过程

将会在这部分阐述我们具体完成这项工程的过程，也算是我们小小奋斗史的简略记录。在这次合作项目中，我们收获的不止是算法和策略的思考，也有团队合作的宝贵经验。

我们的项目从无到有，从基础到慢慢优化，出于管理以及对比需求，我们一直都有在做版本记录，直至我们比赛之前，有几十个版本，当然并不是指我们有几十种策略，我们的棋力也并不是随着版本号线性增长的，但是我们的想法和工作都累积在这些版本之中。在之后的报告中，我将会选择具有代表性的版本进行说明。我们靠 git 管理代码，项目原本在 github 上后来转到了 bitbucket 上面。

工作大致分为功能方法实现，优化方法加速，debug。三种工作每个时期都有，但是侧重都不太相同，但是总的来说基本思路就是前期集中大量开发功能方法，后期根据测试结果做 debug 和优化。

阶段一：集中完成各类方法

我们四个人都是没有围棋经验的人，对于围棋的知识仅仅停留在知道规则而已，并且对于围棋人工智能的东西也知之甚少，同时因为初期基础代码不便于团队合作所以我们一开始的分工是两个人做开发两个人做理论学习。

在此期间我们完成了 version1.x 和 version 2.x 系列

Version 1.x 基本数据结构和基础棋盘

版本 1.x 系列，是在 brown 的基础上，规划工程数据结构，完成新数据结构下的棋盘。因为 Brown 的本身的数据太过单一，不足以支持我们的工程，我们也用空间换时间的方法新添加了一些记录除去棋盘状态之外的数据。

Version 2.x 基本单线程 Uct

Version 3.x 更换棋盘结构以及加速对棋串的维护

我们在这里做了一个对比试验，利用之前加入的 uct，做 uct 在 brown 上和在 version1.0 上面的性能比较，以 10s 能模拟的盘数作为评判标准。发现之前所抽出的高层结构降低了 uct 的性能至少两倍。所以我们总结原因是结构太复杂，耦合性高，同时可能是因为结构体的嵌套导致分配空间时增大时间开销（by TA）。所以我们决定放弃之前的结构，做了最简单的结构，虽然从代码上来说看起来并不优美，但是效率毕竟还是最重要的。而之后我们一系列的修改也证明这次的重写是有意义的，因为在简单的结构上增删更为高效。

我们用并查集对棋串的维护做了优化，加快了棋盘速度。

Version 4.x uct 多线程优化，导入棋谱，加入四种 pattern

到 4.x 版本我们就基本完成了一开始预设的功能方法。

多线程 UCT
四种 pattern
开局定式
棋谱搜索
简单技巧，比如提前吃字，立即救子，提气，做假眼等。

此时整个项目的组合方式是：开局定式->棋谱前 20 步->如果有特殊情况用技巧返回点不然 uct 搜索（uct 搜索中有 pattern 做优化）

阶段二：测试寻找问题，排除逻辑 Bug

Version 5.x – version 7.x

利用助教提供的上一届的 exe 反复对战测试观察问题，因为这时候的 debug 难度要比之前都高很多，毕竟每个功能都有可能出现逻辑 bug。顺便也在这个时期观察下其他 AI，猜一下他们使用的方法。

观察点：
是否有开局定式
是否有 trick，比如提前救子，优先吃子，特定走法
是否有 pattern
是否使用多线程

GnuAplus_v5 对战记录
GnuAplus_v5 先手

GotodieOut	**Demon**	**Gg**	**IAmHungry**	**IdiotGo**
w+17.5	W+45.5	W+1.5	W+15.5	B+16.5

虽然只是部分，但是几乎是全败的记录，但是从对战中确实发现了不少问题。
以下是我们发现的问题以及解决方法，只做部分摘要

现象	**问题**	**解决**
棋局还有可以下的地方却报 illegal 错	我方气判断错误导致合法的地方 illegal	重新计算气
棋下的过于散		重新计算气
棋谱单一		
棋会填自己的眼	因为救子的优先级过高加上气的计算有误导致	重新计算气

可以很明显的看出，我们大部分问题的根源是对于气的计算。我们重新计算了气，然后同时保留了真气和伪气两种计算方式和对应数据。是为之后优化多留一个选择。

阶段三：优化围棋项目

这大概是最难的一部分了。不是实现上的困难而是策略的问题，就是说这阶段做得多不等于做得好，做的东西在不在点上很重要，毕竟测试要花大量的时间。这一阶段里我们以优胜和找到优化方向为目标和上一届的程序模拟了大量的对战。因为前面已经对于我们的技术路线有很详细的解释了，这里对于技术方面就不再赘述了而是从几个重要的组成部分来简述我们的优化过程。

UCT

UCT 无疑是我们整个项目最核心的部分。之前我们都是在为能让他搜的更深更多而做努力——努力减少其他方面带来的开销，甚至不惜放弃 uct 中的策略优化和剪枝。纯 UCT 情况下我们达到 9.5s 内 450K。虽然用纯 UCT 我们已经打败了上届半数以上的成果，但是我们依然觉得我们的 UCT 比他应当表现出的水平要低（这是和别组同学交流而得）。后来我们测试发现并不是模拟的越多越好，虽然加速还是有意义的，但是时间可能应该拿来做别的事情，也根据另外的论文调整了 uct 参数。

策略

我们想过也实现了很多策略，从简单的吃子救子到复杂的前中后场不同策略组合，但是因为测试很难，除非某种组合能 90%赢过纯 UCT 才能说这种组合提高了棋力，但是测试时间太过漫长，我们最终获得的组合也只是一个相对较优的解。因为对 UCT 的重新认识，让我们再一次重视了策略的重要性，也有相关论文给出了较优的排列组合，但是最终我们也是根据自身的情况做了一些调整。

减少时间开支

这一部分一直是我们比较重视的地方，在这里付出的心血也较大。后来我们想了很多方法，比如置换表，GPU 编程等。后期在 UCT 上加上各种策略之后，搜索盘数又成为了我们的一个瓶颈，置换表从数据上来讲，确实达到了 1/10 的重复率，但是实际实验结果是反而因为构建置换表而增加了时间开支。最后也尝试用 GPU 编程，但是因为种种原因没有完成，没有切实的对比数据，这里也就不再赘述了。后期所做的优化真正起作用的还是对数据结构的优化。

以上就是我们完成这个项目的过程概述了，这里毕竟是在做完之后完成的报告，看起来还有些条理，在实际开发过程中，还有太多无法一一表述清楚的纠结过程，我们用了很多的时间走了很多的弯路也遇到了很多的坑，很多应该是可以避免的，但是总的来说项目合作还是很愉快的。