[M]. 中国，人民邮电出版社

[9] 沈宇, 韩金朋, 李灵犀, 王飞跃. 游戏智能中的 AI——从多角色博弈到平行博弈[J]. 智能科学与技术学报, 2020, 2(3): 205-213

[10] Tao Qin, Tie-Yan Liu, Hsiao-Wuen Hon, Junjie Li, Qiwei Ye, Li Zhao. Suphx: Mastering Mahjong with Deep Reinforcement Learning[EB/OL]. arxiv.org/pdf/2003.13590.pdf, 2020

[11] 陈仕镇. 基于神经网络和有限状态机的游戏 AI 决策引擎设计与实现[D].厦门，厦门大学，2016

[12] 何汶俊. 基于行为树的游戏 AI 设计与实现[D].成都，成都理工大学，2018

[13] Aho Alfred V, Ullman Jeffrey D. The theory of parsing, translation and compiling 1[M].美国，Prentice-Hall.

[14] Johnson W. L, Porter J. H , Ackley S. I, Ross D. T. Automatic generation of efficient lexical processors using finite state techniques[J]. Comm ACM. 1968, 11 (12): 805–813. doi:10.1145/364175.364185

[15] Naur Peter. The design of the GIER ALGOL compiler Part II. BIT Numerical Mathematics[J]. 1963, 3 (3): 145–166. doi:10.1007/BF01939983