

1 介绍	11
1.1 关于 GNU Go 和本手册	11
1.2 版权	12
1.3 作者	13
1.4 致谢	13
1.5 GNU Go 任务表	14
1.5.1 总体	14
1.5.2 小项目	15
1.5.3 长期项目	15
1.5.4 思想	17
2 安装	19
2.1 GNU/LINUX AND UNIX	19
2.2 配置选项	20
2.2.1 缓存	20
2.2.2 默认级别	21
2.2.3 DFA 配置选项	21
2.2.4 其他选项	21
2.3 在微软平台下编译 GNU Go	22
2.3.1 Win 95/98, MS-DOS 和 Win 3.x 下使用 DJGPP	23
2.3.2 Windows NT/95/98 下使用 Cygwin	24
2.3.3 Windows NT/95/98 下使用 MinGW32	24
2.3.4 Windows NT/95/98 使用 Visual C 和工程文件	25

2.3.5 Windows NT/95/98 下运行 GNU Go	25
2.4 MACINTOSH	26
3 使用 GNU GO	27
3.1 获取文档	27
3.2 通过 CGOBAN 运行 GNU Go	28
3.3 其他客户端	29
3.4 ASCII 接口	30
3.5 EMACS 的 GNU Go 模式	31
3.6 GMP 协议和 GTP 协议	34
3.7 电脑围棋锦标赛	34
3.8 SGF 格式	34
3.9 启动 GNU Go: 命令行格式	34
3.9.1 一些基本选项	34
3.9.2 其他普通选项	36
3.9.3 其他影响实力和速度的选项	37
3.9.4 Ascii 模式选项	39
3.9.5 开发选项	39
3.9.6 试验选项	43
4 GNU GO 引擎纵览	45
4.1 获取信息	45
4.2 着手生成	46
4.3 着手评估	48

4.4 详细的事件序列	48
4.5 路线图	50
4.5.1 在 "engine/" 中有以下文件:	51
4.5.2 "patterns/" 中的文件	54
4.6 代码风格和体例	56
4.6.1 代码体例	56
4.6.2 跟踪	56
4.6.3 断言	57
4.6.4 <i>FIXME</i>	57
4.7 源码浏览	58
5 分析 GNU GO 的着手	59
5.1 解释跟踪	59
5.2 输出文件	60
5.3 检查识别码	60
5.4 检查 OWL 码	61
5.5 GTP 和 GDB 技术	61
5.6 调试盘	62
5.7 数子	62
5.8 彩色显示	62
5.8.1 棋块显示	62
5.8.2 眼位显示	63
5.8.3 模样显示	63

6 着手生成	65
6.1 简介	65
6.2 着手目标生成	65
6.3 各种着手目标详解	67
6.3.1 攻防着手	67
6.3.2 威胁攻防	68
6.3.3 攻防多用着手	68
6.3.4 切断和联络	68
6.3.5 赢取对杀的着手	69
6.3.6 做眼和破眼	69
6.3.7 坏棋	69
6.3.8 实地	69
6.3.9 攻防棋块	69
6.3.10 组合攻击	70
6.4 着手评估	70
6.4.1 实地价值	71
6.4.2 战略价值	72
6.4.3 棋形因数	72
6.4.4 最小值	72
6.4.5 第二值	72
6.4.6 威胁和后续价值	73
6.5 终局	73

7 棋串和棋块	74
7.1 棋串	75
7.2 合并	79
7.3 联络	79
7.4 后手眼和假眼	80
7.5 棋块	81
7.6 彩色棋群显示	86
8 眼和后手眼	88
8.1 局部博弈	88
8.2 眼位	89
8.3 眼位的局部博弈	90
8.4 实例	91
8.5 图形	92
8.6 眼形分析	93
8.7 眼的局部博弈值	93
8.8 后手眼和假眼拓扑	95
8.9 带劫的眼拓扑	96
8.10 假边缘	100
8.11 OPTICS.C 中的函数	100
9 模式代码	104
9.1 纵览	104

9.2 模式属性—分类	106
9.2.1 条件模式属性.....	106
9.2.2 操作属性.....	107
9.3 模式属性—值	108
9.4 辅助函数	109
9.5 自动辅助和条件	110
9.6 自动辅助操作	111
9.7 自动辅助函数	112
9.8 攻防数据库	118
9.9 联络数据库	119
9.10 联络函数	121
9.11 调整模式数据库	121
9.12 实现	124
9.13 对称和转换	124
9.14 实现细节	125
9.15 “网格”优化.....	126
9.16 定式编译器	127
9.17 定式中的征子	129
9.18 角部匹配.....	130
9.19 编辑模式的 EMACS 模式	132
10 DFA 模式匹配.....	133
10.1 DFA 简介	133

10.2 DFA 是什么	134
10.3 DFA 模式匹配	136
10.4 构建 DFA	137
10.5 增量算法	139
10.6 一些 DFA 优化	140
11 战术识别	141
11.1 识别基础	141
11.1.1 识别代码组织	142
11.1.2 返回代码	142
11.1.3 识别修剪和深度参数	142
11.2 局面 HASH	144
11.2.1 Hash 值计算	144
11.2.2 Hash 表组织	145
11.2.3 Hash 结构	146
11.3 持续的识别缓存	147
11.4 劫争操作	148
11.5 劫争实例 1	150
11.6 劫争实例 2	151
11.7 替代 KOMASTER 设计	152
11.7.1 2.7.232 基本方案	152
11.7.2 2.7.232 修订版本	153
11.8 超串	153

11.9 调试识别代码	154
11.10 联络识别	157
12 基于模式的识别	158
12.1 OWL 代码	158
12.2 组合识别	160
13 影响函数	162
13.1 影响的概念	162
13.2 地、模样和区域	163
13.3 引擎中采用影响函数的地方	163
13.4 影响和实地	164
13.5 实地值的细节	167
13.6 影响函数的核心	167
13.7 影响算法	168
13.8 渗透度	169
13.9 逃跑	170
13.10 打入	171
13.11 包围	174
13.12 影响模块所用的模式	176
13.13 影响的彩色显示和调试	177
13.14 使用 VIEW.PIKE 调整影响	178
14 模样的另一方案：BOUZY 的 5/21 算法	181

14.1 模样历史	181
14.2 Bouzy 的 5/21 算法	182
15 棋盘库	185
15.1 棋盘数据结构	186
15.2 棋盘数组	186
15.3 增量棋盘数据结构	188
15.4 棋盘函数	191
16 在内存中操作 SGF 树	194
16.1 SGF 树数据类型	195
17 GNU GO 的 API	196
17.1 如何在你自己的程序中使用引擎：开始	197
17.2 引擎中的基本结构	197
17.3 棋盘状态结构	198
17.4 局面函数	199
17.5 对局操作	200
17.5.1 <i>Gameinfo</i> 函数	200
18 工具函数	202
18.1 通用工具	202
18.2 打印工具	207
18.3 棋盘工具	209
18.4 “ENGINE/INFLUENCE.C” 中的工具	213

19 GTP 协议	215
19.1 GTP 协议	215
19.2 GTP 方式启动 GNUGO	215
19.3 协议应用	217
19.4 METAMACHINE	218
19.4.1 独立的 <i>Metamachine</i>	218
19.4.2 GNU GO 作为 <i>Metamachine</i>	219
19.5 增加新 GTP 命令	219
19.6 GTP 命令参考	221
20 回归测试	229
20.1 GNU Go 中的回归测试	229
20.2 测试包	230
20.3 性能测试	230
20.4 运行 REGRESS.PIKE	232
20.5 用 EMACS 查看测试	233
20.6 HTML 回归查看	233
20.6.1 配置 HTML 回归视图	233

1 介绍

GNU Go 3.6 是围棋对弈程序，其开发中的版本位于 <http://www.gnu.org/software/gnugo/devel.html> 可供下载研究。

如果有兴趣支持本项目，可联系 gnugo@gnu.org。

1.1 关于 GNU Go 和本手册

1.2 版权	版权
1.3 作者	作者
1.4 致谢	致谢
1.5 GNU Go 任务表	任务表

1.1 关于 GNU Go 和本手册

挑战电脑围棋的目的不是打败电脑而是对计算机编程。

在电脑象棋领域，实力较强的电脑程序已经有能力参与最高级别的比赛，甚至挑战像卡斯帕罗夫这样的棋手。电脑围棋程序甚至连业余初段的水平都达不到。挑战就是写出这样的程序。

确切地说，已有的电脑围棋程序的实力已经足可作为一个对手，写出一个真正强大的程序前景乐观。

GNU Go 的实力越来越强。GNU Go 3.6 的水平大约为 9 级。

直到如今，围棋程序始终是只以目标二进制码方式发布的。这些程序中的算法都是保密的。除了作者没人能够对此进行检查和褒贬。结果任何一个想投身电脑围棋的人都必须一切从头开始。这也许就是围棋程序至今未能达到一个更高水平的原因。

与大多数围棋程序不同，GNU Go 是自由软件。它的算法和源码都是随文档公开的，任何人都可以免费地用作研究和扩展。希望这种自由可以给 GNU Go

带来更强的竞争优势。

本文档是 GNU Go 的手册。无疑这里会有不准确的地方，最终的文档包含在源码及其注释中。

本手册开始的三章是面向普通用户的。第三章是用户指南。本书的其他部分是面向编程者，或对 GNU Go 的内核好奇的人。第四章是引擎的总览。第五章介绍深入 GNU Go 引擎并探知其为何走出某一手棋的各种工具。第六到第七章是 GNU Go API 的程序员参考手册。余下的几章更详细地深入 GNU Go 内部的各个方面。

1.2 版权

除以下注明的以外，1999、2000、2001、2002、2003 和 2004 版权属于自由软件组织。

所有文件都在 GNU General Public License 下（参见 [A.1 GNU GENERAL PUBLIC LICENSE](#)），
“gmp.c”、“gmp.h”、“gtp.c”、“gtp.h”、文件集“interface/html/*”
和“win/makefile.win”除外。

文件“gtp.c”和“gtp.h”版权归属自由软件组织。为推广 GTP 协议这两个文件在较 GPL 更宽松的许可下，对非限制性的使用免费（参见 [A.3 The Go Text Protocol License](#)）。

文件“gmp.c”和“gmp.h”由其作者 William Shubert 发布到公共域上，对非限制性的使用免费。

文件“interface/html/*”不属于 GNU Go，而是一个单独的程序，为方便搜寻 GNU Go 的 CGI 接口而包含在发布文件中。它们已由其作者 Douglas Ridgway 发布到公共域上，对非限制性的使用免费。

文件 `"regression/games/golois/*.sgf"` 版权属于 Tristan Cazenave 授权包含。

`"regression/games/handtalk/"` 中的 SGF 文件版权属于 Jessie Annala 授权包含。

`"regression/games/mertin13x13/"` 中的 SGF 文件版权属于 Stefan Mertin 授权包含。

其余的 SGF 文件版权属于 FSF 或公共域。

1.3 作者

GNU Go 由 Daniel Bump、Gunnar Farneback 和 Arend Bayer 维护。

GNU Go 作者（按贡献顺序排列）是 Man Li、Wayne Iba、Daniel Bump、David Denholm、Gunnar Farneback、Nils Lohner、Jerome Dumonteil、Tommy Thorn、Nicklas Ekstrand、Inge Wallin、Thomas Traber、Douglas Ridgway、Teun Burgers、Tanguy Urvoy、Thien-Thi Nguyen、Heikki Levanto、Mark Vytlačil、Adriaan van Kessel、Wolfgang Manner、Jens Yllman、Don Dailey、Måns Ullerstam、Arend Bayer、Trevor Morris、Evan Berggren Daniel、Fernando Portela、Paul Pogonyshv、S.P. Lee、Stephane Nicolet 和 Martin Holters。

1.4 致谢

应当感谢 Arthur Britto、David Doshay、Tim Hunt、Matthias

Krings、Piotr Lakomy、Paul Leonard、Jean-Louis Martineau、Andreas Roever 和 Pierce Wetter 的相关支持。

感谢所有遇到 bug（并报告）的人！

感谢 Gary Boos、Peter Gucwa、Martijn van der Kooij、Michael Margolis、Trevor Morris、Måns Ullerstam、Don Wagner 和 Yin Zheng 在 Visual C++ 方面提供的帮助。

感谢 Alan Crossman、Stephan Somogyi、Pierce Wetter 和 Mathias Wagner 在 Macintosh 提供的帮助。感谢 Marco Scheurer 和 Shigeru Mabuchi 帮助发现了很多问题。

感谢 Jessie Annala 在手谈程序上的帮助。

特别感谢 Ebba Berggren 创建了标志，在 Tanguy Urvoy 的设计基础上并得到了 Alan Crossman 的建议。旧的 GNU Go 标志来自 Jamal Hannah 的字体，见：<http://www.gnu.org/graphics/atypinggnu.html>。这两个标志可以在“doc/newlogo.*”和“doc/oldlogo.*”找到。

还要感谢 Stuart Cracraft、Richard Stallman 和 Man Lung Li 使本程序成为 GNU 的一部分，William Shubert 写了 Cgoban 和 gmp.c，Rene Grothmann 写了 Jago，Erik van Riper 和合作者写了 NNGS。

1.5 GNU Go 任务表

你可以帮助使 GNU Go 成为最好的围棋程序。

这是一个为所有有兴趣帮助 GNU Go 的人列出的任务表。如果你想在这个项目上投入工作，应先联系并在这个特性如何工作达成共识！

版权说明：自由软件组织拥有 GNU Go 的版权，因此你的代码被 GNU Go

的官方发布中被接受之前，自由软件组织会要求你签署一个版权协议。

当然你也可以在不签署版权协议的情况下开发一个变体版本。你也可以依据 GPL 发布你的变体版本的源码（参见 [A.1 GNU GENERAL PUBLIC LICENSE](#)）。但如果你希望你的变动加入到发布的版本上，就需要签署版权协议。

要获取更多的信息和版权协议文本请联系 GNU Go 维护者 Daniel Bump（bump@sporadic.stanford.edu）和 Gunnar Farneback（gunnar@lysator.liu.se）。

以下是你可以参与的任务列表。其中的一些任务已经开始，但这并不妨碍你的加入。更多的讨论请与开发团队或已经参与任务的人联系。

1.5.1 总体

- 如果可能，报告发现的问题和修订。发现问题后，查明其原因并提出修改方法，发出一个补丁。如果你发现一个有趣的问题但不知道如何修补，也请告知。附带 `sgf` 文件（如果可能）和附带相关信息，如 GNU Go 版本号。在发生终止错误或段保护故障时可能需要知道你使用的操作系统和编译器以确定问题是否与平台相关。

1.5.2 小项目

这些项目是战术性的，即它们关注引擎的一些特性或基础结构。其中一些是相当小的，一个有经验的 GNU Go 编程者可能在一天内就可以完成。对于新的项目成员来说这也可能是很有用的开端。另一些较大并需要对引擎内部更深入的了解。这些问题按大致的复杂程度列出。

- 在 `patterns/mkpat.c` 中进一步增强主流程和强制流程的一致性。
- 从棋谱中分离并生成独立文件。这任务自 3.1.16 就已经开始了。在 `worms.c` 的其他地方已经使用了棋谱，存储提子、逃跑、叫吃和征

子。

- 实现着手列表象棋串那样存储棋块的关键着手和眼位。后手眼已经完成一半。着手已经保存但攻防代码尚未实现（LOSE、KO_A、KO_B 和 WIN）。
- 使 cache 在 64 位系统上不发生浪费内存。
- 实现在棋盘代码上监测 superko 失败。可能只需要在 play_move() 中引入 is_legal() 的一个变体中做选项。
- 棋块数据分为两个数组保存：dragon[] 和 dragon2[]。每块活棋在数组 dragon2 中只有一个入口，而数组 dragon 保存棋盘上每个点的数据。使用数组 dragon2 相同结构完成眼、一眼、死棋、活棋数据的转换。
- 在 eyes.db 和 optics.c 支持劫争。
- 用“clock.c”中的 autolevel 代码将时间操作集成到 play_gtp.c 中。或者，将它们用更好的代码替换。将其基于更坚固的系统识别理论和/或控制理论没有坏处。
- 写代码片断按定式库走棋，并检查引擎在数据库的所有位置都生成了定式手。这在“--nojosekidb”选项下运行会很有趣。

1.5.3 长期项目

这些项目是战略性的。它们可以帮助提高程序的对弈水平和/或加强特定的一些方面。

- 扩展回归测试包。

参见 doc 目录下的 texinfo 手册了解如何去做。测试包在解决普通的死活问题时特别有用。现在已经有效地覆盖了二线棋群、L 棋群和三角型，但举个例子说对于梳状形式、木匠块等等都没做任何事情。其他需要测试包的地方有

fuseki、tesuji 和 endgame。

- 调整模式数据库

这是经常性的修正。调整它们是一种艺术。大多数模式不需要辅助工具，所以不需要任何编程工作。希望再多几个上段棋手学会这项技术。

- 扩展和调整定式数据库

实现一个半自动做这项工作方法会非常有用。目前基于 `sgf` 文件的方式采用已有工具变得很困难。

- 对杀仍然需要改进（正在进行）。

- GNU Go 没有确切生成做模样或侵消/打入对方模样的着法。这种着法

生成可以使用在 `owl` 死活识别或连接识别第 5 点中相同的代码实现。

- 进一步改进的组合模块

现在的组合模块只识别提掉对手棋群的组合。更有用的组合例如识别叫吃棋块或打入对方地域。如果能识别战略着法和更间接攻击则会更加强大。可能可以用 AND-OR 树（DAG?）表示攻击组合，使其可以用二叉树搜索算法搜索（“`combination.c`”的修补正在进行）。

- 加速战术识别

GNU Go 在战术识别中已经相当准确，但并不总是很快。主要问题是试验了太多的无用手，和无需考虑的怪招。可以使着法生成的启发更为精炼来提升，也可以优化 `alpha-beta` 识别中的标准树搜索算法。

- 提高棋块安全识别的启发

这可能要考虑眼/后手眼、角上的模样、边上的模样、中间的模样、与临近活棋的接近程度、对方块棋等。能够准确的判定一手棋如何影响棋盘上所有棋群的安全特别有趣。

1.5.4 思想

有许多思想来自邮件列表。一些是现实的一些只是海市蜃楼。在此列出以

示鼓励。

- 一个好的 GUI

GoThic 已经做了一个开端，是基于 QT 工具包的 goban widget。这已经连接到了 GNU Go 在 gnu.org 上的开发网站。其他的还有基于 GTK 但仅是开始。

- 一个图形模式编辑器

这将使得非程序员可以很方便的提升 GNU Go 的实力。也可以被程序员用作除错工具。这个项目是以 GUI 为前提。这里的挑战是需要一个不仅可以方便地创建模式，还可以方便地浏览和维护数据库的工具。

- 保证引擎线程安全并在 SMP 计算机上使用多 CPU。
- 使引擎使用松散地连接在 internet 或群组的许多计算机。
- 在对手的用时中计算。
- 全局的 alpha-beta 识别。

这将会非常慢只能计算 2 到 3 步，但这可以发现致命的错误并提升 GNU Go 的着法生成。战略模块识别高级目标并转给引擎的其他部分。应当识别是否进入厚势、是否保守行棋或是否冒险（例如形势落后时）。也能够识别可攻击或防守的弱区域。这样的模块也许没有必要用 C 编写，也许 PROLOG、LISP 或其他 AI 语言更好。

- 一个使得 GNU Go 以不同风格行棋的参数

这样的风格可能是“取势”、“攻击”、“欺招”等。这就可以为人类对手提供不同的对手或告知 GNU Go 在比赛中如何对付特定的电脑对手。

- 形象化表示攻击使得可以有一个可以检索的攻击图形化表示，察看不同攻击的相互影响。
- 基于组合对局理论的终盘模块

要使其真正有用应当能够处理较早的终盘位置。

- 自动的定式调整

手工的定式调整非常困难。我应该鼓励有兴趣用 GNU Go 做机器学习试验的人们在定式上工作。这可能是最有可能获得重要价值和快速提高的领域。

- 分类更精确的劫争

创建各种劫争的样式并以代码实现操作。

2 安装

从 ftp.gnu.org 或其镜像站点上可以获取 GNU Go 的最新版本（参见 <http://www.gnu.org/order/ftp.html> 镜像列表）。可以在 <http://www.gnu.org/software/gnugo/> 阅读最新版本和其他信息。

2.1 GNU/Linux and Unix

2.2 配置选项

2.3 在微软平台下编译 GNU Go

2.4 Macintosh

2.1 GNU/Linux and Unix

Untar 源码，更换到 gnugo-3.6 目录。执行：

```
./configure [OPTIONS]
make
```

下一节将解释几个配置选项。除非你对 GNU Go 的性能不满意或想做一些试验，你不需要设置选项。

如：

```
./configure --enable-level=9 --enable-cosmic-gnugo
```

将生成的目标程序默认级别为 9，试验选项“cosmic”被使能。运行 `./configure --help` 可以取得选项列表。更多的信息在下一节介绍（参见 2.2 配置选项）。

运行配置和制作结束后，得到一个名为“interface/gnugo”文件，以 root 身份键入：

```
make install
```

将 gnugo 安装到“/usr/local/bin”。

使用 GNU Go 有多种方法。你可以从命令行键入：

gnugo

但更好的方式是在 X-Windows 下运行 CGoban 1，或在 Java 环境下运行 Jago 或使用其他提供图形界面的客户端程序。

你可以从 <http://sourceforge.net/projects/cgoban1/> 获取 Cgoban 1 的最新版本，早期的版本 1.12 可在 <http://www.igoweb.org/~wms/comp/cgoban/index.html> 获取。

Cgoban 版本号必须在 1.9.1 以上否则无法运行。CGoban 2 无法运行。

参见 3.2 通过 Cgoban 运行 GNU Go 获取如何在 Cgoban 下运行 GNU Go 的方法，或参见 3.3 其他客户端获取如何在 Jago 下运行 GNU Go 的方法。

2.2 配置选项

配置时，特别是对 GNU Go 的性能不满意的时候，可以考虑三个选项。

2.2.1 缓存

2.2.2 默认级别

2.2.3 DFA 配置选项

2.2.4 其他选项

2.2.1 缓存

GNU Go 默认在 RAM 中开辟 8M 缓存供内部使用，缓存用来存储局面分析时的中间结果。

增加缓存经常可在一定程度提高速度，如果你的系统内存充裕，可考虑增加缓存数量。但缓存太大的话就会发生页面交换，导致硬盘动作而降低性能。如果硬盘看起来超负荷工作，则标明缓存可能太大了。在 GNU/Linux 系统中。可以用程序 “top” 检测页面交换动作，用 “f” 命令切换交换显示。

在编译时运行以下命令之一即可重定义缓存：

```
./configure --enable-cache-size=n
```

将缓存设定为 n M，例如：

```
./configure --enable-cache-size=32
```

创建缓存为 32 M。如果省略则默认为 8 MB。重新配置 GNU Go 后必须用 `make` 和 `make install` 重新编译安装。

带选项“`--cache-size n`”运行 `gnugo` 可重定义编译时刻的默认值， n 是所需要的缓存数量，或在选项后表示所需级别。下章会详细介绍参数的设定。

2.2.2 默认级别

GNU Go 可以按不同级别对弈，最多可分 10 级。在 10 级时 GNU Go 更为精确但用时大约比 8 级平均多 1.6 倍。

可以在运行时刻用选项“`--level`”设定级别，如果没有设定，则采用默认级别。用配置选项“`--enable-level=n`”可设定默认级别。例如：

```
./configure --enable-level=9
```

将默认级别设定为 9。如果省略这个参数，编译器设定默认级别为 10。建议除非太慢都使用 10 级。如果决定需要修改默认值，可重新执行配置和编译。

2.2.3 DFA 配置选项

GNU Go 采用两种不同的方式实现模式匹配，DFA（Discrete Finite-state Automata，离散有限状态自动机）在 GNU Go 3.0 时为实验项目，现在已经成为标准。可以用配置选项 `./configure --disable-dfa` 禁止。该选项使得调试困难，但速度明显比原来的匹配器快（参见 10DFA 模式匹配）。

2.2.4 其他选项

引擎中所有新项目一般都以实验选项测试，在编译时刻或运行时刻可以打开或关闭。一些“实验”选项，如打入代码和对杀代码已经不再是实验性的了，默认是打开的。

除非对实验选项感兴趣，否则可跳过本节。

而且一些实验选项在稳定版中去掉了，例如 `owl` 扩展代码会导致崩溃，所以 3.6 中选项 `--enable-experimental-owl-ext` 被禁止了。

需要澄清“默认”一词，因为这里实际有两套默认参数：“`config.h`”定义的运行时刻参数和“`configure`”定义的编译时刻参数（编辑“`configure.in`”并运行 `autoconf` 创建）。例如在“`config.h`”中：

```
/* Center oriented influence. Disabled by default. */
#define COSMIC_GNUGO 0

/* Break-in module. Enabled by default. */
#define USE_BREAK_IN 1
```

这意味着默认禁止实验性的 `cosmic` 选项，该选项使 GNU Go 致力于在中心行棋（使得引擎变弱），但使用打入模块。这些参数在 GNU Go 不带参数运行时使用，可被运行时刻参数取代：

```
gnugo --cosmic-gnugo --without-break-in
```

也可以这样配置 GNU Go：

```
./configure --enable-cosmic-gnugo --disable-experimental-break-in
```

然后重新编译 GNU Go。这就改变了“`config.h`”的默认配置，在运行时刻就不必再向 GNU Go 传递运行时刻参数打开 `cosmic` 选项，禁止打入代码了。

如果想找出那些实验性选项编译进了 GNU Go 二进制代码中，可以运行 `gnugo --options`。以下是 GNU Go 实验性选项列表：

- `experimental-break-in`. 实验性的打入代码（参见 13.10 打



入)。打入代码在 10 级默认有效，9 级禁止，所以可能不需要配置，如果不需要打入代码就在 9 级对弈。

- cosmic-gnugo. 一种注重中央的实验性棋风，对标准的 GNU Go 胜率较高，但对其他对手变弱。
- large-scale. 试图大规模捕杀。原理参见 <http://lists.gnu.org/archive/html/gnugo-devel/2003-07/msg00209.html> 该选项会使引擎变慢。
- metamachine. 使能，运行复制一个新的 gnugo 进程作为“专家”的实验性选项。需要与运行时刻参数“-metamachine”同时使用。其他参数不是实验性的，可以像配置选项或运行时刻选项一样修改。
- chinese-rules 使用中国规则数子法
- resignation-allowed 允许 GNU Go 认输，默认打开。

2.3 在微软平台下编译 GNU Go


GNU Go 是在 Unix 变体下开发的，GNU Go 很容易在这些平台上构造和安装。GNU Go 3.6 支持在 MS-DOS、Windows 3.x、Windows NT/2000 和 Windows 95/98 构造。

在微软平台下构造 GNU Go 有两种方法：

  第一种方法是安装一个基于 GCC 在微软平台上移植版本的类 Unix 环境。这一方法被 GNU Go 开发者完全支持并工作得很好。针对微软平台有几个高质量的免费类 Unix 环境。

这种方法的一个好处是可以很容易地参与 Gnu Go 的开发。这些 unix 环境以“diff”和“patch”程序提供生成和应用补丁。

Unix 环境另一个好处是同样可以构件开发版本（这种版本可能比最新的稳定版本更强）。对 VC 的支持文件并不总能有效工作，而开发版本经常会失去同步，所以 VC 不能无错地构造。

 第二种方法是使用象 Visual C 这样专门面向微软平台的开发工具。GNU Go 2.6 和以后版本支持 Visual C。现在在发布中提供工程文件以支持 Visual C。

这节余下的提供在微软平台上编译 GNU go 的具体方法。

2.3.1 Win 95/98、MS-DOS 和 Win 3.x 下使用 DJGPP

2.3.2 Windows NT/95/98 下使用 Cygwin

2.3.3 Windows NT/95/98 下使用 MinGW32

2.3.4 Windows NT/95/98 使用 Visual C 和工程文件

2.3.5 Windows NT/95/98 下运行 GNU Go

2.3.1 Win 95/98、MS-DOS 和 Win 3.x 下使用 DJGPP

在这些平台上可以使用 DJGPP，GNU Go 安装在 Windows 95/98 长文件名的 DOS 窗口测试成功。GNU Go 可以使用 GCC 的 DJGPP 移植版本按标准 Unix 构造和安装程序编译。

一些 DJGPP 的 URL：

DJGPP 主页：<http://www.delorie.com/djgpp/>

DJGPP 在 simtel 上的 ftp：

<ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/>

<ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/>

一旦你有一个 DJGPP 工作环境，并下载了 gnugo 源码如

gnugo-3.6.tar.gz, 你可以按如下方式构造可执行文件:

```
tar zxvf gnugo-3.6.tar.gz
cd gnugo-3.6
./configure
make
```

作为选项你可以下载 DJGPP 的 glib 得到 snprintf 的工作版本。

2.3.2 Windows NT/95/98 下使用 Cygwin

在这些平台上可以安装 Cygwin 环境。最近的 Cygwin 版本带有安装程序, 安装非常容易。其主页 <http://sources.redhat.com/cygwin/>。

GNU Go 在 Cygwin 环境下用标准的 Unix 构造过程编译。安装 cygwin 并取得“gnugo-3.6.tar.gz”后, 键入:

```
tar zxvf gnugo-3.6.tar.gz
cd gnugo-3.6
./configure
make
```

生成的可执行文件不能独立运行: 它需要 cygwin1.dll 和 Cygwin 环境。cygwin1.dll 包含了 Unix 的仿真层。

Cygwin 主页: <http://sources.redhat.com/cygwin/>。

作为选项, 你可以使用 glib 获取 snprintf 的工作版本。Glib 不在 cygwin 内。

2.3.3 Windows NT/95/98 下使用 MinGW32

Cygwin 环境还带有 MinGW32。这生成一个可执行文件仅依赖于微软的 DLL。这可执行文件与 Visual C 的可执行文件相当, 比 Cygwin 可执行文件更容易发布。在 cygwin 上构造一个适应 win32 平台的可执行文件键入:

```
tar zxvf gnugo-3.6.tar.gz
```

```
cd gnugo-3.6
env CC="gcc -mno-cygwin" ./configure
make
```

生成的可执行代码用 cygwin 安装程序中的 upx 压缩可以大大减小尺寸。

2.3.4 Windows NT/95/98 使用 Visual C 和工程文件

假设你不想改变配置选项。如果需要则要编辑 "config.vc"。注意 configure 运行时, 该文件被 "config.vcin" 的内容覆盖, 因此你也可以编辑 "config.vcin", 以下指令并不需要运行 configure。

1. 打开 VC++ 6 workspace 文件 gnugo.dsw。
2. 将 gnugo 工程设为活动工程 (右击并选择 "Set as Active Project"), 在主菜单选择 "Build", 再选 "Build gnugo.exe", 即可构造所有运行时间子工程。

注意:

- a) 构造也可以从命令行启动:

```
msdev gnugo.dsw /make "gnugo - Win32 Release"
```

- b) 默认的配置为 "Debug", 要构造优化版本在主菜单上选择 "Build", 再选 "Set active Configuration" 并点击 "gnugo - Win32 Release"。参见 Visual Studio help 获得更多的项目配置信息。
- c) 在构造的第一个独立子工程 (utils) 中复制 config.vc 到根目录的 config.h。如果你想修改 config.h, 要把所有更改都写到 config.vc。特别地, 如果你想修改默认级别或默认的缓冲尺寸, 你必须编辑此文件, 具体的效果参见 2.1 GNU/Linux and Unix。
- d) 本工程用 VC 6.0 版本构造和测试。没有在 VC 的早期版本上经过测

试，几乎不能够工作。

2.3.5 Windows NT/95/98 下运行 GNU Go

GNU Go 不带自己的图形用户接口。可以使用 Java 客户端程序 jago。

运行 Jago 需要 Java 运行时环境 (JRE)。可以从 <http://www.javasoft.com/> 获取。这是 Java 开发工具包 (JDK) 中的运行时部分，包括 Java 虚拟机、Java 平台核类和支持文件。IE 5.0 随带的 Java 虚拟机也能工作。

Jago: <http://www.rene-grothmann.de/jago/>

1. 用 `gnugo --quiet --mode gmp` 启动 GNU Go
2. 从 cygwin 或 DOS 窗口运行 `gnugo --help` 获取选项列表

 作为选项定义 `--level <level>` 使对局更快

Jago 对于 Cygwin 和 MinGW32 执行文件都很快。DJGPP 执行文件也能工作，但同在对局结束和数子时同 jago 有些配合问题。

2.4 Macintosh

如果你有 Mac OS X，你可以用源自 GCC 的 Apple 编译器编译 GNU Go。

建议加上 `-no-cpp-precom` 的 CFLAGS 标志。

3 使用 GNU Go

3.1 获取文档

3.2 通过 Cgoban 运行 GNU Go

3.3 其他客户端

3.4 Ascii 接口

3.5 Emacs 的 GNU Go 模式

3.6 GMP 协议和 GTP 协议

3.7 电脑围棋锦标赛

3.8 SGF 格式

3.9 启动 GNU Go: 命令行格式

3.1 获取文档

你可以在“doc/”目录下运行 `make gnugo.ps` 并打印出生成的操作手册打印件。该手册包含了 GNU Go 大量的算法信息。

在其他支持 `info` 文档的平台，你可以（以根身份）从“doc/”目录执行“`make install`”安装操作手册。`Info` 文档可以方便地在 Emacs 中执行指令 `Control-h i` 阅读。

“doc/”中的文档包括一个 `man` 页“`gnugo.6`”、`info` 文件“`gnugo.info`”、“`gnugo.info-1`”，等以及可生成 `info` 文件的 `Texinfo` 文件。`Texinfo` 文件包含本用户指南和向开发者提供的 GNU Go 算法的扩展信息。

如果你需要一份 Texinfo 打印件，可以在“doc/”目录中 `make gnugo.dvi` 或 `make gnugo.ps`。（`make gnugo.pdf` 仅当用 `epstopdf` 之类程序将 doc/下所有 .eps 文件转换为 .pdf 文件后才可用）。

你可以用指令制作 `makeinfo --html gnugo.texi` 一个 HTML 版本，可用 `texi2html -split_chapter gnugo.texi` 生成更好的 HTML 文档。你可以从 <http://www.mathematik.uni-kl.de/~obachman/Texi2html/> 获取 `texi2html` 工具（版本 1.61 或更新）（参见 <http://texinfo.org/texi2html/>。）

可从任何终端或上执行 `gnugo --help` 或 `man gnugo`，或从 Texinfo 文档中获取用户文档。

开发者文档为 Texinfo 文档，在源码中注解。如果你有兴趣在开发本程序中提供帮助请联系 gnugo@gnu.org。

3.2 通过 Cgoban 运行 GNU Go

William Shubert 写了两个不同的程序，都叫做 CGoban，本文档中 CGoban 指旧版本 CGoban 1.x，现在有版本 1.12 或更高的。

CGoban 是一个特别好的运行 GNU Go 的方法。Cgoban 在 X-Windows 下提供良好的图形界面。

启动 Cgoban。当 Cgoban 控制面板出现后，选择“Go Modem”，你可以得到 Go Modem 安装，选择对弈一方(或双方)为“Program”，在框中填入 gnugo 的路径。点击 OK 后得到对局设置窗口，选择“Rules Set”为 Japanese（否则让子无法工作），按你需要设置棋盘尺寸和让子。

如果你想下贴子棋，你需要忍受的是 GMP 不提供贴子的交换。由于这个缺

点，除非你在命令行上设置了贴子，否则 GNU Go 只能去猜。分先贴子是 5.5，让子棋是 0.5。如果这不是你需要的，你可以在 Go Modem 协议设置窗口的命令行上加上“--komi”选项，你只能在下次对局设置窗口出现时重新设置贴子。

准备好对弈后点击 OK。

在 Go Modem 协议设置窗口，当你设置 GNU Go 的路径时，可以给出命令行选项，如“--quiet”屏蔽大多数信息。由于占用了标准 I/O，其他信息都送到了 stderr，无论它们是否是真的错误信息。这会在你启动 Cgoban 的窗口上显示。

3.3 其他客户端

除 CGoban（参见 3.2 通过 Cgoban 运行 GNU Go）外，还有许多可以运行 GNU Go 的客户端程序。这里列出已知的自由软件。另外在 http://www.gnu.org/software/gnugo/free_go_software.html 中还维护了一个围棋程序的大列表。

- qGo (<http://sourceforge.net/projects/qgo/>) 是一个全功能的客户端程序、支持服务器端对弈、SGF 查看/编辑和 GNU Go 客户端，采用 C++ 编程，运行环境 GNU/Linux、Windows 和 Mac OS X。Can play One Color Go. Licensed GPL and QPL.
- glGo (<http://ggo.sourceforge.net/>) Peter Stempel 编写的 C++ 客户端程序，可以与 GNU Go 对弈或在 IGS 上对弈。在 GPL 下提供源码。
- ccGo (<http://ccd.w.org/~cjj/prog/ccgo/>) 是 GPL 下客户端，用 C++ 编制，可以与 GNU Go 对弈或在 IGS 上对弈。

- RubyGo (<http://rubygo.rubyforge.org/>) 是 GPL 下客户端，由 J.-F. Menon 用脚本语言 Ruby 为 IGS 编写。RubyGo 可以使用 GTP 与 GNU 对弈。
- Dingoui (<http://dingoui.sourceforge.net/>) 是一个自由的 GTK 下 GMP 客户端，可以运行 GNU Go。
- Jago (<http://www.rene-grothmann.de/jago/>) 是 GPL 下 Java 客户端，在 Microsoft Windows 和 X-Window 下都能运行。
- Sente Software 的 FreeGoban (<http://www.sente.ch/software/goban/freegoban.html>) 是颇受欢迎的 GNU Go (可能还有其他程序) 的用户接口。GPL 许可。
- Mac GNU Go (<http://www1.u-netsurf.ne.jp/~future/HTML/macgnugo.htm>) 是一个 GNU Go 3.2 的前端，有英文版和日文版。GPL 许可。
- Quickiego (<http://www.geocities.com/secretmojo/QuickieGo/>) 是一个 GNU Go 2.6 的 Mac 接口。
- Markus Enzenberger 编写的 Gogui (<http://sourceforge.net/projects/gogui/>) 是一个 Java 评估器，支持 gtp (<http://www.lysator.liu.se/~gunnar/gtp>) 引擎如 GNU Go。源码在 CVS (http://sourceforge.net/cvs/?group_id=59117)。GPL 许可。Gogui 不支持 gmp 或服务器端对弈，但可能对 GNU Go 或其他引

擎的程序员很有用。

- gGo 是一个 Java 程序最初名为 as qGo for Java。尽管程序的公开源码不再维护了，但仍很有用，可在 sourceforge (<http://prdownloads.sourceforge.net/ggo/>) 或 (<ftp://download.sourceforge.net/pub/sourceforge/g/g/g/ggo/>) 找到。GGo 可作为客户端或 sgf 编辑器，支持 GTP，可以进行服务器对弈也可与 GNU Go 对弈。GPL 许可。
- Quarry (<http://home.gna.org/quarry/>) 是支持 GTP 的 GPL 客户端。运行于 GNU/Linux 下，需要 GTK+ 2.x 和 librsvg 2.5 支持。支持 GNU Go 和其他引擎。不仅可以下围棋，还可以下其他一些棋盘游戏。
- Gobon (<http://www.waz.easynet.co.uk/software.html>) 是一个使用 Wayne Myers 的 GTK 库的 GTP 前端。GPL 许可。

3.4 Ascii 接口

即使你没有安装 Cgoban 你也可以以其默认的 Ascii 接口启动 GNU Go。简单在命令行输入 gnugo，GNU Go 会画出一个棋盘。键入 help 可以得到一个选项列表。对局结束两次 pass，GNU Go 就会提示你数子。你和 GNU Go 必须对死子达成一致。——你可以同意更替要取走的死子状态，结束后 GNU Go 会报告结果。

你可以在对局的任何时候使用 save filename 指令保存棋局。你可以用指令 gnugo -l filename --mode ascii 从该指令生成的 SGF 文件重装棋局。在使用 Cgoban 对局中不允许重装棋局。你可以用 Cgoban 存储一个文

件，再在 `ascii` 方式下重装。

3.5 Emacs 的 GNU Go 模式

GNU Go 可以在 Emacs 中运行，并且可以用光标键或鼠标落子。Emacs 版本 21 以上支持图形化棋盘显示。

用 `M-x load-file` 命令装入 `"interface/gnugo.el"`，如果使用图形化棋盘再装入 `"interface/gnugo-xpms.el"`。

细节：Emacs 使用控制键和换档键，换档键在 PC 键盘上是 `Alt` 键。控制键和换档键分别用 `"C-"` 和 `"M-"` 表示，`M-x` 表示按住 `Alt` 键再按 `x`，在 Emacs 中这提示输入下调命令。输入 `load-file` 并回车，接着输入到达 `"interface/gnugo.el"` 的路径，再次按回车并重复该过程装入 `"interface/gnugo-xpms.el"`

如果想要 `"interface/gnugo.el"` 和 `"interface/gnugo-xpms.el"` 这两个在运行 Emacs 是自动装入，可将它们复制到 `"site-lisp"` 目录（一般是 `"/usr/share/emacs/site-lisp"`）并在 `".emacs"` 文件中加入以下两行：

```
(autoload "gnugo" "gnugo" "GNU Go" t)
(autoload "gnugo-xpms" "gnugo-xpms" "GNU Go" t)
```

现在可以用 `M-x gnugo` 启动 GNU Go，获取命令行提示可参见 3.9 启动 GNU Go：命令行格式。用这些你可以设置让子、棋盘尺寸、颜色和贴子。例如要执白授九子可使用选项：`"--handicap 9 --color white"`。

你立即看到的是 ASCII 界面，但你可以输入 `"i"` 切换到图形界面，（除非你使用的是另一个版本 `"gnugo.el-ascii"` 或者 Emacs 版本不是 21 以上。任何时候都可以输入 `"?"` 获取帮助。

可以用光标键移动到所需位置，按空格键落子，也可以用鼠标点击落子。
可以存取棋局、悔棋。任何时候都可以按下“!”获取估计的目数。

可以向引擎直接输入 GTP 命令。

尽管不显示网格，但可以键入 `showboard RET` 键入显示棋盘局面，这实际上是执行了 GTP，绘制 ASCII 棋盘图形。

“gnugo.el”正在开发中，更新的版本会运行更良好，功能更全，随 GNU GO 3.6 发布的是“gnugo.el-2.2.8”，可以在以下地址或 GNU GO CVS 找到更新的版本：
<http://www.glug.org/people/ttn/software/ttn-pers-elisp/standalone/>

以下是一些默认的键盘定义：

- “?”获取帮助。

- “Return”或“Space”

选择下一手落子。图形模式下也可以用鼠标。

- “q” or “Q”

退出。（以后会加确认）

- “R”

认输。

- “C-l”

刷新。包括重装默认窗口配置。

- “u”

回退两手棋（你和对方）。

- “U”

回退到鼠标光标位置。如果这是刚下的最后一手（因为计算机回应之前你无法输入命令），默认是回退两手。也可以将光标移动到要回退的位置按“U”，或者用前缀“C-”输入要回退的棋子序号。接着所有这手以后的着手都撤销。

- “C-1”

重绘棋盘。

- “_”或“M-_”

隐藏棋盘和控制台（老板键）。

- “p”

空手，即放弃本手落子权。

- “i”

切换到 XPM 显示（如果支持）。

- “w”

运行“gnugo-worm-stones”

- “d”

运行“gnugo-dragon-stones”

- “W”

运行“gnugo-worm-data”

- “D”

运行“gnugo-dragon-data”

- “t”

运行“gnugo-toggle-dead-group”

- “!”

获取 GNU Go 评分

- “: ” 或 “; ”

输入 GTP 引擎命令。

- “=”

显示光标处棋盘局面。

- “h”

在回显区显示“(N moves)”后跟着手历史，最近着手。在*Messages*

缓存。

- “F”

运行“gnugo-display-final-score”。

- “s” or C-x C-w or C-x C-s

运行 “gnugo-write-sgf-file”

- “l”

运行 “gnugo-read-sgf-file”

3.6 GMP 协议和 GTP 协议

GMP 协议由 Bruce Wilcox 根据 David Fotland、Anders Kierulf 和其他一些人的输入开发，其历史见 <http://www.britgo.org/tech/gmp.html>。

该协议是一个所有围棋程序都应支持的标准。由于 Cgoban 支持该协议，所有围棋程序的用户接口都可通过 Cgoban 实现。程序员只需要关注实际的而无需考虑绘制棋子、缩放棋盘和其他无关的事情。

GNU Go 3.0 引入了一个新协议，称 GTP 协议（参见 19GTP 协议），希望可以实现目前 GMP 完成的功能。

3.7 电脑围棋锦标赛

电脑比赛目前使用 GMP 协议。目前在这些比赛中的方法是将两台计算机的串行通信端口以“null modem”电缆连接。如果你运行 GNU/Linux 很方便使用 Cgoban。如果你的程序执黑，在 Go Modem 协议设置窗口中按通常设置。如果执白，选择“Device”，如果你的串行通信端口为 COM1，将设备设置为“/dev/cua0”。如果是 COM2 设置为“/dev/cua1”。

3.8 SGF 格式

SGF 格式，是存储围棋棋局的标准格式。GNU Go 支持 SGF 文件的读写。

SGF 规范 (FF[4]) 在 <http://www.red-bean.com/sgf/>。

3.9 启动 GNU Go：命令行格式

3.9.1 一些基本选项

- `--help`, `-h`

打印描述选项的帮助信息。可以告诉你各种参数的默认值、最重要的级别和缓冲尺寸。默认值可在编译前用 `configure` 设定。如果忘记可用 `--help` 查出。

- `--boardsize size`

设置棋盘尺寸

- `--komi num`

设置贴子

- `--level level`

GNU Go 可以在不同级别和速度下对弈。10 级是默认级别。降低级别可使 GNU Go 更快但识别粗糙。

- `--quiet`, `--silent`

不打印版权和其他信息，但不丢弃其他命令行参数指定参数，如 `--trace`。

- `-l`, `--infile filename`

装入指定的 SGF 文件。GNU Go 给出下一手。如果你要取代它并为另一方下一手，可加选项 `--color <color>`，`<color>` 为 `black` 或 `white`。

- `-L`, `--until move`

到指定手后停止装入。`move` 可以是手数或位置。

- `"-o", "--outfile filename"`

写 `sgf` 输出文件。

- `"-O", "--output-flags flags"`

在 `sgf` 文件中加入有用信息。标志可以是 `"d"`、`"v"` 或组合（即 `"dv"`）。

如果选 `"d"` 在 `sgf` 文件中标注死棋或危险棋块。如果选 `"v"` 注明着手的棋盘效力。

- `"--mode mode"`

强制对弈模式（`"ascii"`、`"emacs"`、`"gmp"` 或 `"gtp"`）。默认的为 ASCII，如果没有监测到终端，就选择 GMP（Go Modem 协议），实际上这就是你一般需要的，可能不需要该选项。

- `"--resign-allowed"`

如果该选项使能 GNU Go 会认输。不幸的是 Go Modem 协议无法传输认输操作，所以该选项在 GMP 模式下无效。

- `"--never-resign"`

GNU Go 不会认输。除非你在构造引擎时配置了选项 `-enable-resignation-allowed` 这是默认的。

3.9.2 其他普通选项

- `"-M", "--cache-size megs"`

以 MB 表示的识别所使用的内存。默认为 8 除非你在编译前用指令 `configure --enable-cache-size=size` 配置过 gnugo 的默认尺寸（参见 2 安装）。GNU Go 在一个 Hash 表中存储识别计算结果（参见 11.2 局面 Hash）。Hash 表满，就清空并继续识别，如果缓冲不是足够大，就要发生交换，有些以前做过的识别有的不得不重复执行，所以需要有更大的缓冲使得 GNU Go 运行得更快。在 GNU/Linux 机器上使用 `top` 程序可以发现交换。当

然，如果你有足够的内存或者性能成了问题你可能需要用此选项修改缓冲尺寸。

- `--chinese-rules`

使用中国规则。即数子法。这在分先棋中可能会差一子，让子棋或双方都放弃时可能差更多（因为在中国规则中让子属于黑方）。

- `--japanese-rules`

使用日本规则。除非你指定`--enable-chinese-rules`为配置选项，这是默认选项。

- `--copyright`：显示版权信息。

- `--version` or `-v`：打印版本号

- `--printsgf filename`：

创建包括棋盘图形的 SGF 文件。与`-l`和`-L`结合从另一个 sgf 创建棋盘图形非常有用。禁手用属性 IL 表示。该属性在 FF4 SGF 规范中未定义，可以使用。该特性在 CGI 接口`interface/html/gg.cgi`使用。

- `--options`

打印编译进程序的试验配置（参见 2.2.4 其他选项）。

- `--orientation n`

与`-l`组合使用，棋盘可按反射和旋转和变换出 8 个不同方式，这个选项取其中一种（默认为 0）。参数`n`为 0 到 7。

3.9.3 其他影响实力和速度的选项

- `--level amount`

级别越高，GNU Go 算路更深。默认为 10 级。如果 GNU Go 在你的机器上运行过慢，你可能需要降低级别。

这个参数`--level`是选择对弈实力强弱最好的办法，它控制了一组其他

一些可以在命令行单独设定的参数。这些参数的默认值可运行 `gnugo --help` 获得。

除非你在程序上做工作，否则你不需要这些选项，只要修改一个变量 `--level`。其他选项是由开发者用来调节程序性能和精确度的。为完整起见，在此列出。

- `"-D", "--depth depth"`

深度搜索修剪。当搜索超过这个深度（默认 10）时，GNU Go 假设任何棋串有 3 气的即是活棋。GNU Go 可以识别任意深度的征子，但其他吃子可能会忽略。

- `"-B", "--backfill-depth depth"`

深度搜索修剪。超过这个深度（默认为 12）时，GNU Go 不考虑回填的招法。

- `"--backfill2-depth depth"`

其它控制 GNU Go 搜索回填招法的深度。在 `backfill2_depth` 以下的搜索较 `backfill_depth` 以下更模糊一些并且对时间敏感的，故此参数值稍低。

- `"-F", "--fourlib-depth depth"`

深度搜索修剪。当搜索超过这个深度（默认为 7）时 GNU Go 假设所有棋串有 4 气为活棋。

- `"-K", "--ko-depth depth"`

深度搜索修剪。超过这个深度（默认为 8）是，GNU Go 不再非常关注分析打劫。

- `"--branch-depth depth"`

设定 `branch_depth`，一般略低于 `depth`，在 `branch_depth` 和 `depth` 之间可以考虑 3 气棋串的攻防，但不考虑分支情况，所以不需要考虑很

多变招。在这个深度以下（默认为 13），GNU Go 仍考虑对 3 气气串的攻击，但只考虑一手棋。

- `--break_chain-cutoff depth`

设定 `break_chain_depth`，在这个深度以下，GNU Go 禁止一些放弃一些试图吃子的防守招法。

- `--aa-depth depth`

识别函数 `atari_atari` 从一系列叫吃开始搜索组合，寻找可以使得最终状态出乎预期的变化（如死棋或危气成了活棋）。这个命令行选项设置参数 `aa_depth`，确定这个搜索组合的函数运行的深度。

- `--superstring-depth`

超串（参见 11.8 超串）是由紧密连接的棋串组成的。有时最好的攻防是对超串中的某一棋串进行攻防。这样的战术在 `superdragon_depth` 以下深度执行，命令行选项允许修改。

以下选项在搜索代码中已经写入文档（参见 11.1 识别基础）。

- `--owl-branch` 在此深度下 Owl 只考虑一手，默认为 8。
- `--owl-reading` 在此深度下 Owl 假定大龙逃出，默认为 20。
- `--owl-node-limit`

如果变化数目超过这个限制，Owl 假定大龙可以做活，默认 1000。提醒用户注意，增加 `owl_node_limit` 并不一定增强程序的实力。

- `--owl-node-limit n`

如果变化数目超过这个限制，Owl 假定大龙可以做活，默认 1000。提醒用户注意，增加 `owl_node_limit` 并不一定增强程序的实力。

- `--owl-distrust n`

在此限制以下一些 owl 搜索别修剪掉。

3.9.4 Ascii 模式选项

- `--color color`

选择你的颜色 (`"black"` or `"white"`)。

- `--handicap number`

选择让子数目 (0--9)。

3.9.5 开发选项

- `--replay color`

为一方或双方复盘。与 `"-o"` 选项同时使用，棋局同时记录着手价值。此选项需要 `"-l filename"`。color 可以是：

- white: 为白方复盘。
- black: 为黑方复盘。
- both: 为双方复盘。

当 `genmove` 生成的着手与 `sgf` 文件不同时，两个着手按以下形式报告：

```
Move 13 (white): GNU Go plays C6 (20.60) - Game move F4 (20.60)
```

对一个例如提高速度或者其他优化修改，这个选项可以确认是否影响了引擎的表现。注意如果几个着手最大价值相同或接近时，着手选择是不确定的（尽管可能由于使用了同一个随机数种子而结果相同），所以 `sgf` 文件中可能会有几个变化。只有报告的价值不同才能认定引擎表现与 `sgf` 文件不一致。参见 20 回归测试。

- `"-a", "--allpats"`

测试所有模式，即使其价值小于已发现的着手。这不会影响 GNU Go 的最终着手，并使得运行更慢。但这在调整 GNU Go 时非常有用。这使得跟踪和输出文

件 (“-o”) 信息更丰富。 控制变量 allpats。

- “-T”, “--printboard”: 棋块彩色显示。

使用 rxvt、 xterm 或 Linux Console (参见 5.8 彩色显示)。

与“-E”互斥, 控制变量为 printboard。

- “--showtime”

向 stderr 打印时间信息。

- “-E”, “--printeyes”: 眼位彩色显示。

使用 rxvt、 xterm 或 Linux Console (参见 5.8 彩色显示)。

与“-T”互斥, 控制变量为 printboard。

- “-d”, “--debug level”

生成调试输出。调试级别由 16 进制给出, 使用位表示“engine/gnugo.h”中的定义。列表可以使用“--debug-flags”获取。如下所列。:

DEBUG_INFLUENCE	0x0001
DEBUG_EYES	0x0002
DEBUG_OWL	0x0004
DEBUG_ESCAPE	0x0008
DEBUG_MATCHER	0x0010
DEBUG_DRAGONS	0x0020
DEBUG_SEMEAI	0x0040
DEBUG_LOADSGF	0x0080
DEBUG_HELPER	0x0100
DEBUG_READING	0x0200
DEBUG_WORMS	0x0400
DEBUG_MOVE_REASONS	0x0800
DEBUG_OWL_PERFORMANCE	0x1000
DEBUG_LIFE	0x2000
DEBUG_FILLLIB	0x4000
DEBUG_READING_PERFORMANCE	0x8000
DEBUG_SCORING	0x010000
DEBUG_AFTERMATH	0x020000
DEBUG_ATARI_ATARI	0x040000

DEBUG_READING_CACHE	0x080000
DEBUG_TERRITORY	0x100000
DEBUG_OWL_PERSISTENT_CACHE	0x200000
DEBUG_TOP_MOVES	0x400000
DEBUG_MISCELLANEOUS	0x800000
DEBUG_ORACLE_STREAM	0x1000000

这些调试标志可叠加。如果你想同时调试大龙和块棋，可以使用 “-d0x420” 。

- “--debug-flags”

打印调试标志列表。

- “-H”, “--hash level”

hash (参见 “engine/gnugo.h” 的位表示)。

- “-w”, “--worms”

打印块棋的更多信息。.

- “-m”, “--moyo level”

moyo 调试，显示模样盘。 level 另外有详细定义(参见 0 使用影响代码棋盘上的空区域按三种方式分区。一点可分为白或黑的实地、模样和区域。函数 whose_territory()、whose_moyo() 和 whose_area() 可返回按该类归属的一方或空。

- 实地

棋盘上预期终局时成为一方或另一方实际点数的部分视为实地 。

- 模样

棋盘上已成实地或如果对方不做侵消容易成实地的部份视为模样。

- 区域

棋盘上一方影响强于另一方的部份视为区域。

一般地实地是模样而模样是区域。为取得这些概念的感性认识，可用选项 “-m 0x0180” 装入一个中盘的 sgf 文件并检查其结果图形 (参见 13.13 Colored display and debugging of influence)。

引擎中采用影响函数的地方)。

`"-b", "--benchmark number"`

排名测试模式，可与 `"-l"` 同时使用，使 GNU Go to 自行反复对弈，采用一些随机的着手开始。此模式在 `twogtp` 程序中被大量改进。

- `"-S", "--statistics"`

打印统计信息（调试用）。控制变量 `showstatistics`。

- `"-t", "--trace"`

打印调试信息，使用两次得到更多信息。控制变量为 `verbose`。

- `"-r", "--seed seed"`

设置随机数种子。这可以保证 GNU Go 在同一局面中生成同样的着手。如果 `seed` 为 0，GNU Go 每次会生成不同着手。

- `"--decide-string location"`

启动战术识别代码（参见 11 战术识别）确定位置 `location` 上的棋串是否能提掉，如果是如何防守。如果和 `"-o"` 一起使用可以生成一个 SGF 文件记录变化。

- `"--decide-owl location"`

启动 `owl` 代码（参见 12.1 Owl 代码）以确定 `location` 位置上的模块是否能吃掉，如果是如何防守。如果和 `"-o"` 一起使用可以生成一个 SGF 文件记录变化。

- `"--decide-connection location1/location2"`

确定 `location1` 和 `location2` 位置上的模块是否可以联络。与 `"-o"` 同时使用非常有用，可将变化写入 SGF 文件。

- `"--decide-dragon-data location"`

打印 `location` 处模块状态的完整信息。

- `"--decide-semeai location1/location2"`

`location1` 和 `location2` 上分别是相邻的双方模块，各自都没有活，

它们的结局（活、死活双活）要靠对杀决定，确定其结果。带 “-o” 参数输出变化到一个 SGF 文件很有用。

- “--decide-tactical-semeai location1/location2”

与“--decide-semeai”类似，但不考虑由 owl 代码推荐的着手。

- “--decide-position”

尝试攻防所有 `dragon.escape<6` 的棋块。带 “-o” 参数输出变化到一个 SGF 文件。

- “--decide-eye location”

评估 location 处的眼位并打印报告。命令行增加“-d0x02”可获取更多信息（参见 8.7 局部博弈值）。

- “--decide-surrounded location”

棋块可能被对手棋块包围（surrounded）但并不一定无法逃跑，但受到攻击时最好有一个提示。此选项绘出对手棋块的势力轮廓，并确定 location 位置上的棋块是否被包围。

- “--decide-combination”

调用函数 `atari_atari` 确定在棋盘上是否存在组合。

- “--score method”

需要 “-l” 确定什么棋局需要数子，以及“-L” 如果你想在棋局任何时候都需要数子而不是终局数子。method 可以是 “estimate”、“finish” 或 “aftermath”。“finish” 和 “aftermath” 在棋局结束或差不多结束时适用，都试图提供一个精确的结果。注意如果棋局没有结束，其运行需要比已经结束的棋局多许多的时间得出结果。“estimate” 模式可以用于在中盘时做一个快速的估计。所有这些选项都可以与“--chinese-rules”同时使用，如果你想用中国（数子法）数子规则。

如果给出 `"-o outputfilename"` 选项，结果同样可以作为备注写入输出文件。对于 `"finish"` 和 `"aftermath"` 数子算法，自动完成的着手也计算在内。

- `estimate`

检查所有棋盘上棋群的状态，使用 Bouzy 5/21 算法给出一个快速评分（参见 14 模样的另一方案：Bouzy 的 5/21 算法）。

- `finish`

自行终局，确定所有棋子状态并使用 Bouzy 5/21 算法计算地域（参见 14 模样的另一方案：Bouzy 的 5/21 算法）。

- `aftermath`

自行终局，采用 `"aftermath"` 精确确定所有棋子状态，即使得所有棋子都为无条件活或无条件死。较 `"--score finish"` 慢，这些算法基本一致，如果有差别，`"--score aftermath"` 更接近正确。

- `--score aftermath --提掉-all-dead --chinese-rules`

此组合启动 Tromp-Taylor 评分。Tromp-Taylor 规则要求棋局终局且将所有死子取走，然后使用数子法（中国规则）评分。选项 `"--提掉-all-dead"` 要求 `aftermath` 代码完成提取死子。.

3.9.6 试验选项

这里的大多数都是配置选项，并在 2.2.4 其他选项描述。

- `"--options"`

打印已编译到程序的试验配置参数。

- `"--with-break-in"`

- `"--without-break-in"`

使用或不使用试验打入代码。此选项在 9 及以下级别没有效果。插入代码在 10 级默认使用，9 级和 10 级的唯一区别就是打入代码在 9 级中被禁止。.

- `--cosmic-gnugo`

使用基于中央的影响。

- `--nofusekidb`

关闭定式数据库。

- `--nofuseki`

完全关闭定式。

- `--nojosekidb`

关闭 joseki 数据库。

- `--mirror`

模仿棋。

- `--mirror-limit n`

n 手后停止模仿棋。

4 GNU Go 引擎纵览

本章是 GNU Go 内核的整体介绍。关于某个模块或者例程的详细信息可在以后的章节或源码文件的注释中找到。

GNU Go 的第一阶段是尝试尽可能读懂当前的棋盘局面，利用在这第一阶段获取的信息，加上着手生成器可以生成一个备选着手列表；最后对每个备选着手根据实地价值（包括吃子或死活影响）和可能的战略影响（如加强弱棋）估值。

注意 GNU Go 事实上做了大量的局部识别，分析可能的吃子、棋群死活等计算，但（至今）没有做全局判断。

4.1 获取信息

4.2 着手生成

4.3 着手评估

4.4 详细的事件序列

4.5 路线图

4.6 代码风格和体例

4.7 源码浏览

4.1 获取信息

这是着手生成中最重要的一个阶段。死活状态识别错误会导致大量的错误；势力估计的错误会导致不精确的着手估值；对弱棋群的识别错误可能导致战略错误。

函数 `examine_position()` 完成局面信息的获取，它首先调用 `make_worms()`。

开始几步很简单：`build_worms()` 识别直接连接的棋子，称棋串

(worms)，并得到其子数和气数。

然后就是最重要的棋串分析：为每个棋串调用战术识别代码（参见 11 战术识别），识别每个棋串是否可被直接提掉，如果一个棋串大于 5 气立即放弃。如果一个棋串可被提掉，引擎当然要搜索防守着手。另外，要做很大的努力找到提掉棋串或者防守的全部招法。

确定了哪些棋串是战术稳定的以后，调用 13 影响函数中的代码制作一个盘上的势力平衡图，该部分代码以后还将调用到。

在此基础上进行棋块分析，棋块 (dragons) 指一群无法被分断的棋子。

相应的函数 `make_dragons()` 第一步自然是识别这些棋块，也即确定哪些棋串是可以被分断的。其中部分工作是采用模式识别完成的，但大多数情况下要调用专门的联络识别代码，这个代码采用最小-最大搜索确定两个给定的棋串是否可联络或分断。

接着依据不同的准则计算确定棋块的强弱：

- 粗略的眼数估计
- 使用影响函数计算结果识别棋块是否与实地或模样连接
- 对受到攻击后逃跑的可能性的估计

对于认定较弱的棋块进行死活分析（参见 12.1Owl 代码）。如果两个相邻棋块被发现都没有活，就试图用对杀模块解决这类棋盘局面。

关于更详细的棋串和棋块分析（以及存储这些信息的数据结构），参见 7 棋串和棋块。

接着第二次调用影响函数代码对可能的实地进行详细的分析。当然此时也考虑了棋块的死活状态。

影响模块得出的势力范围结果经过打入模块的修正，它主要试图分析对手可能会打入哪里，哪个势力范围，使得影响代码难以识别。

4.2 着手生成

确定了所有的棋盘局面后，可以生成最佳着手。着手由一些成为着手生成器的模块生成。着手生成器自己不确定着手的估值，只说明其理由，称着手目标。所有着手和目标都生成好之后，最后就是着手评估。

关于 GNU Go 着手列表和目标，以及它们如何评估，参见 6 着手生成。

有一些着手生成器只是摘取了前面局面检查阶段得到的数据结果：

- `worm_reasons()`

以棋串攻防为目标的着手

- `owl_reasons()`

检查由前一阶段 owl 代码 (see section [12.1 The Owl Code](#)) 确定的棋块状态，选择其中关键性的攻防为目标

- `semeai_move_reasons()`

与 `owl_reasons` 类似，此函数生成与对杀相关的着手

- `break_in_move_reasons()`

打入模块建议的打入对手势力的着手

以下着手生成器则有额外的工作：

- `fuseki()`

在空角上按照数据库 `fuseki` 生成定式着法的前几步

- `shapes()`

这可能是最重要的着法生成器。从 “patterns/patterns.db”、
“patterns/patterns2.db”、 “patterns/fuseki.db” 和定式文件中
取得当前棋盘局面模式。所有模式都经过 8 种可能的方位变换，包括旋转和镜像匹配。如果匹配成功，则进一步确定该模式是否可以在这个局面，称“限制性”测试，这里的限制条件可能包括棋串的气数、死活状态和征子关系等。模式
可以调用手写的 (在 `patterns/helpers.c`) 或者自动生成的辅助函数。

模式可以是不同的目标和不同的类型。例如有棋群攻防的模式、联络或切断的模式和简单整形的模式(除由 `shapes()` 调用的大模式数据库外, 模式匹配还被其他模块用于程序中的其他许多任务, 参见 9 模式代码的详细文档)。

- `combinations()`

检查是否有攻击或叫吃的次序, 并策划攻击或防守。

- `revise_thrashing_dragon()`

此模块不直接生成着手: 如果事先清楚, 对手最后一个着手是死棋的一部分, 但出于安全考虑我们仍继续攻击它。这可以通过将 `thrashing draoon` 状态设定为 `unkown` 以再次进行棋形着手生成并估值来完成。

- `endgame_shapes()`

如果没有发现估值大于 6.0 的着手, 这个模块匹配一套为终局设计的特殊模式。终局模式在“`patterns/endgame.db`”。

- `revise_semeai()`

如果没有找到任何着手, 此模块将对杀局面中对手棋群的状态由 `DEAD` 改为 `UNKNOWN`, 然后再次运行 `shapes` 和 `endgame_shapes` 检查是否产生新着手。

- `fill_liberty()`

填公气。这只在终局中使用。如果需要, 生成回填或提吃的着手。

4.3 着手评估

着手生成模块运行后, 每个备选着手通过 `review_move_reasons` 做一个细致的评估, 同时也启动了一些分析试图发现被忽略的其他着手目标。

最重要的着手估值是势力效果, 其如何确定参见 13.4 影响和实地。

对于不能直接用势力表示的着手目标, 如组合攻击(不清楚几个棋串中会吃掉哪个的时候)、战略效果、联络等, 估值需要修改, 这里需要一个很大的启发规则, 如避免重复估值。这在 0 函数 `atari_atari` 尝试发现一个叫吃顺序最

终使得一不能预期的对方棋串的状态由 ALIVE 变为 CRITICAL。如果发现了此类顺序则尝试去掉无关的着手使之变短。

着手评估中详细描述。

4.4 详细的事件序列

以下是从 `genmove()` 调用 `examine_position()` 时的事件序列供参考。

`purge_persistent_caches()` (参见 11.3 持续的识别缓存)

`make_worms()` (参见 7.1 棋串):

`build_worms()` 查找并识别棋串

`compute_effective_sizes()`

`compute_unconditional_life()`

`find_worm_attacks_and_defenses()`:

 对于所有可攻击的棋串:

`set worm.attack`

`change_attack()` 加入攻击点

`find_attack_patterns()` 另外查找几个攻击点

 对于所有可防守的棋串

`set worm.defend`

`change_defense()` 加入防守点

`find_defense_patterns()` 另外查找几个防守点

测试所有棋串查找其他攻防

 当前气数 (`add_attack_move/add_defense_move`) (3.4) 寻劫

 查找更多的气数 (对于所有棋串)

 查找断点 (对所有棋串, `worm.cutstone`)

(3.4) 对于所有棋串计算棋形

改进攻防: 如果提掉邻串同时防守了本方棋串或攻击了对方棋串则加点。如果棋串既攻击对方同时又受攻, 则修正估值。

 查找直接攻击的棋串

 查找间接攻击的棋串

 识别不必要的棋串 (如点眼)

`compute_worm_influence()`: (参见 13 影响函数)

`find_influence_patterns()`

`value_influence()`

`segment_influence()`

`make_dragons()`:

`find_cuts()`

`find_connections()`

`make_domains()` (确定眼形)

`find_lunches()` (可以提掉的邻串)

`find_half_and_false_eyes()`

```

eye_computations(): 计算每个眼位的价值
    存储攻防点
analyze_false_eye_territory()
对每个棋块 compute_dragon_genus()
对每个棋块 compute_escape() 并设置逃跑路径数据
resegment_initial_influence()
compute_refined_dragon_weaknesses() (owl 后再次调用)
对每个棋块 compute_crude_status()
find_neighbor_dragons()
对每个棋块计算周围状态
对每个弱棋块运行 owl_attack() 和 owl_defend() 确定攻防点
对每个棋块计算棋块状态
对每个棋块共计棋串计算 owl 攻击点
对每个棋块计算棋块安全点
revise_inessentiality()
semeai():
    对于所有对杀局面, 运行 owl_analyze_semeai()
    find_moves_to_make_seki()
identify_thrashing_dragons()
compute_dragon_influence():
    compute_influence()
    break_territories() (参见 13.10 打入)
compute_refined_dragon_weaknesses()
    compute_effective_sizes()
    compute_unconditional_status()

```

以下是信息收集阶段完成后 genmove() 着手生成和选择阶段的事件序列。

```

choose_strategy()
collect_move_reasons():
    worm_reasons(): 对于每个攻防点加入着手目标
    semeai_reasons(): 对于每个 dragon2.semeai 点加入着手目标
    owl_reasons(): 对于每个 owl 攻防点加入着手目标
    break_in_reasons(): 对于每个打入点加入着手目标
fuseki()
break_mirror_go()
shapes(): 在棋盘内进行模式匹配 (参见 9.1 纵览)
combinations(): 查找双重目标的着手和其他策略
    find_double_threats()
    atari_atari()
review_move_reasons()
对于攻击棋串考虑:
    是活棋则重新评估局面
endgame_shapes()
endgame()

```

如果没有发现任何着手，重新评估对杀局面，修改对方死棋的状态

对于活棋再次运行 `shapes()` 和 `endgame_shapes()`

如果到此为止没有发现任何着手，运行 `fill_liberty()`

4.5 路线图

GNU Go 引擎包括在两个目录 `"engine/"` 和 `"patterns/"` 中，用户界面、DGF 文件读写以及测试代码在目录 `"interface/"`、`"sgf/"` 和 `"regression/"` 中，从其他 GNU 程序中借用的代码在 `"utils/"`，该目录中还包括了一些 GNU Go 中开发的但并不与围棋直接相关的程序。文档在 `"doc/"`。

在本文档中还将介绍一些 `"engine/"` 和 `"patterns/"` 中组成引擎代码的独立文件。在 `"interface/"` 中也涉及以下两个文件：

- `"gmp.c"`

GMP 协议接口（感谢 William Shubert 和其他作者）。它实现了与 Cgoban 或其他支持 GMP 协议的程序之间所有的配置和对弈接口。

- `"main.c"`

包括了 `main()` 使得 `"gnugo"` 目标代码在 `"interface/"` 目录中构造。

`"engine/"` 中的文件

4.5.1 在 `"engine/"` 中有以下文件：

- `"aftermath.c"`

包括在终局时生成着手所需的算法，可生成着手以确定棋盘局面，在 GNU Go 可能出错的地方，特别是有双活的局面时，推演一个局面以确定棋盘上所有棋群的状态。

- `"board.c"`

这个文件包括维护棋盘的代码，如它包括了重要的函数 `trymove()` 可在棋盘上尝试落子、`popgo()` 弹出堆栈回溯着手等，同时增量更新重要的信息如每串棋的气数和位置等。

- `"breakin.c"`

检查能够打入或防止打入预定区域的着手的代码。

- `"cache.c"` 和 `"cache.h"`

作为一个加速识别的方法，存储计算结果使得遇到相同局面，（如通过不同的次序）能够快速复用计算结果，以 hash 表实现。

- `"clock.c"` 和 `"clock.h"`

时钟代码，包括允许 GNU Go 自动调整级别以避免在计时比赛中超时负。

- `"combination.c"`

需要一连串的打吃和攻击方能提子的情形称为组合攻击。该文件包含了查找此类攻防的代码。

- `"dragon.c"`

包括 `make_dragons()`，该函数在着手生成模块 `shapes()`、`semeai()` 和其他着手生成器之前、`make_worms` 之后运行。尝试将棋串联络成棋块并收集它们相关的重要信息，如气数、（对手的）棋块是否可以吃掉、是否活棋等。

- `"endgame.c"`

查找终局时各种着手。

- `"filllib.c"`

终局时强制填单官（或提吃）的着手。

- `"fuseki.c"`

从数据库中生成开局定式着手。

- `"genmove.c"`

该文件包括 `genmove()` 及其支持例程，特别是

`examine_position()`。

- “globals.c”

包括 GNU Go 使用的主要全局变量。

- “gnugo.h”

该文件包括引擎公共接口的声明。

- “hash.c” 和 “cache.c”

实现 Zobrist hash 的代码 (参见 11.2 局面 Hash)。“hash.c” 中的代码提供将棋盘局面 hash 成紧缩表示且容易比较的方法。“cache.c” 中的缓存代码使用棋盘 hash 存取识别结果。

- “influence.c” 和 “influence.h”

该代码确定棋盘上哪些区域在哪方的影响之下 (参见 13 影响函数)

- “liberty.h”

引擎的头文件，其名为“liberty” 隐喻自由。

- “matchpat.c”

该文件包括模式匹配器 `matchpat()`，对于给定的棋盘局面搜索匹配模式。实际模式存储在 “patterns/” 目录。函数 `matchpat()` 由所有进行模式匹配的模块调用，特别是 `shapes`。

- “move_reasons.c” 和 “move_reasons.h”

跟踪维护着手目标的代码。

- “optics.c”

该文件包括识别眼型的代码，参见 8 眼和后手眼。

- “owl.c”

该文件做死活识别。着手生成基于模式，而使用 “optics.c” 中的代码评估眼位，是否有点致命着手和是否净活。能够成功逃出的棋块也视为活棋。该

文件也以同样原则实现了对杀识别。

- “persistent.c”

持续缓冲，使得以后着手或计算活动区域外棋子影响时可以再次使用识别结果。

- “printutils.c”

打印例程。

- “readconnect.c” 和 “readconnect.h”

该文件包括确定两棋串能否被切断和连接的代码。

- “reading.c”

该文件包括确定给定棋串能否被攻击和防御的代码，参见 11 战术识别。

- “score.c”

实现 Bouzy 算法 (参见 14 模样的另一方案: Bouzy 的 5/21 算法) 并包含了数子代码。

- “semeai.c”

该文件包括 semeai()，检查出于对杀状态的模块。为确定对杀结果使用了 “owl.c” 中的对杀识别。

- “sgfdeside.c”

对于各种识别生成 sgf 跟踪的代码。

- “shapes.c”

该文件包括 shapes()，为 genmove() 调用查找匹配的模式 (参见 9 模式代码)。

- “showbord.c”

该文件包括 showboard()，以 ASCII 编码表示绘制棋盘、棋块 (棋子用同一字母表示) 和状态 (颜色)。这是 GNU Go 1.2 最初的接口，现在只作为辅助调试所用。

- “surround.c”

确定一个棋块是否被包围，寻找包围和突破包围的着手的代码。

- “utils.c”

一系列例程，后面详述。

- “value_moves.c”

该文件包含的代码在生成所有着手目标后赋值给每个着手。同时还试图某些额外的着手目标。

- “worm.c”

该文件包括 `make_worms()` 代码，在每轮棋开始前运行，先于“draogon.c”中的代码，确定所有棋串的属性，包括气、是否可提掉（如何提掉）等。

4.5.2 “patterns/”中的文件

目录 “patterns/” 包括与模式匹配相关的文件。目前有几种类型的模式：

- “patterns.db” 和 “patterns2.db”文件中的着手生成模式
- “hoshi.db” 等文件中的着手生成模式，可从 “hoshi.sgf” 等文件自动生成。包括了小的 Joseki 库。
- “owl_attackpats.db” 、 “owl_defendpats.db” 和 “owl_vital_apats.db”文件中的模式，为 owl code 生成着手（参见 12.1Owl 代码）。
- “conn.db” 文件中的联络模式（参见 9.9 联络数据库）
- “influence.db” 和 “barriers.db” 文件中的影响模式（参见 13 影响函数）
- “eyes.db” 中的眼模式（参见 8 眼和后手眼）

以下列表包括除源代码文件以外自动生成的文件，如“patterns.c” ，

它们是通过“编译”各种模式数据库得到的 C 源代码，或者在一些情况下（如“hoshi.db”）是“编译”SGF 格式 joseki 文件自动生成的。

- “conn.db”

联络模式数据库。

- “conn.c”

由 mkpat 从“conn.db”编译自动生成的文件，包括结构数组形式的连接模式。

- “eyes.c”

由 mkpat 从“eyes.db”编译自动生成的文件，包括结构数组形式的眼位模式。

- “eyes.h”

“eyes.c”的头文件。

- “eyes.db”

眼型模式数据库，详见 8 眼和后手眼。

- “helpers.c”

支持 matchpat 估值的辅助函数。

- “hoshi.sgf”

星开局 SGF 文件。

- “hoshi.db”

编译“hoshi.sgf”自动生成的星开局模式数据库。

- “joseki.c”

定式编译器，从 SGF 格式的定式文件生成模式数据库。

- “komoku.sgf”

小目开局 SGF 文件。

- “komoku.db”

编译“komoku.sgf”自动生成的小目开局模式数据库。

- "mkeyes.c"

眼型数据库的模式编译器。输入"eyes.db" 输出"eyes.c" 。

- "mkpat.c"

着手生成和联络数据库的模式编译器。输入 "patterns.db" 和自动生成的 Joseki 模式文件 "hoshi.db"、 "komoku.db"、 "sansan.db"、 "mokuhadzushi.db"、 "takamoku.db" 输出 "patterns.c" 或输入 "conn.db" 输出 "conn.c" 。

- "mokuhadzushi.sgf"

目外开局 SGF 文件。

- "mokuhadzushi.db"

编译 mokuhadzushi.sgf 自动生成的目外开局模式数据库。

- "sansan.sgf"

三三开局 SGF 文件。

- "sansan.db"

编译"sansan.sgf"自动生成的三三开局模式数据库。

- "takamoku.sgf"

高目开局 SGF 文件。

- "takamoku.db"

编译 takamoku.sgf 自动生成的高目开局模式数据库。

- "patterns.c"

由 mkpat 从 patterns.db 编译的模式文件。

- "patterns.h"

与模式数据库相关的头文件。

- "patterns.db" and "patterns2.db"

可读格式存储的模式数据库。

4.6 代码风格和体例

4.6.1 代码体例

代码体例参见: http://www.gnu.org/prep/standards_toc.html

请在函数开头书写一段简短的使用说明。

请协助保持 Texinfo 文档及时更新。

4.6.2 跟踪

提供 `gprintf()` 函数, 是 `printf` 的剪裁版本, 仅支持 `%c`、`%d`、`%s`, 且无域宽度等, 但增加了一些有用的功能:

- `%m`

带两个参数, 显示格式化的棋盘坐标。

- `indentation`

跟踪信息根据当前堆栈位置自动缩进, 使得在识别时很清楚是前进还是回溯。

- `"outdent"`

禁止缩进。

一般地, `gprintf()` 封装在下面的一种形式:

```
TRACE(fmt, ...):
```

如果“verbose”变量>0, 打印消息(verbose 在命令行由-t 设定)

```
DEBUG(flags, fmt, ...):
```

TRACE 用来展示 GNU Go 考虑的概况, DEBUG 允许对某个模块深入研究, 一般在出错的时候需要。flags 是“engine/gnugo.h”中的一个 `DEBUG_*` 符号。DEBUG 宏检测 debug 变量中该位是否置位, 如果是则打印消息。debug 变量在命令行由 -d 设定。

变量 `verbose` 控制跟踪，可以是 0（不跟踪）、1、2、3 或 4 跟踪级别递增。可以在命令行用 `-t` 设置 `verbose=1`、`-t -t` 设置 `verbose=2` 等。实际上如果需要 1 级以上得跟踪级别最好使用 `gdb` 到达需要跟踪的断点，变量 `verbose` 经常被临时复位为 0，可以用 `gdb` 命令 `set var verbose=1` 将跟踪打开。

4.6.3 断言

与跟踪相关的是断言。强烈建议开发者在代码中使用断言以保证数据结构符合他们的预期。例如，辅助函数加入断言冻结所评估的着手周围的棋盘内容。

`ASSERT()` 封装在标准 C `assert()` 函数上。为增强测试，使用了一对额外的参数表示“相应的”棋盘局面。如果断言失败，棋盘位置包括在跟踪输出内，调用 `showboard()` 和 `popgo()` 显示堆栈。

4.6.4 FIXME

我们采用这样的体例，在注释中加入 `FIXME` 表示已知的错误。

4.7 源码浏览

如果使用 `Emacs`，可以快速方便地使用 `Emacs` 内置的功能浏览源码。切换到根目录“`gnugo-3.6.x/`”并执行：

```
find . -print|grep "\.[ch]$" | xargs etags
```

将生成一个名为“`gnugo-3.6.x/TAGS`”的文件，为查找任何一个 `GNU Go` 函数，键入 `M-.` 和你希望找的命令，或将光标放到要找的函数上，键回车即可。

首次执行时会提示 TAGS 表的位置，键入 “gnugo-3.6.x/TAGS”的路径以后就可以用最少的击键次数查找文件。

5 分析 GNU Go 的着手

本章讨论 GNU Go 对局面的识别。这些方法对于程序员非常有趣。

实际上大多数对于 GNU Go 的调整是与维护 regression/目录相关的(参见 20 回归测试)。

假定有一个存储为 sgf 文件 GNU Go 的对局，想了解它如果确定着手。

5.1 解释跟踪

5.2 输出文件

5.3 检查识别码

5.4 检查 Owl 码

5.5 GTP 和 GDB 技术

5.6 调试盘

5.7 数子

5.8 彩色显示

5.1 解释跟踪

一个快速找到着手大致目标的方法是运行：

```
gnugo -l filename -t -L move number
```

(可以增加 `--quiet` 去掉版标信息。) GNU Go 3.6 中着手与目标同时列出，并列每着的分析价值。

如果要调整(参见 9.11 调整模式数据库)，可以增加 `-a` 选项，使 GNU Go 报告包括不影响着手选择的所有已匹配的模式。这样可以调整已有模式而不是引入新模式。

使用 `-w` 选项，GNU Go 报告棋盘上棋串和棋块的状态，信息采用另一种

格式（参见 5.6 调试盘和 5.8 彩色显示）。

5.2 输出文件

从选项 `-o filename` 启动 GNU Go 可生成一个输出文件，在 Cgoban 下此参数也可在 Go Modem Protocol Setup 窗口中加入。输出文件显示所考虑的着手及其权重。值得一提的是如果将 CGoban 窗口拉到最大可以显示三位数字。DEAD 状态的棋块标记为 X，CRITICAL 状态标记为!。

如果你的棋局文件不符合标记格式，或者不是由最新版本的 GNU Go 生成的，可运行：

```
gnugo --quiet -l <old file> --replay <color> -o <new file>
```

生成合法标记。

这里 `<color>` 可以是 black、white 或 both。复盘选项也可以发现与旧版本 GNU Go 着手不同的地方。

5.3 检查识别码

`--devide-string` 选项可用以检查战术识别代码（参见 11 战术识别）。

该选项带有一个参数，输入棋盘位置的通用字母表示（如 `--devide-string C17`）。结果可以反映识别代码（在 `engine/reading.c` 中）是否认定棋串能被吃掉，如果是，是否认定可以防守、用以攻防的着手，以及在得出此结论前搜索了多少个节点。注意在 GNU Go 正常运行时（不带 `--devide-string`），攻防点在运行 `make_worms()` 时计算并缓存在 `worm.attack` 和 `worm.defend` 中。

和输出文件（`-o filename`）同时使用，`--devide-string` 会生产一个变化树显示其考虑过的所有变化。这是调试识别代码很有用的方法，且可以展

示它是如何工作的。变化树可用 CGoban 图形显示。

每个节点的注释包含一些信息，如有：

```
attack4-B at D12 (variation 6, hash 51180fdf)
break_chain D12: 0
defend3 D12: 1 G12 (trivial extension)
```

可以解释为：搜索节点在对于 D12 上的棋串调用 engine/reading.c 中函数 attack3() 时生成。括号内的数据表示 count_variations 和 hashdata.hashval 的值。

第二个值 (hash) 除非调试 hash 代码否则无需关心。但第一个值 (variation) 对于 gdb 调试时非常有用，可以先用 -o 选项制作一个输出文件，然后用 gdb 进入代码，用 SGF 文件在调试器中给出坐标，显示 count_variations 值。特别地可从调试器中找到你的所在：

```
(gdb) set dump_stack()
B:D13 W:E12 B:E13 W:F12 B:F11 (variation 6)
```

如果停在生成当前着手的 trymove() 调用之后，SGF 文件中的变化数应与 dump_stack() 显示的变化数一致，且当前着手即最后一着（例子中的 F11），显示到达当前变化的着手顺序，并打印 count_variations-1。

后面两行表明从此节点在 D12 调用了函数 break_chain() 并返回 0，表示没有找到能够通过攻击包围圈中一点解救该棋串的方法，且在 D12 调用了函数 defend3() 返回 1 表示该棋串能够防守，防守着手为 G12。如果寻找生成这些注释的函数有困难，可以尝试设置 sgf_dumpptree=1 并在 sgf_trace 设置断点。

5.4 检查 Owl 码

同样可以使用选项 --decide-dragon 调试 Owl 代码。用法也与

`--devide-string` 相似，同样生成变化树，但一般都比 `--devide-string` 生成的小。

5.5 GTP 和 GDB 技术

可以使用 GTP 协议(参见 19GTP 协议) 确定棋块状态和其他调试所需信息。

GTP 命令 `dragon_data P12` 可列出 P12 处棋块的数据，`worm_data` 可列出棋串数据；其他 GTP 命令也同样有用。使用 GDB 同样可以方便地得到这些信息。在 11.9 调试识别代码推荐 `.gdbinit` 文件，如果装载了这个文件，可以用以下命令列出棋块数据：

```
(gdb) dragon P12
```

同样用 `worm P12` 可以列出棋串数据。

5.6 调试盘

`"interface/debugboard/"` 目录中有一个有用的工具 `debugboard`，可在 Xterm 中运行。由于它显示 50 行 80 列，所以使用小字体。它运行 `examine_position()` 然后制作一个棋盘的图形显示。用光标方向键可在棋盘上移动并读取棋串、棋块和眼数组的信息。

5.7 数子

GNU Go 有数子功能。一般地，GNU Go 会在终局报告它的数子结果，如果要把棋局的这些信息存入文件，可使用选项 `--score` (参见 3.9 启动 GNU Go：命令行格式)。

5.8 彩色显示

使用彩色 `xterm` 或 `rxvt` 窗口可获得彩色显示功能。`Xterm` 编译进彩色支持后才可以正常工作，如果没有彩色显示，试 `rxvt`。也可以使用 Linux 控制台。背景色是黑色时显示效果更好，否则可编辑 `.Xdefaults` 文件，给 `xterm` 或 `rxvt` 加入选项 `-bg black -fg white`。在 Mac OS X 上可在 `.tcshrc` 文件加入 `setenv TERM xterm-color` 使能终端彩色。

5.8.1 棋块显示

可以用彩色显示 ASCII 棋盘，每个棋块分配一个不同的字母，不同的 `i matcher_status` 值 (`ALIVE`、`DEAD`、`UNKNOWN`、`CRITICAL`) 用不同的颜色表示，这使得调试很好掌握。实际上生成了两个图形，原因是匹配状态的计算方法。首先计算 `dragon_status` (参见 7.5 棋块 `s`) 然后对一些棋块而不是所有棋块计算更精确的 `owl` 状态。如果有 `owl` 状态匹配状态就是该状态，如果没有，匹配状态就是 `dragon_status`。`dragon_status` 和 `owl_status` 都显示，颜色图例如下：

```
绿色 = alive
蓝色 = dead
红色 = critical
黄色 = unknown
紫色 = unchecked
```

用 `CGoban` 或 `GNU Go` 带 `-o` 选项将棋局按 `sgf` 格式存储到一个文件，打开一个 `xterm` 或 `rxvt` 窗口，执行 `gnugo -l [filename] -L [movenum] -T` 就得到彩色显示。

其他有用的彩色显示可以按如下方式获取：

5.8.2 眼位显示

用 `-E` 代替 `-T`, 可以得到眼位的彩色显示。盘边的眼位标记! (参见 8 眼和后手眼)。

5.8.3 模样显示

选项 `-m level` 给出 `engine/moyo.c` 计算的各种数值的彩色显示。

GNU Go 包含实地、模样和区域概念 (参见 0 不能被战术攻击, 或有战术防守并轮到先手的棋子称为战术活。类似地, 不能被战略攻击 (死活分析的意义), 或有战略防守并轮到先手的棋子称战略活棋。如果要在确定战略状态前使用影响函数, 则战术活棋都视为战略活棋。

棋盘上所有战略活棋都作为影响源, 其影响向各方向辐射。影响力随距离指数下降。

影响仅当棋盘空时无阻碍扩张, 所有战术活棋 (无论何方) 都作为影响障碍, 如不能穿越对方棋子之间的联络。如一间跳除非其中一子可被吃掉都作为障碍。注意两子之间的联络是否能被打穿并没有多少影响, 因为两个方向上都有影响。

从双方的影响计算出每点介于 -1.0 和 $+1.0$ 之间的实地值, 可视为各方成实地的可能性。

为避免出现假实地, 在可能的侵入点加入额外的影响源, 如在座子下的 3-3、在宽边缘扩展的中间和任何大的开放空间的中心。类似地在看起来是实地的但可能侵入的地方如小飞增加额外的影响源。这些侵入依赖于何方先手。

所有额外的影响源, 和联络一样由模式数据库的两个文件 `patterns/influence.db` 和 `patterns/barriers.db` 控制。细节在 13.12 Patterns used by the Influence module 描述。

地、模样和区域) 两种不同的实现: 首先, GNU Go 用影响函数 计算实地、

模样和区域并以函数 `whose_territory()`、`whose_moyo()` 和 `whose_area()` 回报。为获取该模块得出的影响范围的彩色显示，可用 `-m 0x18` 查看初始影响，如 `-m 0x10 --debug-influence D5` 查看 D5 后的影响。另外还有各种其他选项数值化显示影响，细节描述参见 13.13 影响的彩色显示和调试。

Bouzy 算法(参见 14 模样的另一方案：Bouzy 的 5/21 算法)得出的区域仅用于函数 `estimate_score()`，用以下选项显示：

`-m level`

用或逻辑按位显示：

1	0x01	ascii 实地估值显示 (5/21)
2	0x02	ascii 模样估值显示 (5/10)
4	0x04	ascii 区域估值显示 (5/10)

`-m` 选项可以加上级别组合使用。

6 着手生成

6.1 简介

6.2 着手目标生成

6.3 各种着手目标详解

6.4 着手评估

6.5 终局

6.1 简介

GNU Go 3.0 引入了一个与早期版本有着相当差别的着手生成机制。

在原机制中，各种着手生成器给出带估值的不同着手，然后确定价值最高的着手。这种机制有两个主要的缺点：

- 多目标的有效着手只能由专门用来寻找特定组合的模式发现，如同时联络和分断。同时也没有好的方法在几个攻着中选择。
- 绝对的着手价值随着模式数量的增加越来越难以调整。同时它们是相当主观的，调整中一些东西的改变很容易会意想不到地破坏原来的东西，如棋串评估。

着手生成的新机制是各种着手生成器推荐着手目标，如提子或连接两棋串等。不同着手的所有目标都发现后，开始评估。其主要优点在于：

- 与原机制中的着手价值不同，着手目标是客观的。任何人都可以验证推荐的着手目标是否正确。
- 集中的着手价值更便于调整。同样允许与棋风相关的调整，如影响与实地相比的估值。在胜势下增加安全着手的价值也成为可能。

6.2 着手目标生成

着手生成器分别推荐一些着手，并赋予一个或多个着手目标。在各推荐点

集中着手目标以备以后评估。 GNU Go 考虑的部分着手目标如下：

ATTACK_MOVE

DEFEND_MOVE

棋串攻防

ATTACK_THREAT_MOVE

DEFEND_THREAT_MOVE

威胁棋串攻防

EITHER_MOVE

双目标着手（目前只用于攻击棋串）

ALL_MOVE

目前用于防守两个遭受双重攻击的棋串

CONNECT_MOVE

CUT_MOVE

联络或分断两棋

ANTISUJI_MOVE

声明坏棋或禁止

SEMEAI_MOVE

SEMEAI_THREAT

赢取或威胁赢取对杀

EXPAND_TERRITORY_MOVE

EXPAND_MOYO_MOVE

扩张实地 / 模样的着手。这些目标目前同样处理。

VITAL_EYE_MOVE

死活关键着手

STRATEGIC_ATTACK_MOVE

STRATEGIC_DEFEND_MOVE

“a” 类和 “d” 类模式加入的攻防模块的（也许是模糊的）着手（参见 9.2

模式属性）

OWL_ATTACK_MOVE

OWL_DEFEND_MOVE

owl 攻防着手。

OWL_ATTACK_THREAT

OWL_DEFEND_THREAT

威胁 owl 攻防着手

OWL_PREVENT_THREAT

解除 owl 威胁着手

```
UNCERTAIN_OWL_ATTACK  
UNCERTAIN_OWL_DEFENSE
```

不确定的 owl 攻防。即 owl 代码因节点限制无法确定结果。

```
MY_ATARI_ATARI_MOVE
```

开始一连串的叫吃，最终提子的着手。

```
YOUR_ATARI_ATARI_MOVE
```

对方可发起的一连串叫吃最终提子，而本方是安全的着手。这样的着手先走几乎总可以防守威胁。

攻防着手类型可用后缀表示其结果依赖于劫争，如 OWL_ATTACK_MOVE_GOOD_KO。这里..._GOOD_KO 和..._BAD_KO 分别与 KO_A 和 KO_B 相关，在 11.4 劫争操作说明。详细着手目标参见 engine/move_reasons.h。

注意：其中还有一些不着手目标。

下节讨论更详细的着手目标。

6.3 各种着手目标详解

6.3.1 攻防着手

6.3.2 威胁攻防

6.3.3 攻防多用着手

6.3.4 切断和联络

6.3.5 赢取对杀的着手

6.3.6 做眼和破眼

6.3.7 坏棋

6.3.8 实地

6.3.9 攻防棋块

6.3.10 组合攻击

6.3.1 攻防着手

可战术提掉一棋串的着手称攻击着手，解救将被战术提掉棋串的着手称防守着手。可以理解仅当该棋串可被提掉时才存在防守着手，且一棋串仅被收气或回填着手攻击时没有防守。

重要的是发现某棋串的所有攻防着手，使着手生成器可以有意义地选择如何执行提吃，或发现提吃和 / 或防守几个棋串。

攻防着手首先在 `make_worms` 评估所有棋串战术状态时发现，该步对每个棋串只给出一个攻防着手。紧接着还是在 `make_worms`，测试被攻棋串所有气点寻找额外的攻防着手。更间接的着手则在 `find_attack_patterns` 和 `find_defense_patterns` 匹配 `"patterns/attack.db"` 和 `"patterns/defense.db"` 中的 A(攻击) 和 D(防守) 类模式时寻找。最后一步，测试所有填有意图的着手，是否额外攻防一些棋串。（只分析未稳定的棋串。）

6.3.2 威胁攻防

威胁攻击一棋串，而该棋串可防守，是第二着手目标。这个着手目标可强化着手的价值。没有其他目标的威胁着手还可用作劫材。威胁防守也一样，如果攻方不 `tenuki`，该棋串仍要被提掉。

`owl` 代码发现的威胁着手称 `owl` 威胁，且有自身的 `owl` 目标。

6.3.3 攻防多用着手

有时一着手至少攻击几个棋串中的一个或同时防守几个棋串。这样着手标记自身的目标。

6.3.4 切断和联络

连接两独立棋块的着手称联络着手。阻止这种联络的着手称分断着手。分断和联络着手首先由模式匹配中 C 和 B 类模式发现。

分断和联络着手第二个来源是分断棋子的攻防。攻击一棋串时，如果多个棋块与被攻棋串相邻，则攻着有自动被视为联络着手。类似的守着也被视为分断着手。发现这类模式时就引发一个联络或分断目标。

登记分断或联络目标时当然要存储相关棋块。同一着手可能分断并 / 或联络多个棋块。

6.3.5 赢取对杀的着手

为赢取对杀的必要着手称对杀着手。与攻击着手类似，但同时包括了攻防。象攻防着手一样，重要的是发现所有赢取对杀的着手，以便有意识地选择。

对杀着手由对杀模块设置，目前尚未实现。也可能希望列出对杀中提高领先位置的着手作为第二着手目标。落后位置类似。

6.3.6 做眼和破眼

通过眼位影响眼数量的着手称眼着。眼对于棋块死活是否关键并不必要，但如果是接着会评估较高价值。通常重要的是发现所有改变眼数的着手。

(这是 `eye_finder` 做的部分，目前对每一不确定的眼位只寻找一个致命点。)

6.3.7 坏棋

局部低优先或因某些理由不能落子的着手称坏棋。这些坏棋由模式匹配去成。由于这类着手在任何情况下都不走，所以要关注。

6.3.8 实地

增加实地的着手是一个目标，称扩大实地目标。该着手目标由“patterns/patterns.db”中“e”模式加入。类似地“E”模式试图制造或侵消模样，即尚未确定为实地但有价值的影响区域。这样的模式设置“扩大模样”着手目标。

6.3.9 攻防棋块

就象 战术识别代码试图确定棋串攻防，owl 代码试图确定棋块取得两眼活棋。函数 owl_reasons() 生成相应的着手目标。

owl 攻击和 owl 防守着手目标是自解释的。

当 owl 攻击一对方棋块失败但 owl 代码确定连走两手可以提掉该棋块时生成 owl 攻击威胁目标。攻杀着手存储在 dragon[pos].owl_attack_point 和 dragon[pos].owl_second_attack_point 。

类似地如果一同方棋块是死棋但两手可以救，则生成一个 owl 防守威胁着手目标。

防止威胁目标类似，但方位相反：如果对手有一个攻击威胁着手则解除这一威胁的着手即得到防止威胁着手目标。

owl 代码节点不足时生成 owl 不确定着手目标。为防止 owl 代码运行过长时间，设定一次 owl 识别可以生成的最大节点数。如果超出了，识别就中断并照常存储结果，但标记为不确定，这时生成一个 owl 不确定着手目标。例如 owl 代码发现棋块活但不确信，还会生成一个防守着手。

6.3.10 组合攻击

函数 `atari_atari` 尝试发现一个叫吃顺序最终使得一不能预期的对方棋串的状态由 `ALIVE` 变为 `CRITICAL`。如果发现了此类顺序则尝试去掉无关的着手使之变短。

6.4 着手评估

着手生成过程结束时，函数 `value_move_reasons()` 为选择最佳着手而尝试为着手赋予价值。着手评估的单一目的是尝试将着手排序以使最佳着手评分最高。原则上这些价值可以是强制的，但为了便于评估估值的执行效果，并简化调整，我们尝试使用与人类棋手使用的计算方法一致的赋值，例如象 Ogawa 和 Davies 在 *The EndgameGo* 中解释的那样。

着手考虑四个方面的因素来评估：

- 实地价值
- 战略价值
- 棋形价值
- 第二价值

所有这些是浮点数且可用实际点数衡量。

实地价值是该着手引起的预期实地变化的总量。如果该着手是攻击或防守着手，这还包括群棋群状态的变化。

从 GNU Go 3.0 起，影响函数在实地估计中扮演了一个重要的角色（参见 13.4 影响和实地）。它用来推测每个点属于黑或白。实地价值累计了这些估值的变化。

战略价值是度量着手对于盘上所有棋群安全性的影响。典型的分断和联络这里有主值。边界扩张、包围和向中央的着手战略价值较高。战略价值是相关棋块实地价值一部分的总和。该部分由棋块安全的变化确定。

棋形价值是纯粹的局部棋形分析。一个重要的作用是修正实地价值评估产

生的错误。在开放的局面中，经常值得牺牲几个点（明显的）现实利益而换取一个好形。棋形价值由模式匹配中棋形模式实现。

第二价值为本身不充分的着手目标给出。一个实例是对一个有几个眼的棋块减少眼数，或者攻击一个不能防守的棋串。

所有这些价值计算出来后，累计，可能还要加权（第二价值肯定权重较小），得出着手最终价值。该值用于确定着手。

6.4.1 实地价值

6.4.2 战略价值

6.4.3 棋形因数

6.4.4 最小值

6.4.5 第二值

6.4.6 威胁和后续价值

6.4.1 实地价值

计算实地价值的算法在函数 `estimate_territorial_value` 中。顾名思义，它估计实地的变化。

它考虑了所有因此着手状态由活变死或相反的棋群，同时估计该着手是否可认为安全，如果是，则调用影响模块：函数 `influence_delta_territory` 估计双方棋子和棋群状态变化对实地的影响。

影响模块返回的结果经过一些修正。这是因为一些着手目标无法单独调用影响函数来评估，如依赖于劫争的着手。

6.4.2 战略价值

所有匹配了类型“a”或“d”模式的着手赋予战略攻击或防守目标。这些着手某种（经常是模糊的）意义上是帮助加强或削弱一个棋块。当然加强一个已

经活的棋块得不到太多价值，但生成着手目标时不需要检查状态或安全，是后来在评估阶段做的。

6.4.3 棋形因数

在模式的价域中（参见 9.3 模式属性——值）可以规定一个棋形值。

这用于计算棋形因数，再乘以着手评分。我们取棋形最大正贡献，并对找到的每个额外的正贡献加 1。接着取棋形最大负贡献，并对每个额外的负贡献加 1。数据结果乘 1.05 得到棋形因数。

这个复杂机制背后的原理是所有棋形点都很显著。如果找到两个棋形贡献（如值为 5 和 3），第二个贡献就降为 1。否则引擎就太难调整，因为找到多个棋形贡献就会导致着手显著地过估。

6.4.4 最小值

模式可以赋予一个最小值（有时也有最大值）。例如定式模式有这样预定义的值，或者有 `value` 域。一般不愿使用这手段，但实际上有时需要。

在定式中，经常有几个着手有相同的最小值。GNU Go 在这些着手中随机选择，保证了 GNU Go 对奕的不确定性。在后面的棋局中，GNU Go 对这样着手的基本评估用作第二标准。

6.4.5 第二值

第二着手目标权重很轻。如果所有其他因数都相等，这样的着手可以辅助度量。

6.4.6 威胁和后续价值

后续价值是指在同一局部连走两手的价值。赋予威胁棋串或棋块攻防的着手。另外，从 GNU Go 3.2 开始影响模块对可能的纯粹实地后续着手进行评

佑。在这两方法都不充分的情形下，我们给模式增加一个 `followup_value` 自动辅助宏。

通常后续值只给出小贡献，如后续值很大的话 GNU Go 就会将该着手作为主选估值加倍。而如果盘上最大的着手是非法的劫的话，该着手就成为劫材，所为后继值都参与计算。

6.5 终局

GNU Go 象生成其他着手那样生成终局着手。事实上不存在明确的终局概念，如果找到的最大着手为 6 点以下，则匹配 `"endgame.db"` 中额外的一组模式并重做着手评估。

7 棋串和棋块

7.1 棋串

7.2 合并

7.3 联络

7.4 后手眼和假眼

7.5 棋块

7.6 彩色棋群显示

考虑着手之前，GNU Go 将一些数据收集到几个数组中，本章讨论其中两个 `worm` 和 `dragon`。其余的见 8 眼和后手眼。

这些信息用以辅助评估每个棋群的联络度、眼形、逃跑可能性和 `T` 死活状态。

以后 `genmove()` 调用的例程会访问这些信息。本章说明由“`dragon.c`”中的两个例程 `make_worm()` 和 `make_dragon()` 执行基础分析的原理和算法。

棋串 是棋盘上通过水平线或垂直线相连的一串同方棋子的最大集合。我们经常称之为串。

棋块 是可视为一个单元的棋串的联合体。棋块在每个着手后重新生成。两棋串属于同一棋块，计算机即假定它们死活相同并有效联络。

棋块代码的目的在于允许计算机形成有意义的死活判断。举个例子，考虑以下情形：

```
OOOOO
OOXXXOO
OX...XO
OXXXXXO
OOOOO
```

这里 `x` 可以认作一个整体的群，有一个三目的眼，但由两个独立的棋串组

成。这样我们必须将这两个棋串合并为一个棋块方可得出有意义的结论，在中心点着手可杀死或救活棋块，这对双方都是关键点。如果不首先把 x 看作一个整体就很难得到这个结论。

棋块代码目前的实现包含了简化的假定，在以后的实现中可以改进。

7.1 棋串

结构数组 `struct worm_data worm[MAX_BOARD]` 收集了棋串的信息。

在此给出各域的定义，每个域具有棋串各点的常量。

```
struct worm_data {
    int color;
    int size;
    float effective_size;
    int origin;
    int liberties;
    int liberties2;
    int liberties3;
    int liberties4;
    int lunch;
    int cutstone;
    int cutstone2;
    int genus;
    int inessential;
    int invincible;
    int unconditional_status;
    int attack_points[MAX_tactics_POINTS];
    int attack_codes[MAX_tactics_POINTS];
    int defense_points[MAX_tactics_POINTS];
    int defend_codes[MAX_tactics_POINTS];
    int attack_threat_points[MAX_tactics_POINTS];
    int attack_threat_codes[MAX_tactics_POINTS];
    int defense_threat_points[MAX_tactics_POINTS];
    int defense_threat_codes[MAX_tactics_POINTS];
};
```

● color

棋串是 BLACK 或 WHITE 这就是 color。空串另有一个属性称 bordercolor，取值为 BLACK_BORDER、WHITE_BORDER 或

GRAY_BORDER。特别地，如果一个给定的空串，与其相连接的棋串是同方的，（黑或白）则以此定义 其 `bordercolor.`，否则 `bordercolor` 是灰色。

并没有定义一个新域，而是使用了 `color` 域存储这些数据，这样每个棋串 `color` 域取值为：BLACK、WHITE、GRAY_BORDER、BLACK_BORDER 或 WHITE_BORDER 之一。最后三个是用边界色分类的空串

- `size`

该域包含棋串的子数

- `effective_size`

棋串的棋子数目加上至少较其他棋串同样靠近该棋串的空点。各棋串共有的空点按相同比例计算。这衡量吃掉一个棋串直接的实地价值。

`effective_size` 是浮点数。只计算距离小于等于 4 的点

- `origin`

每个棋串都有一个特定的成员，称 `origin`，该域的作用是使得判断两点属于同一棋串更容易：比较 `origin` 即可。如果想对每个棋串同时执行一些测试，也可以简单地在 `origin` 操作而忽略其他点。`origin` 可由以下测试确定：`worm[pos].origin == pos`。

- `liberties`

- `liberties2`

- `liberties3`

- `liberties4`

对于非空棋串，`liberties` 表示气数，LIBERTIES2、LIBERTIES3 和 LIBERTIES4 补充表示第二、第三和第四顺序气。定义大于 1 的棋适用于检测包围漏洞，特别是需要查看棋群是否被松散包围的时候。顺序 `n` 的气 是落下不少于 `n` 个棋子就可以连接到同方棋串的空点。连接的通道可以穿过中间的同方棋群，落在距离大于 1 的棋子不能接触对方棋子，不允许通过劫争点。这样如

下棋形，我们 \circ 方标记小于顺序 5 的气

.XX...	.XX.4.
XO....	XO1234
XO....	XO1234
.....	.12.4.
.X.X..	.X.X..

顺序大于 1 的气数不能接触对方棋群的约束意味着小飞和单关是无法突破的屏障。这用于确定棋串是否正被包围。

通道不能穿过距离为 1 而两侧都是对方棋子的气点。反映在如下棋形，棋子 \circ 被两个 \times 棋子与左面隔离：

```
X.  
.O  
X.
```

我们称 n 是棋块到顺序为 n 的气点的距离。

- lunch

非零时 lunch 指向可被简单提掉的棋串（与该串是否可防守无关）。在域 worm.cutstone 和 worm.cutstone2 中表示并行使用的两种不同的分断概念。

- cutstone

分断棋子为 2，可能的分断棋子为 1，其他为 0。定义：分断棋子即与对方没有共同气点的两棋串都相邻的棋子。最常见的分断棋串如下：

```
XO  
OX
```

可能的分断棋子 是与共有 1 气的对方两棋串都相邻的棋子，如下的 \times ：

```
XO  
O.
```

对于分断棋串 worm[].cutstone=2，对于可能的分断棋串 worm[].cutstone=1。

- cutstone2

分断点由联络数据库中的模式识别。对于包围棋块分断或联络的着手同样

要考虑攻防着手，所以可以处理一般的分断。`cutstone2` 由 `make_domains()` 调用的 `find_cuts()` 设定。

- `genus`

对于棋串和棋块 `genus` 有两种不同的概念。棋块的概念更重要，因此 `dragon[pos].genus` 远比 `worm[pos].genus` 有用。两者目的在于对眼数的近似估计。棋串的 `genus` 是它本身连接的附属串减一，近似棋串的眼数。

- `inessential`

`inessential` 棋串是满足这样一个判别标准的棋串，除非吃掉包围它的对方棋串，否则不可能活。更精确地，`inessential` 棋串是 `genus` 为零、与其相邻对方棋串都不能被简单提掉，且没有边界气或者第二顺序气，并满足更进一步的特性：如果从棋盘上移走该棋串，留下的空都只与对方棋串相邻。

- `invincible`

`invincible` 棋串是 GNU Go 认为不能被提掉的棋串。函数 `unconditional_life()` 试图找出即使对手连续落子也无法提掉的棋串。

- `unconditional_status`

无条件状态也由函数 `unconditional_life` 设定。将把绝对活的棋串设定为 `ALIVE`；即使防守方允许连续下任意手数也无法转为绝对活的棋串设为 `DEAD`；对方无法构筑活棋的空点称无条件实地，根据所属设定为 `WHITE_TERRITORY` 或 `BLACK_TERRITORY`；最后，如果一个棋子能被提掉，但是它与同方的无条件实地相邻，也设定无条件状态为 `ALIVE`。其他所有情形，无条件状态是 `UNKNOWN`。

要使这些定义有意义，需要注意所有通常意义下的活子（即使是双活）可以通过一连串的着手成绝对活棋群，但这不总是正确的，因为有少数双活棋形不

满足这个条件。完善这些定义是为读者留下的一个练习。目前 `unconditional_life` 严格遵循上面定义，称这样的双活棋群无条件死，这显然错误。算法略微复杂一些即可能避免这个问题，但这留待以后的版本解决。

- `int attack_points[MAX_TACTICS_POINTS]`
- `attack_codes[MAX_TACTICS_POINTS]`
- `int defense_points[MAX_TACTICS_POINTS];`
- `int defend_codes[MAX_TACTICS_POINTS];`

如果战术识别代码（参见 11 战术识别）寻找到能被攻击的棋串，`attack_points[0]` 是攻击点，`attack_codes[0]` 是攻击代码：`WIN`、`KO_A` 或 `KO_B`。如果有多个攻击，使用 `attack_points[k]` 和 `attack_codes[k]` 表示。防守代码和防守点类似。

- `int attack_threat_points[MAX_TACTICS_POINTS];`
- `int attack_threat_codes[MAX_TACTICS_POINTS];`
- `int defense_threat_points[MAX_TACTICS_POINTS];`
- `int defense_threat_codes[MAX_TACTICS_POINTS];`

这些是威胁攻防棋串的点。

函数 `makeworms()` 生成所有棋串数据。

7.2 合并

棋块是作为一个单元的棋群，假定这些棋子有同样死活状态。这样如果棋串合并成为棋块，程序不会试图进行分断。

函数 `make_dragons()` 维护独立的包含类似数据的 `worm[]` 和 `dragon[]`，并将棋串没合并为棋块。棋块是棋串的联合体。所有棋串的 `worm[]` 数组数据是常量，所有 `dragon[]` 数组数据也是常量。

GNU Go 中棋串合并过程如下：首先合并眼形周围的附属，如下例中四个 x

棋子是同一空穴的边界，空穴中可以包含对合并没有影响的死串，故可以合并。

```
.0000.
00XX0.
OX..XO
OX..XO
00XX0.
XXX...
```

这类合并的代码在例程 `dragon_eye()`，将在眼一章介绍。

下一步，合并看起来无法分断的棋串。即合并有两个以上公共气点，或有一个对方无法落下而不被提掉的气点（忽略劫争规则）。

```
X.   X.X   XXXX.XXX   X.O
.X   X.X   X.....X   X.X
      XXXXXX.X   OXX
```

联络模式库在“`patterns/conn.db`”中。

7.3 联络

在对方可能分断的地方由“`conn.db`”中的 B（打入）类模式设定域 `black_eye.cut` 和 `white_eye.cut`。该域有两个重要的用途，分别由辅助函数 `xcut()` 和 `ocut()` 访问。第一种用途是在如下局面停止合并：

```
..X..
00*00
X.O.X
..O..
```

X 可落在 * 位分断各支。这里首先是联络模式 CB1 发现双重分断并将 * 标记为断点。接着搜索 `conn.db` 中的 C（联络）类模式寻找安全的联络合并棋块。一般小尖联络被视为安全的并按联络模式 CC101 合并，但有个约束，即要求两个空节点都不是分断点。

这个体系的弱点是 X 只能分断一个联络而不能分断两个，所以应当允许在其中一个联络上合并。这由联络模式 CC401，在

`amalgamate_most_valuable_helper()` 辅助下确定使用哪个联络。

另一个用途是简化备选联络模式到实连接。辅助函数 `diag_miai` 按联络模式 12 确定一个联络的局面需要标记为分断点，这样可以写出类似 CC6 的联络模式：

```
?xxx?      直接长出联络
XOO*?
O...?

:8,C,NULL

?xxx?
XOOb?
Oa..?

;xcut(a) && odefend_against(b,a)
```

这里我们可以证明在 * 的着手可是对方不能安全的落子在分断点，从而阻止了分断。

7.4 后手眼和假眼

后手眼是这样的位置，如果防守方先，则眼可以现实化，而如果攻击方先，则无法现实化。假眼 是这样的节点，它为一棋块包围但不是眼。后手眼的例子：

```
XXXXXX
OO..X
O.O.X
OOXXX
```

假眼：

```
XXXXXX
XOO.X
O.O.X
OOXXX
```

其他地方描述确定后手眼和假眼的“拓扑”算法（参见 8.8 后手眼和假眼拓扑）。

后手眼数据集中在 `dragon` 数组中。在此之前，由辅助数组

half_eye_data 填充这些信息。对应寻找到的类型，域 type 或者是 0，或者是 HALF_EYE 或 FALSE_EYE。域 attack_point[] 指向攻击后手眼的 4 点，类似地， defense_point[] 指向防守后手眼的点。

```
struct half_eye_data half_eye[MAX_BOARD];

struct half_eye_data {
    float value;           /* Topological eye value */
    int type;              /* HALF_EYE or FALSE_EYE */
    int num_attacks;       /* Number of attacking points */
    int attack_point[4];   /* The moves to attack a topological halfeye */
    int num_defends;       /* Number of defending points */
    int defense_point[4];  /* The moves to defend a topological halfeye */
};
```

结构数组 half_eye_data half_eye[MAX_BOARD] 包含关于后手眼和假眼的信息，如果类型是 HALF_EYE 则最多记录四个对眼的攻防着手，在极少的情形下攻击点与防守点不同。

7.5 棋块

结构数组 dragon_data dragon[MAX_BOARD] 集中棋块的信息。下面给出各域的定义，每个域对于棋块的各节点为常量。（下面讨论各域）

```
struct dragon_data {
    int color;           /* its color */
    int id;              /* the index into the dragon2 array */
    int origin;          /* the origin of the dragon. Two vertices
                          /* are in the same dragon iff they have
                          /* same origin.
    int size;            /* size of the dragon
    float effective_size; /* stones and surrounding spaces
    int crude_status;     /* (ALIVE, DEAD, UNKNOWN, CRITICAL)
    int status;           /* best trusted status
};
```

```
extern struct dragon_data dragon[BOARDMAX];
```

棋块的其他域包含在结构数组 dragon_data2 中（各域下面讨论）

```
struct dragon_data2 {
```

```

int origin;
int adjacent[MAX_NEIGHBOR_DRAGONS];
int neighbors;
int hostile_neighbors;
int moyo_size;
float moyo_territorial_value;
int safety;
float weakness;
float weakness_pre_owl;
int escape_route;
struct eyevalue genus;
int heye;
int lunch;
int surround_status;
int surround_size;
int semeai;
int semeai_margin_of_safety;
int semeai_defense_point;
int semeai_defense_certain;
int semeai_attack_point;
int semeai_attack_certain;
int owl_threat_status;
int owl_status;
int owl_attack_point;
int owl_attack_code;
int owl_attack_certain;
int owl_second_attack_point;
int owl_defense_point;
int owl_defense_code;
int owl_defense_certain;
int owl_second_defense_point;
int owl_attack_kworm;
int owl_defense_kworm;
};

```

```
extern struct dragon_data2 *dragon2;
```

两个数组不同之处是数组 `dragon` 由棋盘索引，且棋块中每个子都有棋块数据的副本，而 `dragon2` 数据只有一个副本。棋块数字编号，`id` 域是指向数组 `dragon2` 的键值。两个宏 `dragon` 和 `dragon2` 可获取两数组的访问权。

```

#define dragon2(pos) dragon2[dragon[pos].id]
#define dragon(d) dragon[dragon2[d].origin]

```

这样如果已知棋块中一个棋子的位置 `pos`，就可以直接访问数组 `dragon`，如用 `dragon[pos].origin` 访问棋头。但要使用数组 `dragon2` 的域，须使用 `dragon2` 宏，如访问其邻棋：

```
for (k = 0; k < dragon2(pos).neighbors; k++) {
    int d = dragon2(pos).adjacent[k];
    int apos = dragon2[d].origin;
    do_something(apos);
}
```

类似地，如果已知棋块编号（即 `dragon[pos].id`），可直接访问数组 `dragon2`，或用 `dragon` 宏访问数组 `dragon`。

以下是数组 `dragon` 的各域定义：

- `color`

对于棋串 `BLACK` 或 `WHITE`。对于空穴 `BLACK_BORDER`、`WHITE_BORDER` 或 `GRAY_BORDER`。意义与棋串相同。

- `id`

棋块编号，指向数组 `dragon2` 的键值。

- `origin`

棋块唯一的特定节点，用于确定两个节点是否同属于一个棋块。合并之前棋串的 `origins` 复制到棋块 `origins`。两个棋块合并共用同一 `origin`。

- `size`

棋块中棋子数

- `effective size`

成员棋串的 `effective sizes` 总和。回顾共享节点影响值是平分的，等于棋块子数加上更靠近它的空节点数。

- `crude_status`

(`ALIVE`, `DEAD`, `UNKNOWN`, `CRITICAL`)。对棋块活棋可能的早期估量。

在运行 `owl` 代码前计算，状态可用时立即替换之。

- `status`

棋块安全的最佳估量。运行 `owl` 代码后计算，运行对杀代码时修正。

以下是数组 `dragon2` 各域定义：

- `origin`

复制 `origin` 域。

- `adjacent`

`adjacent[MAX_NEIGHBOR_DRAGONS]`

相邻的棋块。由函数 `find_neighbor_dragons()` 计算。数组 `dragon2.adjacent` 给出棋块数。

- `neighbors`

相邻棋块数。

- `hostile_neighbors`

相邻的对方棋块数。

- `moyo_size`

`float moyo_territorial_value`

函数 `compute_surrounding_moyo_sizes()` 为每个棋块周围的 `e` 模样分配大小和实地值(参见 0 不能被战术攻击，或有战术防守并轮到先手的棋子称为战术活。类似地， 不能被战略攻击（死活分析的意义），或有战略防守并轮到先手的棋子称战略活棋。如果要在确定战略状态前使用影响函数，则战术活棋都视为战略活棋。

棋盘上所有战略活棋都作为影响源，其影响向各方向辐射。影响力随距离指数下降。

影响仅当棋盘空时无阻碍扩张，所有战术活棋（无论何方）都作为影响障碍，如不能穿越对方棋子之间的联络。如一间跳除非其中一子可被吃掉都作为障碍。注意两子之间的联络是否能被打穿并没有多少影响，因为两个方向上都有影响。

从双方的影响计算出每点介于 -1.0 和 $+1.0$ 之间的实地值，可视为各方

成实地的可能性。

为避免出现假实地，在可能的侵入点加入额外的影响源，如在座子下的 3-3 、在宽边缘扩展的中间和任何大的开放空间的中心。类似地在看起来是实地的但可能侵入的地方如小飞增加额外的影响源。这些侵入依赖于何方先手。

所有额外的影响源，和联络一样由模式数据库的两个文件 `patterns/influence.db` 和 `patterns/barriers.db` 控制。细节在

13.12 Patterns used by the Influence module 描述。

地、模样和区域)。这里是模样大小，记录在各域。

- `safety`

棋块安全取值如下：

`TACTICALLY_DEAD` - 识别代码发现的单棋串的死串（很可靠）

- `ALIVE` - `owl` 或对杀代码发现的活棋
- `STRONGLY_ALIVE` - 活棋疑问不多
- `INVINCIBLE` - 即使脱先若干手也绝对活棋
- `ALIVE_IN_SEKI` - 由对杀代码确定是对杀
- `CRITICAL` - 死活取决于先手
- `DEAD` - `owl` 代码发现的死串
- `INESSENTIAL` - 不重要（如点眼的子）的死串。

- `weakness`

- `weakness_pre_owl`

估量棋块安全的浮点数。棋块弱点为 0 到 1 之间，较大数字表示棋块有更多的安全诉求。域 `weakness_pre_owl` 是运行 `owl` 代码前一个初步的估计。

- `escape_route`

棋块就地不能成两眼时，对安全逃出的可能性的估量。文档在 13.9 逃

跑)。

- `struct eyevalue genus`

棋块期待的近似眼数，不保证精确。在引擎各处使用的 `eyevalue` 结构定

义如下：

```
struct eyevalue {
    unsigned char a; /* # of eyes if attacker plays twice */
    unsigned char b; /* # of eyes if attacker plays first */
    unsigned char c; /* # of eyes if defender plays first */
    unsigned char d; /* # of eyes if defender plays twice */
};
```

- `heye`

依附棋块的后手眼位置。

- `lunch`

如果非零，则是能被提掉的边界棋串。与被吃棋串相比，棋块须能防守。

- `surround_status`
- `surround_size`

了解棋是否被包围在估计棋块安全是很有用。该算法更多信息参见 13.11

包围和“`surround.c`”中的注释。在计算 `escape_route` 时使用，也可从模式调用（目前由 `CB258` 调用）。

- `semeai`
- `semeai_margin_of_safety`
- `semeai_defense_point`
- `semeai_defense_certain`
- `semeai_attack_point`
- `semeai_attack_certain`

如果两不同方棋块状态都为 `CRITICAL` 或 `DEAD` 则处于对杀状态，

由“`owl.c`”函数 `owl_analyze_semeai()` 试图确定其死活状态或双活，这

里先手至关重要。“`semeai.c`”中函数“`new_semeai()`”试图修正状态并根

据结果生成着手目标。如果棋块处于对杀中则域 `dragon2.semeai` 为非零。

对杀攻防点是双方赢取对杀的必争之地。域 `semeai_margin_of_safety` 目的是指示对杀是否结束，但目前尚未维护。域 `semeai_defense_certain` 和 `semeai_attack_certain` 指示对杀代码能在用完节点之前完成分析。

- `owl_status`

这是与 `dragon.crude_status` 类似的分类，基于“owl.c”中的死活识别。由特定启发算法判定安全的棋块略过 owl 代码（参见 12.1Owl 代码），如果未运行 owl 代码，则 owl 状态为 `UNCHECKED`。如果 `owl_attack()` 确定棋块无法被攻击，归类为 `ALIVE`，否则运行 `owl_defend()`，如果可防守归类为 `CRITICAL`，再否则为 `DEAD`。

- `owl_attack_point`

如果棋块可攻击，此即着手。

- `owl_attack_code`

owl 攻击代码，为 `WIN`、`KO_A`、`KO_B` 或（参见 12.1Owl 代码）。

- `owl_attack_certain`

owl 代码未用完节点前能够完成攻击分析。

- `owl_second_attack_point`

第二攻击点。

- `owl_defense_point`

如果棋块可防守，此即着手。

- `owl_defense_code`

owl 防守代码，为 `WIN`、`KO_A`、`KO_B` 或（参见 12.1Owl 代码）。

- `owl_defense_certain`

owl 代码未用完节点前能够完成防守分析。

- `owl_second_defense_point`

第二 owl 防守点。

7.6 彩色棋群显示

可以用彩色显示 ASCII 棋盘，每个棋块分配一个不同的字母，不同的 `matcher_status` 值 (ALIVE、DEAD、UNKNOWN、CRITICAL) 用不同的颜色表示，这使得，调试很好掌握。第二个图形显示 `owl.status` 值，如果是 UNCHECKED，棋块显示为白色。

用 CGoban 或 GNU Go 带 “-o” 选项按 SGF 格式存储棋局。

打开一个 `xterm` 或 `rxvt` 窗口，也可以使用 Linux 控制台。使用控制台需要用 “SHIFT-PAGE UP” 打开第一个图形。`Xterm` 必须编译为支持彩色方可工作——如果看不到彩色可以试用 `rxvt`。设置黑背景白前景。

执行：

```
gnugo -l [filename] -L [movenum] -T 获取彩色显示。
```

彩色方案：绿色 = ALIVE；黄色 = UNKNOWN；青色 = DEAD 红色 = CRITICAL。合并为同一棋块的棋串用同一字母标记。

其他有用的彩色显示可以这样获取：

- 选项 -E 显示眼位（参见 8 眼和后手眼）

选项 -m 0x0180 显示实地、模样和区域（参见 0 不能被战术攻击，或有战术防守并轮到先手的棋子称为战术活。类似地，不能被战略攻击（死活分析的意义），或有战略防守并轮到先手的棋子称战略活棋。如果要在确定战略状态前使用影响函数，则战术活棋都视为战略活棋。

棋盘上所有战略活棋都作为影响源，其影响向各方向辐射。影响力随距离指数下降。

影响仅当棋盘空时无阻碍扩张，所有战术活棋（无论何方）都作为影响障碍，如不能穿越对方棋子之间的联络。如一间跳除非其中一子可被吃掉都作为障碍。注意两子之间的联络是否能被打穿并没有多少影响，因为两个方向上都有影响。

从双方的影响计算出每点介于-1.0 和 +1.0 之间的实地值，可视为各方成实地的可能性。

为避免出现假实地，在可能的侵入点加入额外的影响源，如在座子下的 3-3 、在宽边缘扩展的中间和任何大的开放空间的中心。类似地在看起来是实地的但可能侵入的地方如小飞增加额外的影响源。这些侵入依赖于何方先手。

所有 额 外 的 影 响 源 ， 和 联 络 一 样 由 模 式 数 据 库 的 两 个 文 件 patterns/influence.db 和 patterns/barriers.db 控制。细节在 13.12 Patterns used by the Influence module 描述。

- 地、模样和区域)

彩色显示在其他地方也有文档（参见 5.8 彩色显示）。

8 眼和后手眼

本章描述 GNU Go 在确定眼时使用的算法。

8.1 错误!未找到引用源。

8.2 错误!未找到引用源。

8.3 错误!未找到引用源。

8.4 实例

8.5 图形

8.6 眼形分析

8.7 局部博弈值

8.8 后手眼和假眼拓扑

8.9 带劫的眼拓扑

8.10 错误!未找到引用源。

8.11 `optics.c` 中的函数

8.1 局部博弈

组合博弈理论的基本原理是博弈可以相加成一个群。对于博弈 G 和 H , 命 $G+H$ 为各方着子后形成的博弈, 称 $G+H$ 是局部博弈 G 和 H 的和。

以一个棋块中所有相连的眼位为一个局部博弈, 生成一局部博弈树使得该局部博弈的评分是它产生的眼数。一般地, 一方着子后, 最终评分会相差 0 或 1。此时该局部博弈可以表示为一个整数或者是半数。假定 $n(0)$ 和 $n(X)$ 分别是一方先走, 双方轮流落子 (不放弃) 的评分, 则博弈可表示为 $\{n(0) | n(X)\}$ 。这样 $\{1 | 1\}$ 即一眼, $\{2 | 1\}$ 是一眼加一后手眼, 依此类推。

特例的博弈 $\{2 | 0\}$ 尽管很少见, 但可能出现。我们将产生该博弈的眼位称为聚形 (CHIMERA)。局部博弈最终评分 2 或以上的棋块为活棋, $\{2 | 1\}$ 和 $\{3 | 1\}$ 没有差别, 故 $\{3 | 1\}$ 不视为聚形。

下面是聚形的一个例子:

XXXXXX

```
XOOOX
XO.OOX
XX..OX
XXOOXX
XXXXX.
```

8.2 眼位

为将所有眼位赋给棋块,需要将其周围的棋块合并(参见[7.2. 合并](#))。该操作由函数 `dragon_eye()` 完成。

一棋块的眼位是与其相邻的交点集合,这些交点可能成为眼位,且可以是未完全封闭的。如果一个开放的眼位足够大,就可能做成两眼。眼位边缘的交点(与眼位外空点相邻)称边缘(MARGINAL)。

以下例子:

```
| . X . X X . . X O X O
|X . . . . X X O O O
|O X X X X . . X O O O
|O O O O X . O X O O O
|. . . . O O O O X X O
|X O . X X X . . X O O
|X O O O O O O O X X O
|. X X O . O X O . . X
|X . . X . X X X X X X
|O X X O X . X O O X O
```

中央被包围的O棋块有开放眼位,该眼位中心是三个X死子,眼位足够大所以O不会被杀。我们抽取其中眼型如下:

```
| - X - X X - - X O X O
|X - - - - X X O O O
|O X X X X - - X O O O
|O O O O X - O X O O O
|! . . . O O O O X X O
|X O . X X X . ! X O O
|X O O O O O O O X X O
|- X X O - O X O - - X
|X - - X - X X X X X X
|O X X O X - X O O X O
```

所关注的形表示形式如下:

```
!...
.XXX.!
```

边缘点标记为!。眼位中被吃住的 X 棋子仍标记为 X。

确定眼位的精确算法有些复杂, `optics.c` 中函数 `make_domains()` 的源码注释中有文档说明。

运行 `gnugo -E` 可以用彩色 `ascii` 图形方便地显示眼位。

8.3 眼位的局部博弈

抽象的眼位包括一组标记的点:

! . X

数据库 `patterns/eyes.db` 有许多眼位表, 每个表都可以看作一个局部博弈。其结果在眼位后以如下形式列出: `max,min`, `max` 是 0 先走时模式产生的眼数, `min` 是 X 先走时模式产生的眼数。在此讨论时假定眼位属于 0。因为三眼比两眼并无更多价值, 故不尝试区分, 命 `max` 不大于 2, 且忽略 `min>1` 的模式。

如:

Pattern 548

x

xX.!

:0111

这里的标注与上述相同, x 表示 X 或空, 模式的结果不变。

按如下方式抽象局部棋局, 0 和 X 双方轮流着子或有一方放弃。

规则 1: 0 可以去除任何标记!或.的点。

规则 2: X 可以用 X 代替.。

规则 3: X 可以去除!。此时, 所有与被去除的!相邻的.变为!。如果被去除的!邻接一个 X, 则去除这个点及与其相邻的点。所有与被去除的 X 相邻的.则变为.。

这样如果 0 先走可将以上眼位例变换为:

... or !...
.XXX.!. .XXX.

如果 X 走, 可以去除!, 其相邻的.变为!。这样如果 X 先走可将眼位变换为:

!.. or !..
.XXX.!. .XXX!

注意：下面 X:1、O:2 交换后，有细微的差别。O 威胁吃掉三个 X 子，在 2 左面有一后手眼。这很微妙而抽象中还有捕获不到的细微，其中一些至少可以通过修正方案解决，但目前只使用简化的模型。

```
| - X - X X - - X O X O
|X - - - - - X X O O O
|O X X X X - - X O O O
|O O O O X - O X O O O
|1 2 . . O O O O X X O
|X O . X X X . 3 X O O
|X O O O O O O O X X O
|- X X O - O X O - - X
|X - - X - X X X X X X
|O X X O X - X O O X O
```

我们不尝试为局部博弈的终端状态定性（其中有些是双活）或评分。

8.4 实例

以下是无论哪方先手都产生一眼的局部博弈：

```
!
...
...!
```

以下是 O 先手时的变化。

```
!      (start position)
...
...!
```

```
...      (after "O"s move)
...!
```

```
...
..!
```

```
...
..
```

```
.X.      (nakade)
..
```

以下是另一变化:

! (after "X"'s move)

. .
..X!

. .
..X!

. !
..!

8.5 图形

考虑眼形相关的局部博弈只依赖于图形, 即一个基于棋盘点的集合, 其当且仅当两点在棋盘相邻时两元素相连。如以下两眼形:

..
..

和

....

不同棋形图形不同, 结果作为局部博弈也同形异构。这样可减少眼形数据库 `patterns/eyes.db` 的数量。在对待后手眼和假眼时做了进一步的简化。这类模式由拓扑分析 (参见 [8.8. 后手眼和假眼拓扑](#)) 识别。

后手眼与模式 (!.) 同形异构, 为此考虑以下两眼形:

X000000
X.....0
X000000

和

XX00000
X0a...0
Xb00000
XXXXXXX

是等价的眼形, 同形异构博弈 $\{2|1\}$, 前一棋形:

!....

后一眼形当 0 或 X 在 b 落子后, a 位有后手眼。这由拓扑准则发现 (参见 [8.8. 后手眼和假眼拓扑](#))

`eye_shape` 的图形，表面上“....”是在图形匹配时用!替代了左侧的.。假眼与模式(!)同形异构，为此考虑以下眼形：

```
XXX000000
X.Oa....O
XXX000000
```

等价于前两个眼形，同形异构局部棋局 {2|1}。

该眼位在 a 点有个假眼。也由拓扑准则发现。

`eye_shape` 的图形，表面上“.....”是在图形匹配时用!替代了左侧的.。这直接修正眼数据，并不限于图形匹配。

8.6 眼形分析

`patterns/eyes.db` 中的模式编译为图形，在 `patterns/eyes.c` 中以数组表示。

各实际眼位也编译为图形。后手眼按如下操作，参考上面讨论的例子

```
XX00000
X0a...O
Xb00000
XXXXXX
```

b 点作为边缘点加入眼位。图上邻接条件用宏表示 (`optics.c` 中)：如果两点相邻，或一个是后手眼另一个是其关键点，则这两点邻接。

在 `recognize_eyes()` 中，每个由实际眼位引发的图形都与 `eyes.c` 中的图形匹配，得到局部博弈的结果。如果匹配不成功则局部博弈设定为 {2|2}。

8.7 眼的局部博弈值

`eyes.db` 中的棋局价值以简化的机制给出，可灵活有效地表示大多数可见的眼形。

模式下面的分号行给出匹配眼形的眼价值，包括四个数字，分别是下列条件下取得的眼数：

1. 攻方先走对手脱先攻方再走一手。。
2. 攻方先走对手局部应一手。
3. 守方先走攻方局部应一手。

4. 守方先走攻方脱先守方再走一手。

第一种情形不一定意味允许攻方连走两手，下面例子解释。

另外，两眼即活，多于两眼也按两眼计算。

关注以下 15 种情形：

- 0000 0 眼。
- 0001 0 眼，但守方可以威胁成一眼。
- 0002 0 眼，但守方可以威胁成两眼。
- 0011 $1/2$ 眼，如果守方先走 1 眼，攻方先走 0 眼。
- 0012 $3/4$ 眼，守方先走 $3/2$ 眼，攻方先走 0 眼。
- 0022 1* 眼，守方先走 2 眼，攻方先走 0 眼。
- 0111 1 眼，攻方可威胁破眼。
- 0112 1 眼，攻方可威胁破眼，守方可威胁再成一眼。
- 0122 $5/4$ 眼，守方先走 2 眼，攻方先走 $1/2$ 眼。
- 0222 2 眼，攻方可威胁全破。
- 1111 1 眼。
- 1112 1 眼，守方可威胁再做一眼。
- 1122 $3/2$ 眼，守方先走 2 眼，攻方先走 1 眼。
- 1222 2 眼，攻方可威胁破一眼。
- 2222 2 眼。

$3/4$ 、 $5/4$ 和 1* 眼值与 Howard Landman 的论文围棋眼位价值一致。攻防点仅在其对于眼价值有确切效果时，才在模式中标注，不标记纯粹的威胁。

在这个文件的模式中可以找到所有不同情形的例子。其中一些略微与直觉相反，故这里解释一个重要的情形，考虑

Pattern 6141

X
XX.@x

:1122

例如匹配以下局面：

.OOOXXX
OOXOXOO

```
OXXba.O
OOOOOOO
```

现在看上去 X 在 a 落子接着下 b, 可破两眼, 故眼价值为 0122。这是第一个数字定义的精妙之处的体现。它并不是说允许攻方连走两手, 而是允许其走“另一手”。这个形的关键属性是当 X 在 a 着手破 (至少) 一眼, 0 可应在 b, 结果:

```
.OOOXXX
OO.OXOO
O.cOX.O
OOOOOOO
```

现在 X 必须继续着手 c 保证 0 只有一眼。此后即使 0 脱先 X 仍无法破另一眼。这样眼价值实际是 1122。

最后要注意的是一些眼价值允许依靠自杀的威胁, 如

```
Pattern 301
X.X
:1222
```

这个数据库总是允许自杀, 如果不允许自杀, 这样的着手很容易在上层滤掉。

8.8 后手眼和假眼拓扑

后手眼是成眼与否取决于哪方先走的眼模式, 如下例的 0:

```
OOXX
O.O.
OO.X
```

假眼是不能成眼的空, 如下两例的 0:

```
OOX      OOX
O.O      O.OO
XOO      OOX
```

在此描述发现后手眼和假眼的拓扑算法, 在本节忽略劫的可能性。

假眼和后手眼可以由眼位对角点的状态定性。不在眼位内的对角点有三种可能的状态:

- 对方 (X) 棋子且不会被吃。
- X 可安全落子, 或可攻可守的 X 棋子。
- 0 棋子、或可攻不可守的 X 棋子, 或 X 不能安全落子。

第一种情形取值 2，第二种取值 1，最后一种取值 0。将对角点值累加，按下述规则判定：

- $\text{sum} \geq 4$ ：假眼
- $\text{sum} = 3$ ：后手眼
- $\text{sum} \leq 2$ ：眼

如果眼位在边缘，以上值要减 2，另一个方法是给落在盘外的对角点赋值 1。为获取准确的等效还必须给角上的盘外对角点即纵横都出界的点赋值 0。

发现所有假眼和后手眼拓扑的算法：对于至多只有一个邻点同属眼位点的所有眼位点，根据上述规则计算对角点状态并根据累加值分类。

8.9 带劫的眼拓扑

本节在拓扑眼位分析中扩展劫争，这里将劫争分为 0 先劫和 X 先劫：

.?O? 0 提劫
OO.O
O.O?
XOX.
.X..

.?O? X 提劫
OO.O
OXO?
X.X.
.X..

首先假定前一棋形对角点值为 a 而后形为 b ，显然 $0 < a < 1 < b < 2$ 。命 e 为 e 拓扑眼值（同样是四个对角点值的累加），有如下特性：

$e \leq 2$ - 眼
 $2 < e < 3$ - 介于眼和后手眼之间
 $e = 3$ - 后手眼
 $3 < e < 4$ - 介于后手眼和假眼之间
 $e \geq 4$ - 假眼

为确定 a 和 b 分析含劫眼拓扑的典型情形：

.X.. (略) 优于眼
(a) ..OO $e < 2$
OO.O
O.OO $e = 1 + a$
XOX.

.X..

.X.. 后手眼和眼之间
(a'') ..00 $2 < e < 3$
00.0
0X00 $e = 1 + b$
X.X.
.X..

.X.. 后手眼和眼之间
(b) .X00 $2 < e < 3$
00.0
0.00 $e = 2 + a$
XOX.
.X..

.X.. 假眼和后手眼之间
(b'') .X00 $3 < e < 4$
00.0
0X00 $e = 2 + b$
X.X.
.X..

.X..
XOX. (略) 优于眼
(c) 0.00 $e < 2$
00.0
0.00 $e = 2a$
XOX.
.X..

.X..
XOX. 眼 some aji
(c'') 0.00 $e \sim 2$
00.0
0X00 $e = a + b$
X.X.
.X..

.X..
X.X. 后手眼和眼之间
(c'') 0X00 $2 < e < 3$
00.0
0X00 $e = 2b$

	X.X.	
	.X..	
	.X...	
	XOX..	后手眼和眼之间
(d)	O.O.X	$2 < e < 3$
	OO.O.	
	O.OO.	$e = 1 + 2a$
	XOX..	
	.X...	
	.X...	
	XOX..	后手眼, some aji
(d'')	O.O.X	$e \sim 3$
	OO.O.	
	OXOO.	$e = 1 + a + b$
	X.X..	
	.X...	
	.X...	
	X.X..	假眼和后手眼之间 (d''')
	OXO.X	$3 < e < 4$
	OO.O.	
	OXOO.	$e = 1 + 2b$
	X.X..	
	.X...	
	.X...	
	XOX..	假眼和后手眼之间
(e)	O.OXX	$3 < e < 4$
	OO.O.	
	O.OO.	$e = 2 + 2a$
	XOX..	
	.X...	
	.X...	
	XOX..	假眼, 有味道
(e'')	O.OXX	$e \sim 4$
	OO.O.	
	OXOO.	$e = 2 + a + b$
	X.X..	
	.X...	
	.X...	
	X.X..	(略) 劣于假眼

(e'') OXOX 4 < e
 OO.O.
 OXOO. e = 2 + 2b
 X.X..
 .X...

显然应当采用

(i) a=1/2, b=3/2

但这有一些缺点，可以采用以下之一解决

(ii) a=2/3, b=4/3

(iii) a=3/4, b=5/4

(iv) a=4/5, b=6/5

总结一下以上分析，有以下四种 a 和 b 的选择：

case	symbolic	a=1/2	a=2/3	a=3/4	a=4/5	desired
value	b=3/2	b=4/3	b=5/4	b=6/5	interval	
(a)	1+a	1.5	1.67	1.75	1.8	e < 2
(a'')	1+b	2.5	2.33	2.25	2.2	2 < e < 3
(b)	2+a	2.5	2.67	2.75	2.8	2 < e < 3
(b'')	2+b	3.5	3.33	3.25	3.2	3 < e < 4
(c)	2a	1	1.33	1.5	1.6	e < 2
(c'')	a+b	2	2	2	2	e ~ 2
(c''')	2b	3	2.67	2.5	2.4	2 < e < 3
(d)	1+2a	2	2.33	2.5	2.6	2 < e < 3
(d'')	1+a+b	3	3	3	3	e ~ 3
(d''')	1+2b	4	3.67	3.5	3.4	3 < e < 4
(e)	2+2a	3	3.33	3.5	3.6	3 < e < 4
(e'')	2+a+b	4	4	4	4	e ~ 4
(e''')	2+2b	5	4.67	4.5	4.4	4 < e

注意 (i) 不适用情形 (c'')、(d)、(d'') 和 (e)。另外三个选择均正确。它们的主要区别是 (c'') 和 (d) (或 (d'') 和 (e) 的排位)。进一步分析可以看到两种情形下”0”先走可保证无条件成眼而”X”先走可以劫争破眼。差别在于情形 (c'') 中”X”必须先开劫，而情形 (d) 0 必须先开劫，如 (c'') 对于 0 较优，其拓扑眼值应小于 (d)。这意味着 (iv) 是最佳选择。

根据以上分析可以看到 a、b 满足 $a+b=2$ 和 $3/4 < a < 1$ 即与 (iv) 等价。一个有趣的选择是 $a=7/8$ 、 $b=9/8$ 这样可以避免浮点误差。后一属性分属 $a=3/4$ 和 $a=1/2$ 。

同一眼位有三劫时更为复杂，不过很少见可以忽略。

8.10 假边缘

以下情形很少见但很特殊值得单独研究：

0000XX

0XaX..

这里 a 与 0 眼位相邻，又与不可攻击的 X 棋群相邻，但如果 X 在 a 着手真好是被叫吃。我们称 a 为假边缘。

眼代码中 0 的眼位应为 (X)，而非 (X!)。

8.11 optics.c 中的函数

公共函数 `make_domains()` 调用函数 `optics.c` 中的静态函数 `make_primary_domains()`，计算各方的影响区域，为确定眼位所用。注意：这里所用的影响区别于 `influence.c` 中的影响。

忽略不活的棋串之后，算法将本方的影响赋给：

- 本方活子；
- 与本方活子相邻的空点；
- 两相邻空点（不在一路上）在各方向（一般是 8 个）有本方影响，除去有两个下述的特例。

这样在下图中，如果 a 和 b，或 a 和 d 有本方影响，则 e 赋给本方影响。b”和 d 不相邻，所以本方影响条件不充分。

uabc

def

ghi

两相邻点不在 1 路上的约束是为防止三线棋子的影响漏洞。

第一个特例：即使 a 和 b 有本方影响，如果 d 有对方活子则 e 没有本方影响。这个约束防止了小飞的漏洞。

第二个特例：即使 a 和 b 有本方影响，如果 u 和 c 由对方活子，则 e 没有本方影响。这个约束防止二间关的漏洞。

角点的处理略有差别。

+---

|ab

|cd

如果 **b** 或 **c** 有本方影响且 **b**、**c** 或 **d** 没有对方活子则 **a** 有本方影响。

以下是 `optics.c` 中的公共函数，其中不包括由自动辅助函数调用的简单接口函数。静态声明的函数文档在源代码中。

- `void make_domains(struct eye_data b_eye[BOARDMAX], struct eye_data w_eye[BOARDMAX], int owl_call)`

该函数由 `make_dragons()` 和 `owl_determine_life()` 调用。标记黑白域（眼位区域）并收集一些状态。

- `void partition_eyespace(struct eye_data eye[BOARDMAX], int color)`

寻找连接的眼位部分并计算相关状态。

- `void propagate_eye(int origin, struct eye_data eye[BOARDMAX])`

将 `origin` 的数据复制到眼的其他点（仅非变量）。

- `int find_eye_dragons(int origin, struct eye_data eye[BOARDMAX], int eye_color, int dragons[], int max_dragons)`

寻找眼位周围的棋块。将最多 `max_dragons` 个棋块加入棋块数组，返回棋块数量。

- `void compute_eyes(int pos, int *max, int *min, int *attack_point, int *defense_point, struct eye_data eye[BOARDMAX], struct half_eye_data heye[BOARDMAX], int add_moves, int color)`

对于首位 `pos` 的眼位计算可产生的最小和最大眼数。如果 `max` 和 `min` 不等，则生成攻防关键点。

- `void compute_eyes_pessimistic(int pos, int *max, int *min, int *pessimistic_min, int *attack_point, int *defense_point, struct eye_data eye[BOARDMAX], struct half_eye_data heye[BOARDMAX])`

类似 `compute_eyes()`，给出做眼机会的悲观判断。

- `void add_false_eye(int pos, struct eye_data eye[BOARDMAX], struct half_eye_data heye[BOARDMAX])`

将眼位转换为边缘

- `int is_eye_space(int pos)`
- `int is_proper_eye_space(int pos)`

这些函数用作识别眼位的约束条件，主要用于终局着手。

- `int max_eye_value(int pos)`

返回眼位可产生的最大眼数。为确定眼位是否能产生实地很有用。

- `int is_marginal_eye_space(int pos)`

- `int is_halfeye(struct half_eye_data heye[BOARDMAX], int pos)`

- `int is_false_eye(struct half_eye_data heye[BOARDMAX], int pos)`

这些函数简单返回分析得到的关于眼位信息（不能真正工作）。

- `void find_half_and_false_eyes(int color, struct eye_data eye[BOARDMAX], struct half_eye_data heye[BOARDMAX], int find_mask[BOARDMAX])`

分析对角点寻找拓扑后手眼和假眼。

- `float topological_eye(int pos, int color, struct eye_data my_eye[BOARDMAX], struct half_eye_data heye[BOARDMAX])`

返回眼拓扑：

- 眼：2；
- 劫争破眼：2 和 3 之间；
- 后手眼：3；
- 劫争做眼：3 和 4 之间；
- 假眼：4 或以上

对角点控制攻防点存储在 `heye[]` 数组。`my_eye` 是眼位信息。

- `int obvious_false_eye(int pos, int color)`

与 `topological_eye()` 相关。使用基本相同的算法，只考虑对角点明显是安全对方棋串的情况。可确定假眼和后手眼，不会过高估计。

- `void set_eyevalvalue(struct eyevalvalue *e, int a, int b, int c, int d)`

（参见 [8.7. 眼的局部博弈值](#)）

```
struct eyevalvalue { /* number of eyes if: */
    unsigned char a; /* attacker plays first twice */
    unsigned char b; /* attacker plays first */
    unsigned char c; /* defender plays first */
    unsigned char d; /* defender plays first twice */
}
```

- `int min_eye_threat(struct eyevalvalue *e)`

攻防连走两手的眼数（第一着是威胁）。

- `int min_eyes(struct eyevalue *e)`

攻防先走轮流的眼数。

- `int max_eyes(struct eyevalue *e)`

守方先走轮流的眼数。

- `int max_eye_threat(struct eyevalue *e)`

守方连走两手的眼数（第一着是威胁）。

- `void add_eyevalues(struct eyevalue *e1, struct eyevalue *e2, struct eyevalue *sum)`

增加眼值 `*e1` 和 `*e2`，结果保存在 `*sum`。使 `sum` 为 `e1` 或 `e2` 较安全。

- `char * eyevalue_to_string(struct eyevalue *e)`

生成包含眼值的字符串。注意：结果字符串为静态存储，下次调用覆盖。

- `void test_eyeshape(int eyesize, int *eye_vertices) /* Test whether the optics code evaluates an eyeshape consistently. */`

- `int analyze_eyegraph(const char *coded_eyegraph, struct eyevalue *value, char *analyzed_eyegraph)`

分析眼图形确定眼值和关键着手。眼图形由字符串给出，编码%表示新行，

0 表示空，如：

```
!  
.X  
!...
```

编码为 `00!%0.X%!...`。（编码需要 GTP 接口函数）。结果是眼值和一个类似上例的非编码模式图并显示关键着手。例子的眼值为 `0112` 图形如下（显示劫争威胁）：

```
.X  
!.*.
```

如果眼图形不能识别，返回 0 否则返回 1。

9 模式代码

9.1 纵览

9.2 模式属性——分类

9.3 模式属性——值

9.4 辅助函数

9.5 自动辅助和条件

9.6 自动辅助操作

9.7 自动辅助函数

9.8 攻防数据库

9.9 联络数据库

9.10 联络函数

9.11 调整模式数据库

9.12 实现

9.13 对称和转换

9.14 实现细节

9.15“网格”优化

9.16 定式编译器

9.17 定式中的征子

9.18 角部匹配

9.19 编辑模式的 Emacs 模式

9.1 纵览

在 `patterns` 目录中有多个模式数据库。本章首先研究 `patterns.db`、`patterns2.db` 和 `hoshi.sgf` 编译得到(参见 9.16 定式编译器)的 `hoshi.db` 中的模式文件。这些文件根本上的区别, 仅在于 `patterns.db` 和 `patterns2.db` 是手写代码。它们合并到 `patterns.c` 后使用 `mkpat` 编译。将文件 `patterns2.db` 分列是考虑其更容易组织。`patterns.db` 组织不是很好, 正在缓慢地转向 `patterns2.db`。

在执行 `genmove()` 时, 模式在 `shapes.c` 中匹配查找着手目标。

同样的基本模式为 `attack.db`、`defense.db`、`conn.db`、`apats.db` 和 `dpats.db` 所使用, 这些模式用于不同的目的, 在其他部分研究。`eyes.db` 中的模式则与其他的完全不同(参见 8 眼和后手眼)。

模式数据库采用如下格式 `ascii` 表示:

```
Pattern EB112
?X?..?      jump under
  O.*oo
  O....
  o....
  -----

:8,ed,NULL
```

这里“O”表示本方棋子, “X”表示对方棋子, “.”表示空点, “*”表示本方下一手, “o”表示本方棋子或空点, “x”表示对方棋子或空点, “?”表示无关的点。

本例底行“-”表示棋盘边界——这是一个边上的模式, 也可以表示角上的模式。对于无关点不生成结果, 但并非完全忽略, 参见下面介绍。“:”开始的行描述模式属性, 如它的对称法则和类型。有时候还可提供一个 `helper` 函数辅

助匹配器确定是否接受着手。大多数模式不需要辅助函数，用 `NULL` 填充。

`matchpat.c` 中的匹配器查找棋盘上此类模式出现的地方，然后执行 `shapes.c` 回调函数 `shapes_callback()` 注册着手目标。

在模式后有一些如下格式的支持信息：

```
:trfno, classification, [values], helper_function
```

这里 `trfno` 表示模式需要考虑的变换数目，一般是 8（出于历史原因不考虑对称），或者 `"|"`、`"\"`、`"/"`、`"-"`、`"+"`、`"X"` 中的一个，分别表示对称方向（如 `"|"` 表示纵向对称）。

上面的模式可以等价地表示在左边：

```
|oOO?  
|...X  
|..*?  
|..o.  
|..o?
```

```
:8,ed,NULL
```

`mkpat` 程序能够识别这种格式的模式，并匹配到上边或右边，或者到角上。作为体例，`patterns.db` 中的模式都写在底行或者左下角。在 `patterns/` 目录中 `transpat` 程序可以将模式旋转和倒置。这个程序不是默认安装的——如果你需要可以将它 `make transpat` 到目录 `patterns/` 中并参考 `patterns/transpat.c` 开始的使用说明。

9.2 模式属性——分类

模式的属性行包括一个字符串，每个字符有不同的含义。这些属性可以分为两种：确定该模式是否匹配的条件属性和确定匹配后操作的操作属性。

9.2.1 条件模式属性

不检查着手安全性。适用于弃子模式。如果未给出该属性，匹配器要求该着手不可能被简单提掉。即使给出该属性，也对此着手进行合法性检查。

- n

除确认该着手不能被简单提掉，还检查对方在此着手不会被吃掉。

- O

检查模式中所有本方棋子所归属的棋块，其状态或为 ALIVE 或 UNKNOWN。

(参见 7.5 棋块)，不能是 CRITICAL。一串棋有可能在战术上处于危险，但仍然是 ALIVE 状态，因为它被合并到了一个 ALIVE 棋块，状态与后者一致。这样，如果模式中包含一串战术危险的棋，就需要执行一个额外的测试，如果空点“*”不能提供有效帮助，则抛弃该模式。

- o

检查模式中所有本方棋子所归属的棋快，其状态或为 DEAD 或 UNKNOWN。

- X

检查模式中所有对方棋子所归属的棋快，其状态或为 ALIVE、UNKNOWN 或 CRITICAL。注意这与本方棋子条件不同，后者抛弃 CRITICAL 棋块。

- x

检查模式中所有对方棋子所归属的棋快，其状态或为 DEAD 、UNKNOWN 或 CRITICAL。

9.2.2 操作属性

- C

如果模式中出现两个或两个以上本方棋块，着手目标设定为联络各棋块，例外的情况是这些棋块可被战术包围，并且着手没有能进行防守。

- c

对于所有本方棋块增加战略防守着手和小型评分。这个分类适用于弱联络模式。

- B

如果模式中出现两个或两个以上对方棋块，着手目标设定分断对方棋块。

- e

围实地。

- E

增强影响和建立/扩大模样。

- d

战略防守模式中本方除已被战术包围和无法战术防守的其棋块。如果本方棋块已经安全，此着手目标价值为 0。

- a

攻击模式中所有对方棋块。

- J

标准定式着手。除非棋盘上有紧要着手，否则立即采用本手。这与同时加 d 和 a 属性并置最小可接受价值为 27 等价。

- F

表示手筋模式。与 j 或 t 属性同时使用时有效，用于确认手筋中非确定着手。

- j

紧要性次于定式。着手在 J 属性着手之后选择。增加 e 和 E 属性。如果本手有 F 属性，即得到固定价值 20.1，否则取得最小可接受价值 20（这使得 GNU Go 可具有 jF 属性的着手间随机选择）。

- t

缓定式着手（可脱先）。等价于同时增加 e 和 E 属性，固定价值 15.07（如果着手有 F 属性）或最小价值 15（否则）。

- U

急定式着手（不可脱先）。等价于同时增加 d 和 a 属性，棋型评分 15 和

最小可接受价值 40。

一个较常用的类型为 $\circ X$ （抛弃任一方有死子的模式）。字符串“-”用作占位。（事实上所有上列字符和“,”以外的字符都会被忽略）。

属性 \circ 和 \bigcirc 可以在类别中出现，表示只应用于 UNKNOWN。 X 和 x 也同样可以共用。所有属性可任意组合。

9.3 模式属性——值

模式中“:”行接下去第二个以后的域是可选的，形式为 $value1(x)$ 、 $value2(y)$ 可能的值如下：

- $terri(x)$

强制着手地域值至少为 x 。

- $minterri(x)$

强制着手地域值至少为 x 。

- $maxterri(x)$

强制着手地域值至多为 x 。

- $value(x)$

强制着手最终价值至少为 x 。

- $minvalue(x)$, $maxvalue(x)$

强制着手最终价值至少/至多为 x 。

- $shape(x)$

着手棋型价值加 x 。

- $followup(x)$

着手逼应价值加 x 。

以上价值的意义参见 6 着手生成。

9.4 辅助函数

辅助函数用来辅助匹配器确定是否接受一个模式、注册着手目标和设定各种着手价值。辅助函数针对编译的模式入口表和“*”点的（绝对局面）提供。

一个困难在于辅助函数必须能够处理模式所有可能的变换。为此使用了 `OFFSET` 宏将模式相对坐标变换成绝对棋盘位置。

实际的辅助函数位于 `helpers.c`。声明在 `patterns.h`。

作为一个书写辅助函数的例子，我们考虑一个虚拟的辅助函数 `wedge_helper`，这样的辅助函数以前存在，现在为条件属性所取代。由于简单，它仍然是一个好例子。辅助函数开始有一段注释：

```
/*  
  
?O.          ?Ob  
.X*          aX*  
?O.          ?Oc  
  
:8,C,wedge_helper  
*/
```

左边是实际的模式，在右边我们复制了模式并增加了字母标签表示 `apos`、`bpos` 和 `cpos`。* 位置即模式采用之后通过参数 `move` 传递的 GNU Go 着手。

```
int wedge_helper(ARGS)  
{  
    int apos, bpos, cpos;  
    int other = OTHER_COLOR(color);  
    int success = 0;  
  
    apos = OFFSET(0, -2);  
    bpos = OFFSET(-1, 0);  
    cpos = OFFSET(1, 0);  
  
    if (TRYMOVE(move, color)) {  
        if (TRYMOVE(apos, other)) {  
            if (board[apos] == EMPTY || attack(apos, NULL))  
                success = 1;  
            else if (TRYMOVE(bpos, color)) {  
                if (!safe_move(cpos, other))
```

```

        success = 1;
        popgo();
    }
    popgo();
}
popgo();
}

return success;
}

```

OFFSET 行通知 GNU Go a、 b、 和 c 三子的位置。为确定模式是否保证一个联络，我们做一些识别。首先使用 TRYMOVE 宏在 move 落子，然后对方落子在 a 连回，接着调用 attack() 检查是否包围对方。a 处事先有子的情况没有必要检查，但实际还是做一下为好，因为落子前如果受攻棋子已经被包围 GNU Go 会产生保护错误。

如果攻击失败，我们在 b 落子并使用 safe_move() 函数检查 c 点的分断是否能被立即包围。在返回结果前我们要从识别堆栈中拿掉刚才的落子，这由函数 popgo() 完成。

9.5 自动辅助和条件

除了 helpers.c 中的手写辅助函数，GNU Go 可以自动按照图表的标签和条件表达式生成辅助函数。条件图表描述标签置于“:”命令行下面，条件表达式置于图表下面以“;”为起始的行上。条件只用来接受或抛弃给定模式。如果条件评估为 0 (false)，则该模式被抛弃，否则接受（当然仍取决于是否通过其他测试）。为给出一个简单的例子我们考虑一个联络模式：

```

Pattern Conn311

O*.
?XO

:8,C,NULL

```

```
O*a
?BO
```

```
;oplay_attack_either(*,a,a,B)
```

这里我们在所考虑的着手右边给出标签 `a`，在对方棋子处给出标签 `B`，另外，`*` 表示任何一方。标签可以是 `OoXxt` 以外的任何大写或小写 `ascii` 字母。作为体例我们使用大写字母表示对方着手，小写字母表示对方着手或空点。标签一串棋中棋子时，串中其他棋子也标签同样字母。（模式编译器并不强烈要求这个体例，但为数据库一致性和易读性考虑应该遵从）。

现在可以在条件表达式中使用标签。在本例中我们有一个识别条件，可解释为：“在`*`落子，接着在`a`对方落子，如果可包围`a`或/和`B`则接受”。

条件中使用的这个函数在 9.7 自动辅助函数中列出。技术上条件表达式由 `mcpat` 翻译成一个自动生成的辅助函数到 `patterns.c`。条件中的函数为 `C` 表达式所代替。原则上条件中可使用所有合法的 `C` 代码，但实际上除自动辅助函数以外没有理由使用布尔和算术以外的操作。条件可以分行最后合并。

9.6 自动辅助操作

作为只识别接受和抛弃的条件的补充，可以规定一个操作在模式通过所有测试最终被选中时执行。

例如：

```
Pattern EJ4
```

```
...*.      continuation
.OOX.
..XOX
.....
-----
```

```
:8,Ed,NULL
```

```
...*.      never play a here
.OOX.
```

```
.aXOX
.....
-----

>antisuji(a)
```

“>”开始行是操作行。在这里它通知着手生成器无论其他模式匹配了什么着手目标，a 处着手不能考虑。操作行使用条件图表中的标签。条件和操作可以用在同一模式下。如果操作只需要参考着手点，不需要条件图表。与条件一样，操作可以分行。

以下是可以从操作行调用的自动辅助宏的部分列表，列表并不完整。如果你在这个列表找不到操作行中使用的宏，可以参考 `mkpat.c` 查找引擎中实际调用的宏。如果没有你所需要的宏，你可以编辑 `mkpat.c` 中的列表增加之。

- `antisuji(a);`

标记 a 为禁止。

- `replace(a,*)`

如果着手*明显优于 a 点非常适用。该宏增加了一个落子重定义规则。着手生成器置为 a 的所有落子重定义为*。

- `prevent_attack_threat(a)`

因为对手在此落子将攻击 a，所以增加反向逼应价值。

- `threaten_to_save(a)`

因为落子解救 a 死棋，所以增加逼应价值。

- `defend_against_atari(a)`

估计防守可能被叫吃的串棋 a 的价值并加到反向逼应价值。

- `add_defend_both_move(a,b);`

- `add_cut_move(c,d);`

增加着手目标分断 c 和 d。

9.7 自动辅助函数

自动辅助函数由 `mkpat.c` 翻译成 C 代码，为检查函数是如何实现的，参考该文件中的自动辅助函数定义。自动辅助函数在条件和操作行都可以使用。

- `lib(x)`
- `lib2(x)`
- `lib3(x)`
- `lib4(x)`

数字 1、2、3、4 表示相应棋串的气数，参见 7 棋串和棋块。

- `xlib(x)`
- `olib(x)`

如果在空点 `x` 落子，对方和本方的气数。

- `xcut(x)`
- `ocut(x)`

调用 `cut_possible` (参见 18 工具函数) 确定对方或本方着手能分断空点 `x`。

- `ko(x)`

如果 `x` 有子或是劫位返回真值。

- `status(x)`

棋块的匹配状态，返回 `ALIVE`、`UNKNOWN`、`CRITICAL` 或 `DEAD` (参见 7 棋串和棋块)。

- `alive(x)`
- `unknown(x)`
- `critical(x)`
- `dead(x)`

如果棋块具有相关匹配属性返回真值，否则返回假值 (参见 7 棋串和棋块)。

`status(x)`

返回 `x` 处棋块状态 (参见 7 棋串和棋块) 。

- `genus(x)`

棋块的眼数。仅比较 0、 1 或 2 有效。

- `xarea(x)`

- `oarea(x)`

- `xmoyo(x)`

- `omoyo(x)`

- `xterri(x)`

- `oterri(x)`

调用 `whose_territory()`、 `whose_moyo()` 和 `whose_area()` (参见 0 不能被战术攻击, 或有战术防守并轮到先手的棋子称为战术活。类似地, 不能被战略攻击 (死活分析的意义), 或有战略防守并轮到先手的棋子称战略活棋。如果要在确定战略状态前使用影响函数, 则战术活棋都视为战略活棋。

棋盘上所有战略活棋都作为影响源, 其影响向各方向辐射。影响力随距离指数下降。

影响仅当棋盘空时无阻碍扩张, 所有战术活棋 (无论何方) 都作为影响障碍, 如不能穿越对方棋子之间的联络。如一间跳除非其中一子可被吃掉都作为障碍。注意两子之间的联络是否能被打穿并没有多少影响, 因为两个方向上都有影响。

从双方的影响计算出每点介于 -1.0 和 +1.0 之间的实地值, 可视为各方成实地的可能性。

为避免出现假实地, 在可能的侵入点加入额外的影响源, 如在座子下的 3-3 、在宽边缘扩展的中间和任何大的开放空间的中心。类似地在看起来是实地的但可能侵入的地方如小飞增加额外的影响源。这些侵入依赖于何方先手。

所有额外的影响源, 和联络一样由模式数据库的两个文件 `patterns/influence.db` 和 `patterns/barriers.db` 控制。细节在

13.12 Patterns used by the Influence module 描述。

地、模样和区域), 例如 `xarea(x)` 如果 `x` 是对方活子或对方区域内空点返回真值, `oarea(x)` 表示本方棋子和区域。区域、模样和实地的区别在于区域只产生影响, 模样有变为实地的现实可能, 实地则是信为安全的地域。

- `weak(x)`

棋块认定为弱棋时返回真值。

- `attack(x)`

- `defend(x)`

这是战术识别的结果, `attack(x) = true` 表示棋串可以被提掉, `defend(x) = true` 表示有防守着手。请注意对棋串没有攻着时 `defend(x)` 返回 `false`。

- `safe_xmove(x)`

- `safe_omove(x)`

分别在对方或本方棋子可以安全落在位置 “x” 时返回 `true`, 安全的概念是着手合法且不会立即被提掉。

- `legal_xmove(x)`

- `legal_omove(x)`

分别在对方或本方棋子在位置 `x` 上着手合法时返回 `true`。

- `o_somewhere(x,y,z, ...)`

- `x_somewhere(x,y,z, ...)`

分别表示 `O` (或 `X`) 在列表中的点上有子。在图上这些点标记为?。

- `odefend_against(x,y)`

- `xdefend_against(x,y)`

如果 `x` 上着子可以阻止对方在 `y`, 安全落子返回 `true`, 后者相反。

- `does_defend(x,y)`

- `does_attack(x,y)`

如果 x 上着手防守/攻击了 y 上的棋串返回 `true`，该结果尝试一手得到。

- `xplay_defend(a,b,c,...,z)`
- `oplay_defend(a,b,c,...,z)`
- `xplay_attack(a,b,c,...,z)`
- `oplay_attack(a,b,c,...,z)`

这些函数使得更复杂的条件识别实验得以进行，着手次序 a 、 b 、 c 按函数名指示的 x 或 o 开始双方交替执行，然后分别测试 z 处的棋串是否可被攻击或防守。轮到哪方走棋无关，任一方的棋串都可能被攻击和防守。对于攻着，受攻棋串的对方先行，对于守着，本方先行。如果该局面上棋串不会受攻或即使受攻也可防守则防守函数返回 `true`，如果有可能提掉棋串，无论是否可防守攻击函数都返回 `true`。如果着手列表走完后 z 处没有棋子，则假定攻击成功或防守失败。如果其中有非法着手，一般是因为自杀，就象什么也没发生一样继续走下一手，并测试攻防。这种方式在不需许多特例就可给出适当的结论。

等着用特殊的标签?表示，`oplay_defend(a,?,b,c)` 在 a 和 b 连续着手 o ，模拟 x 第二手在棋盘其他地方落子，并请求检测 c 是否可防守。等着不能作为次序中的第一着，也没有必要：象 `oplay_defend(?,a,b,c)` 可以用 `xplay_defend(a,b,c)` 代替。

- `xplay_defend_both(a,b,c,...,y,z)`
- `oplay_defend_both(a,b,c,...,y,z)`
- `xplay_attack_either(a,b,c,...,y,z)`
- `oplay_attack_either(a,b,c,...,y,z)`

这些函数与上面的类似，区别是最后两个参数表示棋串同时被攻击和防守，显然 y 和 z 必须是同方的。如果任一位置为空，则假定攻击成功或防守失败。这些函数一般用在分断模式中确认提掉分断棋子。

函数 `xplay_defend_both` 从 a 着手 x 开始交替落子，然后将最后两个

参数传递给 `engine/utils.c` 中的 `defend_both`，该函数检查并确定这两个棋串是否可同时防守。

函数 `xplay_attack_either` 从 `a` 着手 `x` 开始交替落子，然后将最后两个参数传递给 `engine/utils.c` 中的 `attack_either`。该函数寻找一个着手至少可以提掉两个棋串其中之一。在目前的实现中，`attack_either` 只寻找不同的攻着所以可能错过双打。

- `xplay_connect(a,b,c,...,y,z)`
- `oplay_connect(a,b,c,...,y,z)`
- `xplay_disconnect(a,b,c,...,y,z)`
- `oplay_disconnect(a,b,c,...,y,z)`

函数 `xplay_connect(a,b,c,...,y,z)` 开始在 `a` 着手 `x`，`b` 着手 `o`，这样继续，最后调用 `worm_connect()` 确定 `x` 和 `y` 能否连接。其他函数类似（参见 0 识别代码寻找一个通过着手树确定是否能吃住一棋串的途径。检查工具是选项“`--decideworm`”，可带或不带输出文件。

简单地执行

```
gnugo -t -l [input file name] -L [movenumber] --decideworm [location]
```

即运行 `attack()` 确定是否能吃住棋串，同时运行 `find_defense()` 确定是否能防守。给出识别的变化数。“`-t`”是必须的，否则 GNU Go 不报告找到的结果。

如果增加“`-o output file`”GNU Go 生成一个输出文件包含考虑的所有变化。变化在注释中编号。

没有浏览代码的方式时变化文件没有太大作用。它由 GDB 源文件生成列在最后。可以从 GDB 生成，或作为 GDB 初始文件。

如果使用 GDB 调试 GNU Go，可以发现不优化编译更不容易产生混淆。优化有时会改变程序的执行顺序。如不优化编译“`reading.c`”，编

编辑“engine/Makefile” 从文件中删掉字符串 -O2 联编“engine/reading.c”。注意 Makefile 是自动生成的，接着会被覆盖。

如果在识别过程中需要分析一个结果，而函数是从 hash 代码缓存局面中取值，用命令行“--hash 0”关闭 hash 重新运行实例，应当得到相同的结果。除非有更好的理由，否则不要运行“--hash 0”，因为那样会增加变化数。

装入后面给出的源文件，可以浏览变化。在 cgoban 中使用小“-fontHeight”可使变化窗口显示更多。（可以改变棋盘尺寸）。

假设检查文件后发现变化 17 有趣并想看到确切的过程。

宏“jt n”可跳转到第 n 个变化。

```
(gdb) set args -l [filename] -L [move number] --decidestring [location]
(gdb) tbreak main
(gdb) run
(gdb) jt 17
```

跳转到问题位置。

实际上攻防变化是分别编号的。（但 find_defense() 仅当 attack() 后运行，所以防守变化也许存在也许不存在。）每次都 tbreak main 有些多余，可以用两个宏 avar 和 dvar。

```
(gdb) avar 17
```

重启程序，跳转到第 17 攻击变化。

```
(gdb) dvar 17
```

跳转到第 17 防守变化。两个变化可在同一 sgf 文件中找到，但分别编号。

文件中定义的其它命令如下：

dump 打印着手堆栈。

nv 移动到下一变化

ascii i j 转换(i,j)到 ascii

```
#####
#####      .gdbinit file      #####
#####

# this command displays the stack

define dump
set dump_stack()
end

# display the name of the move in ascii

define ascii
set gprintf("%o%m\n", $arg0, $arg1)
end

# display the all information about a dragon

define dragon
set ascii_report_dragon("$arg0")
end

define worm
set ascii_report_worm("$arg0")
end

# move to the next variation

define nv
tbreak trymove
continue
finish
next
end

# move forward to a particular variation

define jt
while (count_variations < $arg0)
nv
end
nv
dump
end
```

```

# restart, jump to a particular attack variation

define avar
delete
tbreak sgffile_decidestring
run
tbreak attack
continue
jt $arg0
end

# restart, jump to a particular defense variation

define dvar
delete
tbreak sgffile_decidestring
run
tbreak attack
continue
finish
next 3
jt $arg0
end

```

联络识别)。

- `xplay_break_through(a,b,c,...,x,y,z)`
- `oplay_break_through(a,b,c,...,x,y,z)`

这些函数用来设定类似下面的局面：

```

.O.   .y.
OXO   xXz

```

X 目标是至少提掉 x、 y 和 z 其中之一。如果成功则返回 1，如果未成功

X 尝试分断任一边，如果这样成功则返回 2，当然同样形态可以用于对方。

重要提示： x、 y 和 z 必须按上图或其反射和旋转后的顺序给出。

- `seki_helper(x)`

检查 x 处棋串是否可攻击周围棋串。如果是，表示着手做出的双活无效，

返回 false。

- `threaten_to_save(x)`

调用 `add_followup_value` 作为着手目标增加提掉对方棋串救活 `x` 处棋串的保守估值。

- `area_stone(x)`

返回 `x` 周围的棋子数量。

- `area_space(x)`

返回 `x` 周围的空数量。

- `eye(x)`
- `proper_eye(x)`
- `marginal_eye(x)`

如果 `x` 分别是任一方的眼位、非盘边眼位或盘边眼位时返回 `true`。

- `antisuji(x)`

通知着手生成器 `x` 是低于标准的着手不必尝试。

- `same_dragon(x,y)`
- `same_worm(x,y)`

如果 `x` 和 `y` 分别属于相同棋块或棋串返回 `true`。

- `dragonsize(x)`
- `wormsize(x)`

指明棋块或棋串的棋子数量。

- `add_connect_move(x,y)`
- `add_cut_move(x,y)`
- `add_attack_either_move(x,y)`
- `add_defend_both_move(x,y)`

确切通知着手生成器模式中的着手目标。

- `halfeye(x)`

如果“`x`”空点是后手眼返回 `true`。

- `remove_attack(x)`

通知战术识别猜测的攻击事实无效。

- `potential_cutstone(x)`

如果棋串数据中的 `cutstone2` 域大于一返回 `true`。表明救活该棋串至少产生两个新断点。

- `not_lunch(x,y)`

阻止 `x` 被 `y` 吃掉的错误报告。例如下面模式通知 GNU Go 即使 `a` 上棋子能被提掉，也不归 `b` 棋串，因为提子没有成眼：

```
XO|      ba|
O*|      O*|
oo|      oo|
?o|      ?o|
```

```
> not_lunch(a,b)
```

- `vital_chain(x)`

调用 `vital_chain` 确定提掉 `x` 处棋子是否为相邻棋块成眼。目前的实现只检查棋子在首行是否是独子。

- `amalgamate(x,y)`

连接 `x` 和 `y` 处棋串（参见 7 棋串和棋块）。

- `amalgamate_most_valuable(x,y,z)`

当 `x`、`y`、`z` 指向三个（最好是不同的）棋块时调用，如：

```
.O.X
X*OX
.O.X
```

在此情形下，对方在 `*` 落子可防止三棋块连接。`O` 可决定允许哪个分断。这个辅助函数在 `y` 连接 `x` 或 `z` 中最大的棋块。

- `make_proper_eye(x)`

这个自动辅助函数当“`x`”是眼位但被误识为空的时候调用，重新归类为正常眼位（参见 8.2 眼位）。

- `remove_halfeye(x)`

从眼位中删除后手眼。这个辅助函数在 `make_dragons` 执行后不能再运行，因为此时眼位已经分析完毕。

- `remove_eyepoint(x)`

删掉一个眼点。这个函数只能在眼位分割前使用。

- `owl_topological_eye(x,y)`

这里 `x` 是一个空点，是一些棋块的眼或后手眼，`y` 是棋块中的一个棋子，只用来确定眼位的归属方。返回 `x` 尖点的值的合计，见 8.8 后手眼和假眼拓扑。如果 `x` 处眼位是假眼返回不小于 4，后手眼返回 3，真眼返回 2。

- `owl_escape_value(x)`

返回 `x` 处的逃跑值。只用于 owl 攻防模式。

9.8 攻防数据库

`attack.db` 和 `defense.db` 用以辅助战术识别，寻找攻防棋串的着手。

在 `make_worms()` 中所有棋串的战术状态确定的情况下调用。这些数据库不使用以上所述的分类，而分为两类。

- D

对于模式中所有可被战术提掉的 O 棋串 (`worm[m][n].attack_code != 0`)，尝试着手*，如果是防守有效，注册其目标，并将棋串的防守点指向*。

- A

对于模式中所有 X 棋串，测试*着手是否可以提掉棋串。如果是，注册其目标，棋串的攻击点指向*，如果不是，且以前未被攻击，则寻找防守着手。

进一步，模式 A 仅在 `attack.db` 中使用，模式 D 仅在 `defense.db` 中使用，这些数据库里可能出现未分类的模式，使用时必须依靠操作属性使其有效。

9.9 联络数据库

`conn.db` 中的模式用于帮助 `make_dragons()` 将棋串合并为棋块，并在一定程度修改眼位。这个数据库中的模式使用分类 B、C 和 e。模式 B 用来寻找无法合并的分断点，模式 C 用来寻找可以合并的已有连接，模式 e 用来修改眼位和重估死子。也有一些未分类的模式，使用操作属性增补，并与模式 C 一同匹配。详情和例子可以在 7 棋串和棋块找到。

举例演示这些数据库，如下情形： -

```
XOO
O.O
```

X 无法在断点安全着子，故 O 棋块可以合并，这里匹配了两个模式：

```
Pattern CC204
```

```
O
.
O
```

```
:+,C
```

```
O
A
O
```

```
;!safe_xmove(A) && !ko(A) && !xcut(A)
```

```
Pattern CC205
```

```
XO
O.
```

```
:\,C
```

```
AO
OB
```

```
;attack(A) || (!safe_xmove(B) && !ko(B) && !xcut(B))
```

条件多很清楚，如第二个模式中如果 X 棋子不会受攻且 X 可以安全在 B 着子或 B 是劫，则不能匹配。条件 `!xcut(B)` 表示联络在此之前没有被 `find_cuts` 禁止。例如以下情形：

```
OOXX
O.OX
X..O
X.OO
```

对于前一个模式匹配了两次，X 可以放入并分断了一个联络，为修补它包含一个模式：

```
Pattern CB11

?OX?
O!OX
?*!O
??O?

:8,B

?OA?
OaOB
?*bO
??O?

; !attack(A) && !attack(B) && !xplay_attack(*,a,b,*) && !xplay_attack(*,b,a,*)
```

发现这个模式后，自动辅助宏 `xcut` 在 *、a 和 b 点返回 `true`，故模式 CB204 和 CB205 不匹配，棋块不能合并。

9.10 联络函数

以下是“connections.c”公共函数

- `static void cut_connect_callback(int m, int n, int color, struct pattern *pattern, int ll, void *data)`

在 (m,n) 尝试所有联络模式。对于所有的匹配，如果是模式 B，在棋串数据结构中设置断点，并对模式的联络禁止入口做眼，如果是模式“C”将模式中

的棋块合并。

- `void find_cuts(void)`

寻找可禁止联络的断点并完成相邻的眼位。遍历联络数据库中的类型 B。

找到这样一个函数后，启动函数 `cut_connect_callback`。

- `void find_connections(void)`

寻找确切的联络模式并合并相应棋块。遍历联络数据库中除类型 B、E 或 e 以外模式。找到这样一个函数后，启动函数 `cut_connect_callback`。

- `void modify_eye_spaces1(void)`

寻找确切的联络模式并合并相应棋块。遍历联络数据库中的类型 E（参见 9.9 联络数据库）。找到这样一个函数后，启动函数 `cut_connect_callback`。

- `void modify_eye_spaces1(void)`

寻找确切的联络模式并合并相应棋块。遍历联络数据库中的类型 e（参见 9.9 联络数据库）。找到这样一个函数后，启动函数 `cut_connect_callback`。

9.11 调整模式数据库

由于模式数据库和着手目标的评估决定了 GNU Go 的个性，故花费了很多时间去：“调整”它，以下是一些建议：

如果想实验修改模式数据库，用 `-a` 选项启动。这将重新评估所有模式，即使有些因各种优化可能被略过。

至少有两种途径可以获取 SGF 格式的棋谱：一是使用 CGoban 记录棋局，一是使用 `-o` 选项让 GNU Go 记录棋局，最好结合 `-a`。不要在对局结束并关闭棋局窗口前读取 SGF 文件，当然这并不意味着一定要下完，棋局中关闭 CGoban

窗口 GNU Go 就会关闭 SGF 使得你可以读取。

如果使用 `-o` 选项记录 SGF，GNU Go 会在棋盘上加入标签显示所有考虑的着手及其估值。这是一个极其有用的特性，因为可以一眼看到着手生成器是否以合理的权重推荐了正确的着手。

首先，由于未知的错误，GNU Go 偶尔可能收不到 CGoban 表示结束的 SIGTERM 信号，此时 SGF 文件结尾没有结束的括号，CGoban 就打不开文件，可以在命令行键入以下命令：

```
echo ")" >>[filename]
```

加入结束括号。或者可以用编辑器加。

CGoban 能够显示超过 99 的着手价值（很少见），但必须将窗口放大才能看见三位数字。拖拉 CGoban 窗口右下方空白处当窗口变大时放开就可以看到三个数字。

如果没有使用 `-o` 选项对局并希望分析一个着手时，仍可以使用 CGoban 的“Save Game”按钮得到一个 SGF 文件，当然其中没有标注着手价值。

一旦有了 SGF 格式的棋谱，就可以分析任何一个着手，运行：

```
gnugo -l [filename] -L [move number] -t -a -w
```

可以了解为什么 GNU Go 做出这个着手，如果对模式数据库做了修改并重新编译了程序，可以请求 GNU Go 重复着手看行动是怎么改变的。如果使用 emacs，建议用缓冲中的 shell (`M-x shell`) 运行 GNU Go，这样翻阅和查找功能较好。

除了着手序号，还可以用 `-L` 给出着手坐标停在第一次这个位置上的着手，如果没有 `-L` 选项，考虑该着手以后着手。

如果推荐了一手坏棋，可能有几种原因。首先，程序应按棋盘上的实际点对每个着手尽可能精确地进行评估，如果不是，那就出错了。这可能有两种原因，一种可能是着手的目标缺失或找到了错误的目标。另一种可能是着手评估

函数估值错误。调整模式很少是因为第一种问题以外的情况。

如果找到了错误的目标，检查该模式负责的跟踪输出（一些着手目标，如大多数战术攻防不是来自模式的，如果错误目标不是模式生成的，就不是调整问题）。这个模式可能太普遍或有错误的条件，试着让他更明确，如果有错误就修正它。如果模式和条件都看起来正常，验证战术识别是否正确评估了条件，如果不是，那或者是识别错误或者识别对于 GNU Go 过于复杂。

如果发现了联络着手目标，但棋串已经有效联络了，可能是 `conn.db` 有缺少的模式，同样，`conn.db` 过于普遍或错误的模式可能会将棋串错误地合并。

为获取 `conn.db` 模式匹配的跟踪过程，可以加入第二个 `-t` 选项。

如果着手目标缺失，数据库中可能有漏洞。已有的模式不必要的限定、错误的条件或者由于识别错误被丢弃都有可能导致这类错误。除非熟悉模式数据库，否则很难验证却有模式缺失。浏览数据库试图获得一个感觉，它们是如何组织的（这是模式数据库的一个弱点，现在的目标是让他们组织更合理）。如果确认需要一个新模式，是这使他尽可能通用，从 `snOoXx` 和条件使用正确的分类使其不允许错误的匹配。识别函数可以有很好的用途。使模式尽可能通用的理由是我们需要其数量更少，便于维护。当然如果需要太复杂的条件，一般最好是把模式分开。

如果着手目标正确但评估仍然错误，这一般不是一个调整问题，但有可能与模式的使用相关。特别地，如果 `delta_terri()` 给出了奇怪的结果势力值就会超出范围，模式工作时就可能赋给其最小或最大值。这一般出现在终局模式中，这时经常会遇到最小值。如果需要，最小值和最大值都会用到。这些可能性使用应保守，因为这样的模式在开发出更好的（或者至少不同的）势力估计之类的函数后就会被弃用。

为在相同目标的着手中进行选择，如联络两棋块的不同方法，需要模式带有一个非 0 的形态值，为好形态或好味道赋给正值，为坏形态和坏味道赋给负值。注意这些值是可以累加的，模式并非唯一，所以这很重要。

模式跟着的值指明了着手价值，这一般不需要非常精确，而一个很好的原

则就是要相当保守。一般这应当根据棋盘的实际点来衡量。这些值同样是可累加的，同样需要小心避免非预期的多个匹配。

可以使用 `-T` 选项查看棋块的直观显示。GNU Go 默认配置构造的是使用光标 `curses` 或 `ansi` 转义序列的彩色支持。至少在很多系统中，彩色支持在 `rxvt` 下比在 `xterm` 下更为常见，因此我们建议在 `rxvt` 窗口运行：

```
gnugo -l [filename] -L [move number] -T
```

如果看不到彩色显示，且你的主机是 GNU/Linux 机器，可在 Linux 控制台上再试。

属于同一棋块的棋串用相同的字母标记，颜色表示 `moyo.c` 设置的域 `dragon.safety` 的值。

绿色：GNU Go 认为棋块活棋

黄色：未知状态

蓝色：GNU Go 认为棋块死棋

红色：危险状态（1.5 眼）或“`moyo.c`”算法判定弱棋

如果需要让棋局一遍遍再现，可以用 `-r` 选项提供一个固定的随机数种子以去除 GNU Go 对弈的随机性。

9.12 实现

GNU Go 中的模式代码实现概念非常直接，但因为匹配器在选择着手时消耗了大部分时间，所以代码为速度做了优化，为此实现的细节略有些难解。

在 GNU Go 中，用一个独立的程序 `mkpat.c` 将 `ascii` 编码的 `.db` 文件预编译到表（见 `patterns.h`），结果 `.c` 文件再编译并连接为 `gnugo` 可执行文件。

每个模式都编译成一个标题、一串元素，并在各个局面和棋盘的各个方向连续检查。这些元素与“锚”（或初始）模式相关联。用一个 `x` 或 `o` 棋子代表模式的开始。（我们不能说一个或另一个，因为有些模式只包含一方棋子）。所有元素都按坐标关联到位置。所以一个模式在棋盘位置 (m, n, o) 匹配意即该模式

的锚子在 (m, n) 且当该模式按变换号 \circ 变换后其它棋子得到匹配(参见下面变换的细节, 尽管这并不是必须的)

9.13 对称和转换

一般地, 每个, 模式要在 8 个不同的方位尝试, 反映棋盘的对称。但有些模式本身是对称的而不必尝试所有 8 种匹配。“:”后第一个字符可以是“8”、“|”、“\”、“/”、“X”、“-”、“+”的其中之一, 代表对称的方向。也可以是“O”表示 180 旋转对称。

transformation	I	-		.	\	l	r	/
	ABC	GHI	CBA	IHG	ADG	CFI	GDA	IFC
	DEF	DEF	FED	FED	BEH	BEH	HEB	HEB
	GHI	ABC	IHG	CBA	CFI	ADG	IFC	GDA
	a	b	c	d	e	f	g	h

则在下列对称形式下模式转换:


| c=a, d=b, g=e, h=f
- b=a, c=d, e=f, g=h
\ e=a, g=b, f=c, h=d
/ h=a, f=b, g=c, e=d
O a=d, b=c, e=h, f=g
X a=d=e=h, b=c=f=g
+ a=b=c=d, e=f=g=h


我们可以选择变换 a、d、f、g 作为模式在“|”、“-”、“\”或“/”对称的唯一变换。


这样我们选择定制变化 a、g、d、f、h、b、e、c 并选择前 2 个为“X”和“+”, 前四个为“|”、“-”、“/”和“\”中间 4 个为“O”所有 8 个为不对称模式。

每个反射操作符(e-h) 等同一个方位外加一个旋转 (a-d) , 我们可以选择对称方位(没有网络变化)并得出结合每个 e-h 等同反射外加 a-d 。这些参数扩展包含了“-”和“/”以及“|”和“\”。


9.14 实现细节


 模式标题中的一个入口表示锚是 x 还是 o。这辅助了性能，因为假如锚子没有匹配的话所有变换都可以抛弃。（理想地，我们可以简单定义锚永远是 o-或永远是 x，但有些模式不包含 o 而有些不包含 x。）

 模式标题包含模式相对于锚的大小（即左上和右下元素的坐标）。这使得模式可以在给定的方位缺少足够可供匹配的空间时快速被抛弃（即太靠近盘边）。边界信息在测试前必须象元素那样经过变换，变换后也需要计算出左上和右下角的位置。

 边界条件的实现是靠用 ? 或列在宽高两个方向填充到 19（或当前棋盘尺寸）。然后如果不在边界，模式就很快被抛弃。故上例模式编译成：

```
"example"
.OO????????????????
*XX????????????????
o????????????????
:8,80
```

 模式中的元素排序市的非空元素先于空元素检查，在棋局大部分时间，棋盘更多的点是空的，所以这样做模式可以更快地被抛弃。

 实际的测试是采用 and-compare 顺序执行的。每个棋盘位置是一个 2 位属性：%00 表示空，%01 表示 o， %10 表示 x。我们可以用 %11 与测试确定的模式（无操作），然后与 0、 1 或 2 比较。测试 o 与测试 not-x 相同，即 not %10，所以与 %01 匹配后得到 0，类似地可以使 bit 0 不置位测试 x。

9.15 “网格”优化

模式和棋盘之间的比较以 2 位位操作执行，这样在 32 位计算机上可以同时并行执行 16 个操作。

假定棋盘形态如下：

```
.X.O....OO
XXXXO.....
.X..OOOOOO
X.X.....
....X...O.
```

内部存储为（二进制）二维数组。

```
00 10 00 01 00 00 00 00 01 01
10 10 10 10 01 00 00 00 00 00
00 10 00 00 01 01 01 01 01 01
10 00 10 00 00 00 00 00 00 00
00 00 00 00 10 00 00 00 01 00
```

将其编译成一个复合数组每个元素存储 4×4 网格的状态：

```
???????? ???? ???? ???? ...
??001000 00100001 10000100
??101010 10101010 10101001
??001000 00100000 10000001

??001000 00100001 ...
??101010 10101010
??001000 00100000
??001000 10001000

...

??100010 ...
??000000
????????
????????
```

“??”是盘外。

将这些 32 位复合体存储到一个二位合并棋盘数组，用%11 代替??。

类似地，对于所有模块，mkpat 为靠近锚的模式元素生成适当的 32 位 and-值掩码，这样与上述相同的测试就简单化了，同时可执行 32 位。

9.16 定式编译器

GNU Go 在 patterns/joseki.c 包含了一个定式编译器。处理 SGF 文件

（和变体）并生成一系列可馈送给 `mkpat` 的模式。定式数据库目前在 `patterns/` 中文文件 `hoshi.sgf`、`komoku.sgf`、`sansan.sgf`、`mokuhazushi.sgf` 和 `takamoku.sgf`。这种分划必要时可以修订。

SGF 文件用 `joseki.c` 中的程序变换为模式数据库 `.db` 格式。这些文件接着用 `mkpat.c` 中程序变换为 C 代码，C 文件再编译连接进 GNU Go 二进制文件。

并不是 SGF 中所有节点都贡献到模式，贡献到模式的是在右上角有定式的节点，边界用方块标记，其他信息确定在注解中标记的结果模式。

意图是着手评估应当有能力用正常的估值在可能的变着中进行选择。当这失败时就接着用形态值增减，也可以增加反手筋变着以禁止普通低于标准的着手。通常的条件也可以使用，例如在工作梯级定义一个条件变着。

定式格式对于每个 SGF 节点格式如下：

- 一个方块标记，（SQ 或 MA 属性）决定模式应包含棋盘的多大部分。
- 一个着手（W 或 B 属性）和自然解释。如果没有方块标记或着手是 `pass`，则不为该节点生成模式。
- 选项标签（LB 属性），每个必须是单个字母。如果至少有一个标签，则用这些标签生成条件图。
- 一个注解（C 属性）作为首字母应为一下字母中一个，已确定类型：
 - U - 紧急
 - S 或 J - 标准着手
 - S 或 j - 少见定式
 - T - 欺着
 - t - 最小定式着手（可脱先）
 - 0 - 反手筋（也用“A”）

行中除此之外其他的都忽略掉，如果没找到则假定是标准定式着手。

此外，识别以下字母开始的行：

- # – 注释。复制到模式文件，在图形上方。
- ; – 条件。复制到模式文件，在条件图形下方。
- > – 操作。复制到模式文件，在条件图形下方。
- : – 列线。这有点复杂，因为生成模式的列线总以:8,s 开头作为变换号和牺牲模式类（一般不是牺牲而是无目的地花费时间检查战术安全）。增加一个定式模式类字符最后包含在 SGF 节点的注释中。

例如：在 SGF 文件的注释中：

```
F
:C,shape(3)
;xplay_attack(A,B,C,D,*)
```

生成的模式有列线：

```
:8,sjC,shape(3)
```

条件：

```
;xplay_attack(A,B,C,D,*)
```

9.17 定式中的征子

作为如何在定式编译器使用自动辅助函数的例子，考虑征子失败的定式错误。假定走大斜并考虑模式外围：

```
-----+
.....|
.....|
.....|
...XX...|
...OXO...|
...*O...|
....X...|
.....|
.....|
```

但这除非征子有利否则是错误的。为检查这个增加了条件：

```
-----+
```

```

.....|
.....|
...XX...|
...OXO...|
...*OAC...|
....DB...|
.....|
.....|

```

```
;oplay_attack(*,A,B,C,D)
```

为接受这个模式需要分号行评估为 `true`。这个特殊条件解释为“从 `O` 开始在 `*`、`A`、`B` 和 `C` 交替走棋，然后检查 `D` 子能否被提掉”，即到这个局面：

```

-----+
.....|
.....|
...XX...|
...OXO...|
...OOXX...|
....XO...|
.....|
.....|

```

并调用 `attack()` 查看下面的 `X` 子能否被提掉。这不受征子约束，但在这个特别例子当然包含了一个征子。

上面字母形式的条件图形就是存储在 `.db` 文件中的格式。定式编译器知道如何从 SGF 节点的标签创建。Cgoban 有个选项可以创建单字符标签，但这应该是 SGF 编辑器的公共特性。

这样为了在 SGF 中实现本例，需要在四点上加标签和一个注释：

```
;oplay_attack(*,A,B,C,D)
```

接着将适当的条件（自动辅助宏）加入定式 `.db` 文件。

9.18 角部匹配

GNU Go 为定时模式是用一个特殊的匹配器，在可匹配的模式上有确定的条件，并且较标准的匹配器更快、存储模式占用空间更少。

角部匹配器适用的模式必须满足以下条件：

- 必须只在角部可以匹配（这也是名字的由来）。
- 只能包含“O”、“X”和“.”元素。
- 在所有模式值（参见 9.3 模式属性—值）中，角部匹配器只支持 `shape(x)`。这并不是因为匹配器原理上不能处理其他值，只是在定式数据库中没有用到。

角部匹配器是专门为定式模式设计的，自然满足以上所有条件。经过一些修改角部匹配器也可以用作布局匹配器，但全盘匹配器做得更好。

这个匹配器的思路和 DFA 匹配器非常相似（参见 10.3 DFA 模式匹配）：一次性检查所有模式而不是一次检查一个。一个经过修改的 DFA 匹配器可以用作定式匹配，但数据库会相当大。角部匹配器掌握这样一个事实，即每个模式中棋子都很少。

角部模式数据库组织成一个树。树的节点称为“变化”。变化表示棋盘角上的一串棋子。根变化代表一个空角，向下一层等同在角部增加一个棋子。每个变化都有几个属性：

- 棋子对于角部的位置
- 确定棋子是否与首个匹配棋子同色的标志。
- 变化在角区的棋子数量（见下面）

在角部定义一个矩形称角区，对角为棋盘当前角和棋子位置（含）。例如，如果棋盘当前角为 A19 则棋子 C18 的角区包括 A18、A19、B18、B19、C18 和 C19。

根变化的直接子变化当该变化位置上有子且该棋子在其角区唯一时匹配。

树上更深层的变化当该变化位置上有规定方的棋子，且在其角区的棋子数量等于变化结构中定义的数量时匹配。

当一特定的变化匹配后，其所有子节点必须递归检查匹配。

所有叶子变化和一些内部变化有相匹配的模式。对于需要匹配的模式，要求其父结点是匹配的。此外，要检查要匹配的是适当的一方（用变化的“stone

color" 属性)且在模式要匹配的区域内棋子数量实际等于模式的棋子数量。模式变化的"stone position" 属性决定模式建议的着手。

考虑这个四子的定式模式。

```
-----+
.....|
.....|
.O*...|
.XXO..|
.....|
.....|
```

为角部匹配器编码必须使用五个变化，每个都作为子节点连接到先前一个。

树层	位置	方	子数
1	R16	同	1
2	P17	同	1
3	Q16	对	2
4	P16	对	4
5	Q17	同	1

第五个变化有一个相关的模式。注意第五个变化的棋方是同方，因为本模式最先匹配的模式是 O 表示*着手的推荐对象。

树由所有模式的所有变化组成。各模式的变化排序使其可以非常快地抛弃树的分支，同时保持数据库足够小。更多细节参见 mkpat.c。

角部匹配器驻留在 matchpat.c 有两个函数：corner_matchpat() 和 do_corner_matchpat()。第一个计算 num_stones[] 数组存储角区在各种可能的变化下的棋子数量。corner_matchpat() 也匹配最上层的变化。do_corner_matchpat() 负责在变化树上递归匹配并在模式匹配中调用回调函数。

角部匹配器类似树结构的数据库由 mkpat 程序创建。数据库同时生成几个函数，主要有 corner_best_element()、corner_variation_new()、corner_follow_variation() 和 corner_add_pattern()。

9.19 编辑模式的 Emacs 模式

如果使用 GNU Emacs (XEmacs 也一样), 可以使用特殊的模式编辑 GNU Go 的模式数据库。模式在 `patterns/gnugo-db.el`。

将该文件复制到 `emacs/site-lisp` 目录。以后可以用 `(require gnugo-db)` 装载。也可以将此行加入配合文件 (`~/.emacs`)。或者使用 `gnugo-db-mode` 命令切换到模式修改模式, 或者使用以下代码片段自动打开 `.db` 后缀的文件:

```
(setq auto-mode-alist
      (append
        auto-mode-alist
        '(("*.db" . gnugo-db-mode))))
```

模式编辑模式提供以下特性:

- 关键字 (`Pattern`, `goal_elements` and `callback_data`) 和注释高亮显示;
- 段落与模式 (`M-h`, `M-{`, `M-}` 及其他操作等价;
- 模式新建命令自动选择名字 (`C-c C-p`) 和复制主图到条件图命令 (`C-c C-c`),

条件和注释自动缩行

10 DFA 模式匹配

本章描述 gnugo DFA 模式匹配的原则。该系统的目的是在时间紧张时允许类似 owl 模块的快速模式匹配 (12.1Owl 代码)。从 GNU Go 3.2 开始默认使能。但仍可以运行 `configure --disable-dfa` 并重新编译 GNU Go 回到传统的模式匹配器。

另外，可以从模式库离线生成一种有限状态机称“确定有限状态自动机” (Deterministic Finite State Automaton) (10.2DFA 是什么)，可在运行时刻使用，加速模式匹配 (10.3DFA 模式匹配和 10.5 增量算法)。运行时刻加速代价是存储器用量和编译时间。

10.1DFA 简介	通过一个路径扫描棋盘
10.2DFA 是什么	回忆语言理论
10.3DFA 模式匹配	如何获取 dfa 围棋模式
10.4 构建 DFA	开放对局
10.5 增量算法	确定的乐趣

0DFA 模式匹配器的增量版本还没有在 gnugo 实现，在此解释如何工作。定以一个确定状态自动机，扫描同一字符串每次将到达同一状态。

模式匹配中每个到达的状态存储在堆栈 `top_stack[i][j]` 和 `state_stack[i][j][stack_idx]` 节点 (i, j) 使用一个堆栈。棋盘点 (x, y) 通过预计算的反路径表可取得其在从 $(0, 0)$ 起始的螺旋扫描路径位置中的位置 `reverse(x, y)`。

新子放置在棋盘上 (lx, ly) 时，模式匹配器所要做的只是：

```
for(each stone on the board at (i,j))
    if(reverse(lx-i,ly-j) < top_stack[i][j])
```



```

{
    begin the dfa scan from the state
    state_stack[i][j][reverse(lx-i,ly-j)];
}

```

大多数情形下，`reverse(lx-i,ly-j)` 低于 `top_stack[i][j]`，这可以加速大量模式匹配。

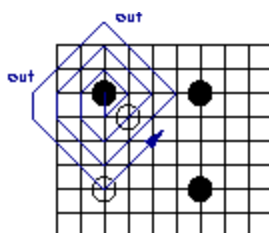
一些 DFA 优化

一些可能的优化

10.1 DFA 简介

基本思想如下：

对于棋盘上所有点，按照预定的路径扫描其邻点。实际所用的路径并没有太大关系；GNU Go 使用如下螺旋路径：



在路径的每一步，模式匹配器跳转到棋盘上目前找到的确定状态。如果在这步已经成功匹配了一个或多个模式，状态（在其属性）立即告知。但状态编码隐含可以匹配更多的模式：状态中存储的信息包含下一步跳转状态，取决于是否在路径下一步找到黑子、白子或空点（或盘外点）。状态也立即告知是否没有更多模式（跳转到错误状态）。

以上肤浅的解释在下节（10.2 DFA 是什么）的定义后会更清晰一些。

沿预定的路径识别棋盘将二维模式匹配降阶为线性文本搜索问题。以下模式作为例子：

```

?X?
.O?
?OO

```

沿以下路径扫描：

```

B
C4A
5139

```

给出棋串 "OO?X.*O*?*?" 而 "?" 表示 "不关心" , "*" 表示 "不关心, 甚至可以是盘外" 。

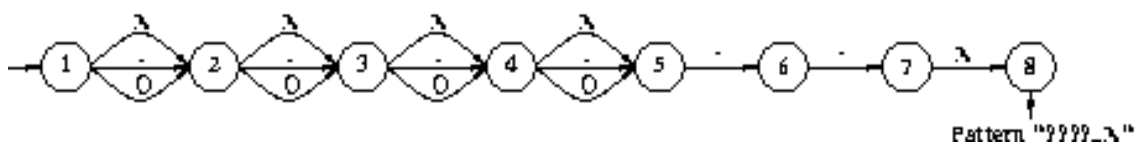
这样就离开了二维模式而考虑线性模式。

10.2 DFA 是什么

DFA 是 Deterministic Finite state Automaton 的缩写, (详细参见 <http://www.eti.pg.gda.pl/~jandac/thesis/node12.html> 确定有限状态自动机或 Hopcroft 和 Ullman "Introduction to Language Theory" for more details)。DFA 是编译器设计中的通用工具(Read Aho, Ravi Sethi, Ullman "COMPILERS: Principles, Techniques and Tools" for a complete introduction) , 许多高效的文本搜索算法如 Knuth-Morris-Pratt 或 Boyer-Moore 算法都是基于 DFA 的(参见 <http://www-igm.univ-mlv.fr/~lecroq/string/> 模式匹配算法大全)。

DFA 根本上是一组由标签转移连接的状态。标签是棋盘上识别的值, 在 gnugo 中是 EMPTY、 WHITE、 BLACK 或 OUT_BOARD, 分别用 '.'、'O'、'X' 和 '#' 标记。

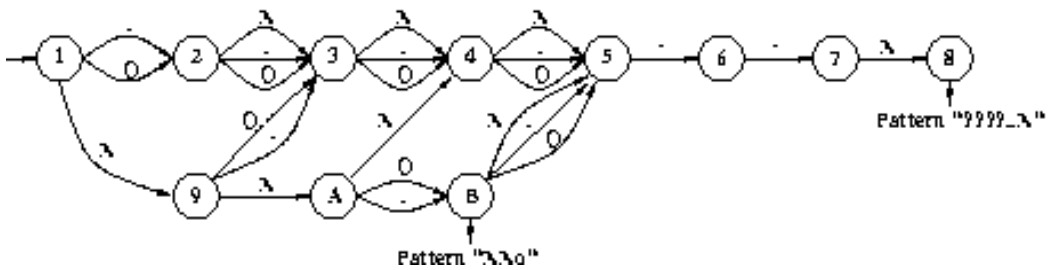
dfa 最好的表示方法是画出转移图: 模式 "????..X" 由以下识别:



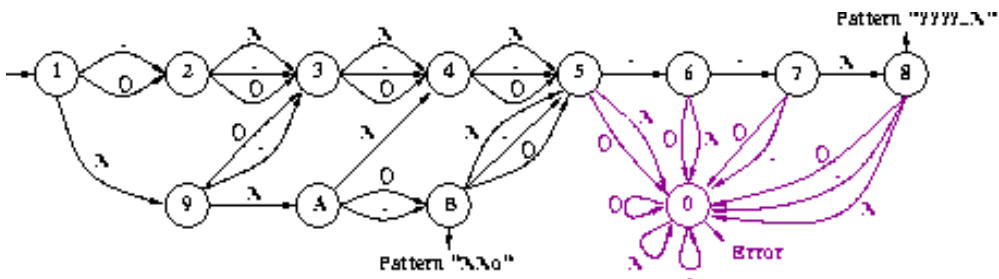
这表示由状态 [1] 开始, 如果在棋盘上识别到 '.'、'X' 或 'O', 则转到

状态 [2] 并继续直到到达状态 [5]。从状态 [5]，如果识别到 '.'，则转移到状态 [6]，否则转移到错误状态 [0]。接着到达状态 [8]。一旦到达状态 [8]，则识别到了模式 "????..X"。

增加一个模式象 "xxo" ('o' 表示非 'x')) 可用同步乘积直接转换自动机 (10.4 构建 DFA) 考虑如下 DFA:



增加特殊的错误状态，对于没有到错误状态的各状态增加到错误状态的转移，即容易地将一 DFA 转换为完全确定有限状态自动机 Complete Deterministic Finite state Automaton (CDFA)。同步乘积 (10.4 构建 DFA) 只能用于 CDFA。



CDFA 图形可编码为状态数组：状态 0 为“错误”状态，初始状态为 1。

状态	.	0	X	#	注释
1	2	2	9	0	
2	3	3	3	0	
3	4	4	4	0	
5	6	0	0	0	
6	7	0	0	0	
7	0	0	8	0	
8	0	0	0	0	????..X
9	3	3	A	0	
A	B	B	4	0	
B	5	5	5	0	xxo

对每个状态连接一个经常为空的属性列表，列出到达该状态后识别的模式索引。在“dfa.h”中由两个结构表示：

```

/* dfa state */
typedef struct state
{
    int next[4]; /* transitions for EMPTY, BLACK, WHITE and OUT_BOARD */
    attrib_t *att;
}
state_t;

/* dfa */
typedef struct dfa
{
    attrib_t *indexes; /* Array of pattern indexes */
    int maxIndexes;

    state_t *states; /* Array of states */
    int maxStates;
}
dfa_t;

```

10.3 DFA 模式匹配

用 DFA 识别非常简单所以很快（参见“engine/matchpat.c”中的“scan_for_pattern()”）。

从初始状态开始，只需要沿着螺旋路径识别棋盘，从根据棋盘上识别到的值标签转移状态，收集模式索引。如果到达错误状态（零），则意味着未匹配到更多模式。该算法最坏情况下复杂度是 $O(m)$ ，这里 m 是最大模式大小。

以下是扫描的例子：

首先建立一个最小 dfa 识别这些模式：“X..X”、“X???”、“X.OX”和“X?oX”。注意王牌“?”、“o”或“x”给出多个转移。

状态	.	O	X	#	注释
1	0	0	2	0	
2	3	10	10	0	
3	4	7	8	0	
4	5	5	6	0	
5	0	0	0	0	2
6	0	0	0	0	4 2 1
7	5	5	8	0	
8	0	0	0	0	4 2 3
9	5	5	5	0	
10	11	11	9	0	
11	5	5	12	0	

12 0 0 0 0 4 2

扫描串 "x..xxo...." 由状态 1 开始:

当前状态: 1, 扫描子串: x..xxo....

识别到 "x", 转移到状态 2。

当前状态: 2, 扫描子串: ..xxo....

识别到 ".", 转移到状态 3.....

Current state: 3, substring to scan : .xxo....

Current state: 4, substring to scan : xxo....

Current state: 6, substring to scan : xo....

Found pattern 4

Found pattern 2

Found pattern 1

到达状态 6 后匹配了模式 1、2 和 4, 这里没有向外转移, 故停止匹配。为保持标准算法中同样的匹配顺序, 模式索引收集到数组并按索引排序。

10.4 构建 DFA

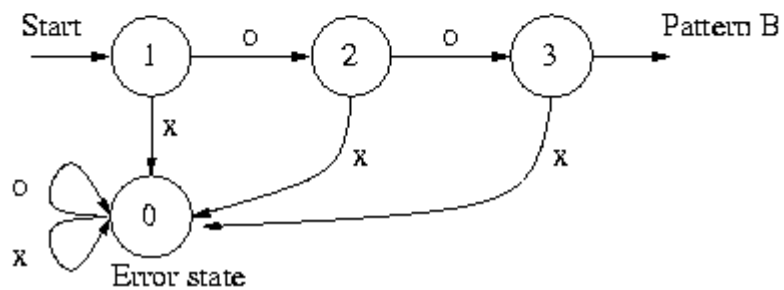
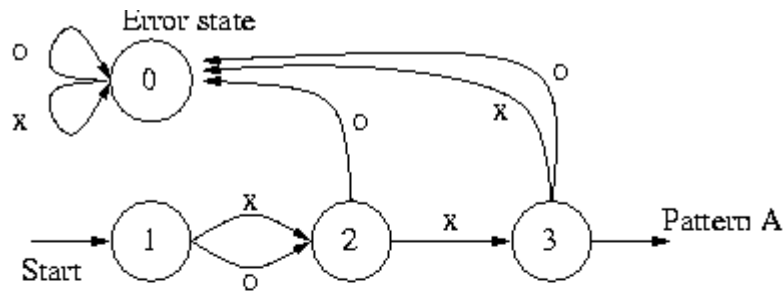
最有趣的一点是构建最小 DFA 识别给定的模式组。为将新模式加入一个已有的 DFA 必须重新构建 DFA: 原则是建立最小 CDFA 识别新模式并用同步乘积替换原 CDFA。

首先给出形式定义: 命 L 是左 CDFA, R 是右 DFA。命 B 是 L 和 R 的同步乘积。其状态是 (l, r) l 是 L 的状态而 r 是 R 的状态。状态 $(0, 0)$ 是 B 的错误状态而 $(1, 1)$ 是初始状态。对于 (l, r) 连接 l 和 r 所识别模式的并集。 B 的转移集是 $(l_1, r_1) \xrightarrow{a} (l_2, r_2)$ 对于符号 'a', $l_1 \xrightarrow{a} l_2$ 在 L 中且 $r_1 \xrightarrow{a} r_2$ 在 R 中。

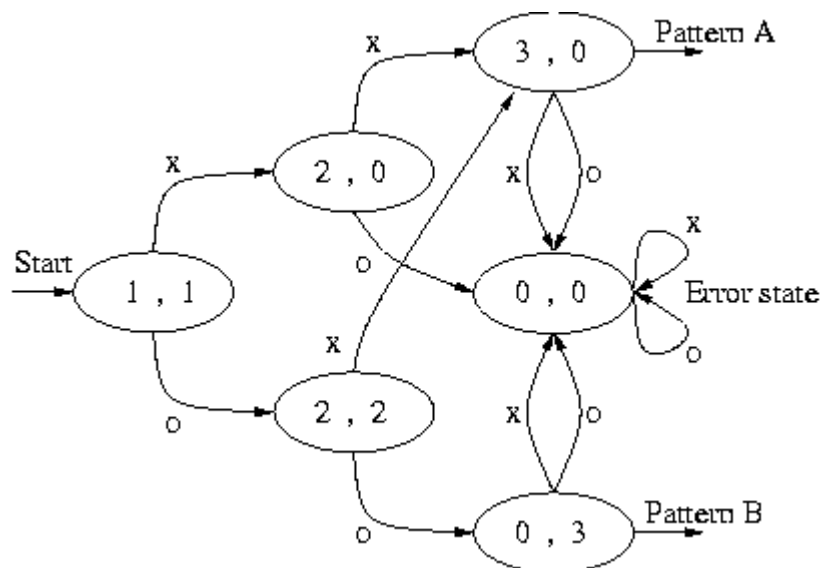
B 的状态最大数是 L 和 R 状态的乘积, 但几乎所有这些状态都不能由初始状态 $(1, 1)$ 到达。

函数 `sync_product()` 中所用的算法保持了可到达的状态而建立最小乘积 DFA。它同时按照 **L** 和 **R** 的转移递归扫描乘积 CDFA。使用一个 hash 表 (`gtest`) 检查状态 (l, r) 是否可到达, 将可到达的状态重映射到一个新的 DFA。这样获取的 CDFA 是最小的且可识别两个模式集的并集。

如下例两个 CDFA:



同步乘积如下:



也可以建立一个特殊的模式数据库生成“爆炸性”自动机：DFA 的尺寸最坏情况下是其识别模式数目的指数。但这实际情况中不会发生：dfa 尺寸趋近稳定。稳定是指如果增加一个模式大大地增加了 dfa 的尺寸，则同时也增加了下一次加入新模式不增加尺寸的可能性。无论如何有许多方法减小 DFA 的尺寸。Aho, Ravi Sethi, Ullman "COMPILERS: Principles, Techniques and Tools" 基于 DFA 模式匹配的优化一章介绍了好的压缩方法。

10.5 增量算法

DFA 模式匹配器的增量版本还没有在 gnugo 实现，在此解释如何工作。定以一个确定状态自动机，扫描同一字符串每次将到达同一状态。

模式匹配中每个到达的状态存储在堆栈 `top_stack[i][j]` 和 `state_stack[i][j][stack_idx]` 节点 (i, j) 使用一个堆栈。棋盘点 (x, y) 通过预计算的反路径表可取得其在从 $(0, 0)$ 起始的螺旋扫描路径位置中的位置 `reverse(x, y)`。

新子放置在棋盘上 (lx, ly) 时，模式匹配器所要做的只是：

```
for(each stone on the board at (i,j))
  if(reverse(lx-i,ly-j) < top_stack[i][j])
  {
    begin the dfa scan from the state
    state_stack[i][j][reverse(lx-i,ly-j)];
  }
```

大多数情形下，`reverse(lx-i,ly-j)` 低于 `top_stack[i][j]`，这可以加速大量模式匹配。

10.6 一些 DFA 优化

构建 dfa 时为使存储器跳转最少而做出一些值出现频度的假定：EMPTY 最经常出现，故 '.' 转移栽存储器中几乎总是后继。OUT_BOARD 很少遇到，故 '#' 转移几乎总是长跳转。

11 战术识别

11.1 识别基础

11.2 局面 Hash

11.3 持续的识别缓存

11.4 劫争操作

11.5 劫争实例 1

11.6 劫争实例 2

11.7 替代 Komaster 设计

11.8 超串

11.9 调试识别代码

11.10 联络识别

模拟双方可能的着手，审度各种着手应对结果的过程称为"识别"。GNU Go 做了三种不同的识别：战术识别主要计算各棋串的死活，Owl 识别计算棋块的死活和联络识别。本章描述在"engine/reading.c"中的战术识别代码。

11.1 识别基础

所谓战术识别是分析一个单独的棋串是否可被立即提掉或者有一个着手可以避免之。

如果识别模块发现某棋串能被提掉，该结果（一般）是可信的，但如果说可以防守，就不见得如此了。经常是某棋串死定了，但从战术识别（和启发是搜索的修剪策略）的水准看还不能被提掉。

战术识别由"engine/reading.c"中的函数实现，是一个最小-最大搜索，攻方发现能提吃对方棋串或守方发现棋串有足够气数即为成功。有五气的

棋串总被认定是活棋。在搜索树的深层更少的气数也会使 GNU Go 放弃攻击，参见 11.1.3 识别修剪和深度参数。

识别使用堆栈压入棋盘局面。GNU Go 检查实际棋盘局面时参数 `stackp` 设为零 `is_exam` 如果大于零，GNU Go 检查假想若干着以后的局面。

最重要的公共识别函数是 `attack` 和 `find_defense`。是在 `reading.c` 静态声明的函数 `do_attack` 和 `do_find_defense` 的封装。函数 `do_attack` 和 `do_find_defense` 互相递归调用。

11.1.1 识别代码组织

封装函数 `do_attack` 和 `do_find_defense` 根据气数调用 `attack1`、`attack2`、`attack3` 或 `attack4` 和对应的 `defend1`、`defend`、`defend1` 或 `defend1`。

函数调整好以高效的顺序生成和尝试着手。它们自己生成几个着手（多数是棋串的直接气数），接着调用辅助函数系 `..._moves` 建议较不明显的着手。具体调用哪个函数取决于气数和当前搜索深度）。

11.1.2 返回代码

识别（和 `owl`）函数的返回代码和 `owl` 可为 `0`、`KO_B`、`KO_A` 或 `WIN`。每个识别函数确定某方（着手方）特定问题是否有解，典型的是棋串的攻防。

返回代码 `WIN` 表示成功、`0` 失败、`KO_A` 和 `KO_B` 表示劫争。函数返回 `KO_A` 表示着手方先提劫（对方必须先寻劫），`KO_B` 表示着手方先寻劫。

如果 GNU Go 编译时带配置选项 `--enable-experimental-owl-ext` 则 `owl` 函数还可返回代码 `GAIN` 和 `LOSS`。代码 `GAIN` 表示攻击（或防守）不

成功，但尝试攻击或防守过程中吃住一对方棋串，代码 `LOSS` 表示攻击或防守成功，但攻击或防守中本方另一着手死。

11.1.1.3 识别修剪和深度参数

识别深度由参数 `depth` 和 `branch_depth` 控制。`depth` 默认值 `DEPTH` (在 `"liberty.h"` 中)，设定为 `16`，但可用命令行参数 `"-D"` 或 `"--depth"` 选项设置。`depth` 增加，GNU Go 会变强但变慢。GNU Go 可以超出深度识别，但这样做简化假设而导致错失着手。

特别地，当 `stackp > depth`，GNU Go 假定棋串一旦有 3 气即活。这样的假定足可识别征子。

`branch_depth` 典型值为小于 `depth`。在 `branch_depth` 和 `depth` 之间考虑攻击 3 气的棋串，但禁止分支，所以考虑变化较少。

`findex small_semeai` 目前识别代码并不尝试攻击对方多于两气的对方棋串以防守，这个限制存在漏洞。其中一个问题就是将两个相邻的有三或四气的棋串会被分类为死棋。为解决此类问题，使用函数 `small_semeai()` (在 `"engine/semeai.c"` 中) 寻找这样的棋串对并修正分类。

`backfill_depth` 是类似的变量，默认值为 `12`。低于此深度时，GNU Go 尝试“回填”吃子，如：

```
.000000.  
00XXXXXO  
.aObX.XO  
-----
```

在盘边 O 可以吃掉 X，但必须先于 a 接住再于 b 叫吃，称回填。

仅在 `stackp <= backfill_depth` 时尝试回填，参数 `backfill_depth` 可用 `"-B"` 选项设置。

`fourlib_depth` 参数默认值为 7。低于此深度时，GNU Go 尝试攻击四气的棋串。`fourlib_depth` 可用“-F”选项设置。

参数 `ko_depth` 是类似的修剪。`stackp < ko_depth` 时，识别代码试验非法提劫（即假定寻劫且对方应劫）。该参数可用“-K”选项。

- `int attack(int str, int *move)`

确定 `str` 上棋串是否可攻击，如果是，在 `*move` 返回攻着，除非 `*move` 是控指针。（如果只关心攻击结果而不关心攻着可用空指针调用）。如果攻击成功返回 `WIN`，失败返回 0，结果依赖于劫则返回 `KO_A` 或 `KO_B`。

- `find_defense(int str, int *move)`

尝试发现防守 `str` 棋串的着手。如果能找到返回 `true`，在 `*move` 返回着手（除非 `*move` 是空指针）。由于不考虑脱先防守，因此 `!attack(str)` 时可能返回错误结果。如果结果依赖于劫则返回 `KO_A` 或 `KO_B`。

- `safe_move(int str, int color) :`

函数 `safe_move(str, color)` 检查着手 `str` 是否合法且不能立即被提掉。如果 `stackp == 0` 缓存结果。如果着手只能劫争提吃，则视为安全。这可能是个好方法也可能不是。

11.2 局面 Hash

为加速识别过程，我们注意到多种不同的路径可能到达同一局面。实际上，一个先前已经检查过的局面，非常有可能在同一次搜索中，由递归树的另一个分支上又一次被检查到。

这样会浪费很多计算资源，因此可在很多地方存储当前局面、所在函数和参与攻防的棋串；该局面搜索完毕后，再存储搜索结果和成功攻防的着手。


所有这些数据都存储在一个 `hash` 表中，有时称为转移表，着手位置作为键值，相关函数和群的识别结果作为数据。可以使用 `-M` 或 `--memory` 选项增加

Hash 表的大小，参见 3.9 启动 GNU Go：命令行格式。

Hash 表在对局开始时由函数 `hashtable_new()` 创建并分配内存，在每手棋之前从 `genmove()` 调用 `hashtable_clear()` 重新初始化。


11.2.1 Hash 值计算

Hash 算法采用 Zobrist，该算法是围棋和国际象棋编程的标准技术，其工作原理如下：

 首先定义围棋局面。局面包括


- a) 实际棋盘即棋子归属方和位置。
- b) 劫争点，如果有劫的话。劫争点定义为上一手被提单子的位置。
- c) 作为局面一部分没有必要规定先手方（白或黑），原因是识别结果对于

各种识别函数是分别存储的，如 `attack3` 在调用函数中隐含先手方。

 对于棋盘上每个位置生成随机数：

- 该位置是白子时；
- 该位置是黑子时；
- 该位置是劫争点时；

这些随机数在初始化时生成并在 hash 表整个生存周期中使用。

 某局面的 hash 键值为用于该局面三个随机数的异或计算结果。（白子、黑子和劫争点）

11.2.2 Hash 表组织

Hash 表包括 3 部分：

- Hash 节点，包括：
 - 上述着手位置

- 该位置 hash 值
- 识别结果指针（如下）
- 指向其他节点的指针
- 识别结果。存储该着手位置的调用函数，参与攻防的棋串和识别结果。

每个识别结果包括：

- 函数 ID(0 到 255 的整型数)，参与攻防的棋串的位置和搜索深度，封装为 32 位整型数。
- 识别结果（整型数）和着手位置封装为 32 位整型数。
- 指向其他识别结果的指针。
- 指向 hash 节点的指针数组。为 hash 表属性。

创建 hash 表时，这 3 个域由 `malloc()` 分配内存，hash 建立后，所有内容都由 Hash 节点和识别函数生成，不再额外分配内存，一旦用完了所有的节点或结果，hash 表就满了。除非全部清空重新使用，不删除 hash 表中任何一个节点。

函数使用 hash 表前，先使用 `hashtable_search()` 查找当前位置，如果该位置不存在，则可使用 `hashtable_enter_position()` 加入。

当一个函数指向 hash 表中一个包含了函数的节点，可以使用 `hashnode_search()` 获取以前的搜索结果，并使用，如果没有，则可以在搜索后使用 `hashnode_new_result()` 将结果加入。

在 Hash 表中有相同入口的节点构成一个简单的链表。相同位置上，不同函数、不同的攻防棋串生成的识别结果也构成一个简单的链表。

Hash 入口需要多少位是一个问题，为了节省内存和加速计算，可以采用 32 位，这样偶尔会出现不可恢复的故障（可能是中止故障）。采用 64 位的话

这样的故障会低于 10%，如果这还不够安全，可以设定为 96 位。

11.2.3 Hash 结构

Hash 结构在“engine/hash.h”和“engine/cache.c”中定义

```
typedef struct hashposition_t {  
    Compacttype board[COMPACT_BOARD_SIZE];  
    int ko_pos;  
} Hashposition;
```

表示棋盘和可能的劫争点，即禁着点。不记录着手方。

```
typedef struct {  
    Hashvalue hashval[NUM_HASHVALUES];  
} Hash_data;
```

表示函数的返回值。

```
typedef struct read_result_t {  
    unsigned int data1;  
    unsigned int data2;  
  
    struct read_result_t *next;  
} Read_result;
```

域 data1 封装为 32 位包含以下域：

```
komaster: 2 bits (EMPTY、BLACK、WHITE, 或 GRAY)  
kom_pos : 10 bits (允许 MAX_BOARD 最大 31)  
routine : 4 bits (目前 10 种选择)  
str1 : 10 bits  
stackp : 5 bits
```

域 data2 封装为 32 位包括以下域：

```
status : 2 bits (0 free、1 open、2 closed)  
result1: 4 bits  
result2: 4 bits  
move : 10 bits  
str2 : 10 bits
```

域 komaster 和 kom_pos 参见 11.4 劫争操作。

新建结果节点时，“status”设为 1 “open”，结果输入后设为 2 “closed”，其主要用途是部分清除 hash 表时标识识别开放结果节点。另一

可能的用途是在识别中标识重复局面，特别是局部双劫或三劫的局面。

```
typedef struct hashnode_t {
    Hash_data      key;
    Read_result    * results;
    struct hashnode_t * next;
} Hashnode;
```

Hash 由 hash 节点组成，每个 hash 节点由局面 hash 值、局面和实际信息。

另外还有一个指针在排序时指向另一 hash 节点（如下）。

```
typedef struct hashtable {
    size_t      hashtablesize; /* Number of hash buckets */
    Hashnode    ** hashtable; /* Pointer to array of hashnode lists */

    int         num_nodes; /* Total number of hash nodes */
    Hashnode    * all_nodes; /* Pointer to all allocated hash nodes. */
    int         free_node; /* Index to next free node. */

    int         num_results; /* Total number of results */
    Read_result * all_results; /* Pointer to all allocated results. */
    int         free_result; /* Index to next free result. */
} Hashtable;
```

Hash 表包括三个部分：

- hash 表属性：由链表操作的 hash 中冲突数量。
- hash 节点。在创建时刻分配，目前实现中永不移除或重分配。
- 搜索结果。同一局面可能做不同的搜索，故会多于 hash 节点。

11.3 持续的识别缓存

一些计算结果可以可靠地按着手存储。如果对方的着手不靠近本方的棋串或棋块，下一着手不必重新考虑棋群死活，而是用在持续缓冲中存储的结果。引擎对于几种识别结果使用了持续缓存。

- 战术识别
- Owl 识别

- 联络识别
- 打入代码

本节讨论战术识别的持续缓存，同样原则可用于其它持续缓存。

持续缓存是一个重要的运行特性，然而它可能会导致错误和调试问题——GNU Go 可能在调试期间生成正确的着手而在对局中出错着。如果怀疑持续缓存的影响可以用 `--replay` 选项装载 `sgf` 文件检查错误是否复现（参见 3.9 启动 GNU Go：命令行格式）。

函数 `store_persistent_cache()` 由 `attack` 和 `find_defense` 调用，而不是由其静态递归变体 `do_attack` 和 `do_defend`。函数 `store_persistent_reading_cache()` 尝试缓存代价最高的识别结果，函数 `search_persistent_reading_cache` 尝试从缓冲中获取结果。

如果所有缓冲入口都用光了，尝试覆盖最无用的一个。这由 `score` 域即该识别所消耗的的节点数，再乘以取出次数确定。

确定一个（固定）着手后，由函数 `purge_persistent_reading_cache()` 清除无效的缓冲入口。函数 `store_persistent_cache()` 计算识别镜像和活动区域以确定清除的时机。如果在活动区域接着一个固定的着手，则缓冲结果失效。以下详细解释该算法。

识别镜像是所有着手所有变化的组合，还包括尝试非法着手的局面。

识别结束后，识别镜像展到可被缓存的活动区域，目的是当该活动区域再没有着手时，可以安全使用缓冲值。

以下算法在 `store_persistent_reading_cache()` 中，用以计算活动区域，该算法在持续缓存中存储代价最高的识别。

- 识别镜像和受攻棋串标记字母“1”，还包括成功着手，后者通常是识别镜像的一部分，但有时不是，如函数 `attack1()`。

- 接着扩展识别镜像，将与标记“1”的区域相邻的棋串和空点标记为字母“2”。
- 接着将与标记“2”的空节点相邻的节点标记为字母“3”。
- 接着标记所有与前述节点相邻的节点。这时的标记是“-1” 而不是更逻辑化的“4” ，这样编码能够略微快一些。
- 如果堆栈指针 > 0 则将着手堆栈中标记 4 的已走着手加入。

11.4 劫争操作

战术识别中劫争操作的原则与 owl 识别相同。

已经提到（参见 11.1 识别基础）如果结果依赖于劫争，GNU Go 使用返回代码 KO_A 或 KO_B 。返回代码 KO_B 表示如果着手方有足够劫材的话局面取胜。为验证之，函数必须模拟寻劫并假定对方应劫即在局面非法提劫，称条件提劫。

条件提劫由函数 `tryko()` 实现，该函数类似 `trymove()` 区别在与不考虑合法性。

静态识别函数和全局函数 `do_attack` 和 `do_find_defense` 带有参数 `komaster`、`kom_pos` ，在提劫时防止循环提劫。

一般地 `komaster` 为 `EMPTY` 但也可以是 `BLACK`、`WHITE` 或 `GRAY`。
`color` 条件提劫时 `komaster` 赋值 `color` ，这时 `kom_pos` 为被提劫点。

如果对方是 `komaster`，识别函数不尝试在 `kom_pos` 提劫。同样，`komaster` 一般不允许提另一个劫。例外的是连环劫，即被提劫点在 `kom_pos` 对角点上，如下：

```
.OX
OX*X
OmOX
```

“m” 是 kom_pos ，则允许“*” 上着手。

该规则的理论依据是如果棋盘上有两个劫，komaster 不可能两个都赢，成为 komaster 时已经选择可要赢的劫，但在连环劫中，赢取一个劫的前提条件是赢取另一个，所以允许连环劫。

如果 komaster 的对方提劫，则双方各提一劫，此时 komaster 赋值 GRAY 且不允许再提。

如果在 kom_pos 粘劫，则 komaster 赋值 EMPTY 。

Komaster 方案详细描述如下。着手方为“O” 。在 GNU Go 中包含了数种不同的方案，该方案称方案 5。

- Komaster 是 EMPTY :

- 1a) 允许无条件提劫。

如果前着手不是提劫 Komaster 保持 EMPTY，如果前着手是提劫且 kom_pos 赋值 board_ko_pos 原值则 Komaster 赋值 WEAK_KO。

- 1b) 允许条件劫争。

Komaster 赋值 O ， kom_pos 为被提劫点。

- Komaster 是 O:

- 2a) 只允许连环劫。Kom_pos 是新提劫点。

- 2b) 如果 komaster 在 kom_pos 粘劫则 komaster 赋值 EMPTY 。

- Komaster 是 X:

不允许 kom_pos 着手。允许其它劫争。如果 O 提另一劫，komaster 赋值 GRAY_X。

- Komaster 是 GRAY_O 或 GRAY_X:

不允许提劫。如果在 kom_pos 粘劫则 komaster 赋值 EMPTY。

- Komaster is WEAK_KO:

- 5a) 非劫争着手后 komaster 赋值 EMPTY。

- 5b) 仅当连环劫时允许无条件提劫。

Komaster 改为 WEAK_X 且 kom_pos 为 board_ko_pos 原值。

- 5c) 根据规则 1b 允许条件提劫。

11.5 劫争实例 1

为 实 际 考 察 komaster 方 案 ， 考 虑“regressions/games/life_and_death/tripod9.sgf” 文件中的局面，建议采用以下命令研究例子中的变化。

```
gnugo -l tripod9.sgf --decide-dragon C3 -o vars.sgf
```

在左下角，A2 和 B4 有劫。W 可胜其一，因此黑无条件死。

```
8 . . . . .
7 . . O . . . .
6 . . O . . . .
5 O O O . . . .
4 O . O O . . . .
3 X O X O O O O .
2 . X X X O . . .
1 X O . . . . .
  A B C D E F G H
```

Komaster 这样解释。B（即 X）先 B4 提劫，W 在 A2 提劫，变化为：

```
8 . . . . .
7 . . O . . . .
6 . . O . . . .
5 O O O . . . .
4 O X O O . . . .
3 X . X O O O O .
2 O X X X O . . .
1 . O . . . . .
```

A B C D E F G H

现在 B 只有在 A1 再次提劫（非法）才行，这样 B 再提劫成为 komaster，

变化为：

```
8 . . . . . komaster: BLACK
7 . . O . . . kom_pos: A2
6 . . O . . . .
5 O O O . . . .
4 O X O O . . . .
3 X . X O O O O .
2 . X X X O . . .
1 X O . . . . .
A B C D E F G H
```

在 B3 提劫后 komaster 赋值 GRAY 且不允许再提劫：

```
8 . . . . . komaster: GRAY
7 . . O . . . kom_pos: B4
6 . . O . . . .
5 O O O . . . .
4 O . O O . . . .
3 X O X O O O O .
2 . X X X O . . .
1 X O . . . . .
A B C D E F G H
```

B 不允许提劫而死，这样 komaster 方案得到正确结果。

11.6 劫争实例 2

现在考虑一个例子，komaster 消劫后复位到 EMPTY 的情况。此时粘劫或劫争点不再是劫。

导致如下局面的文件在“regressions/games/ko5.sgf”：

```
. . . . . O O 8
X X X . . . O . 7
X . X X . . O . 6
. X . X X X O O 5
X X . X . X O X 4
. O X O O O X . 3
O O X O . O X X 2
. O . X O X X . 1
F G H J K L M N
```

建议使用如下研究例子的变化：

```
gnugo -l ko5.sgf --quiet --decide-worm L1 -o vars.sgf
```

正解是 H1 无条件吃 L1，而 K2 劫争防守（代码 KO_A）。

黑（X）在 K2 提劫后，白只有再条件提劫成为 komaster，B 只有 K4 而白。结果如下：

```
. . . . . O O 8
X X X . . . O . 7
X . X X . . O . 6
. X . X X X O O 5
X X . X X X O X 4
. O X O O O X . 3
O O X O . O X X 2
. O O . O X X . 1
F G H J K L M N
```

重要的是“O”不再是 komaster。如果“O”还是 komaster，则要 N3 提劫没办法吃住 B。

11.7 替代 Komaster 设计

以下是建议的替代方案，假定“O”是着手方。

11.7.1 2.7.232 基本方案

- Komaster 是 EMPTY :
 - 允许无条件提劫。Komaster 保持 EMPTY 。
 - 允许条件劫争。 Komaster 赋值 O ， kom_pos 为被提劫点。
- Komaster 是 O :
 - 不允许条件提劫。
 - 允许无条件提劫。Komaster 参数不变。
- Komaster 是 X :

- 不允许条件提劫。
- 允许除着手在 kom_pos 无条件提劫，Komaster 参数不变。

11.7.2 2.7.232 修订版本

- Komaster 是 EMPTY :
 - 允许无条件提劫。Komaster 保持 EMPTY 。
 - 允许条件劫争。 Komaster 赋值 O ， kom_pos 为被提劫点。
- Komaster 是 "O" :
 - 仅当着手后 is_ko(kom_pos, X) 返回 false 时允许提劫。此时 kom_pos 更新为新的劫争点即该着手的提子。
- Komaster 是 "X" :
 - 不允许条件提劫。
 - 允许除着手 kom_pos 外无条件提劫。Komaster 参数不变。

11.8 超串

超串是按下述联络扩展的棋串：

1. 实联络（棋串）

OO

2. 对角联络或一间跳而对方在间隔点着手为自杀或被叫吃。

...

O.O

XOX

X.X

3. 双联络。

OO

..

OO

4. 对角联络而两个相邻点均为空。

.O

0.

5. 通过可战术吃住的相邻点或对角点联络。当调用来自“reading.c”时超串代码忽略这种类型的联络，调用来自“owl.c”时保留。

像棋块一样，超串是棋串的合并，但棋子组织较棋块更紧密，且用途不同。有时攻防超串的一部分是攻防棋串的最好方式，所以在战术识别中可以用到超串，而棋块在战术识别中是被忽略的。

11.9 调试识别代码

识别代码寻找一个通过着手树确定是否能吃住一棋串的途径。检查工具是选项“--decideworm”，可带或不带输出文件。

简单地执行

```
gnugo -t -l [input file name] -L [movenumber] --decideworm [location]
```

即运行 `attack()` 确定是否能吃住棋串，同时运行 `find_defense()` 确定是否能防守。给出识别的变化数。“-t”是必须的，否则 GNU Go 不报告找到的结果。

如果增加“-o output file” GNU Go 生成一个输出文件包含考虑的所有变化。变化在注释中编号。

没有浏览代码的方式时变化文件没有太大作用。它由 GDB 源文件生成列在最后。可以从 GDB 生成，或作为 GDB 初始文件。

如果使用 GDB 调试 GNU Go，可以发现不优化编译更不容易产生混淆。优化有时会改变程序的执行顺序。如不优化编译“reading.c”，编辑“engine/Makefile”从文件中删掉字符串“-O2”联编“engine/reading.c”。注意 Makefile 是自动生成的，接着会被覆盖。

如果在识别过程中需要分析一个结果，而函数是从 hash 代码缓存局面中取值，用命令行“--hash 0”关闭 hash 重新运行实例，应当得到相同的结果。

除非有更好的理由，否则不要运行“--hash 0”，因为那样会增加变化数。

装入后面给出的源文件，可以浏览变化。在 `cgoban` 中使用小“-fontHeight”可使变化窗口显示更多。（可以改变棋盘尺寸）。

假设检查文件后发现变化 17 有趣并想看到确切的过程。

宏“jt n”可跳转到第 n 个变化。

```
(gdb) set args -l [filename] -L [move number] --decidestring [location]
(gdb) tbreak main
(gdb) run
(gdb) jt 17
```

跳转到问题位置。

实际上攻防变化是分别编号的。（但 `find_defense()` 仅当 `attack()` 后运行，所以防守变化也许存在也许不存在。）每次都 `tbreak main` 有些多余，可以用两个宏 `avar` 和 `dvar`。

```
(gdb) avar 17
```

重启程序，跳转到第 17 攻击变化。

```
(gdb) dvar 17
```

跳转到第 17 防守变化。两个变化可在同一 `sgf` 文件中找到，但分别编号。

文件中定义的其它命令如下：

`dump` 打印着手堆栈。

`nv` 移动到下一变化

`ascii i j` 转换 (i,j) 到 `ascii`

```
#####
#####      .gdbinit file      #####
#####

# this command displays the stack

define dump
set dump_stack()
```

```

end

# display the name of the move in ascii

define ascii
set gprintf("%o%m\n", $arg0, $arg1)
end

# display the all information about a dragon

define dragon
set ascii_report_dragon("$arg0")
end

define worm
set ascii_report_worm("$arg0")
end

# move to the next variation

define nv
tbreak trymove
continue
finish
next
end

# move forward to a particular variation

define jt
while (count_variations < $arg0)
nv
end
nv
dump
end

# restart, jump to a particular attack variation

define avar
delete
tbreak sgffile_decidestring
run
tbreak attack

```

```

continue
jt $arg0
end

# restart, jump to a particular defense variation

define dvar
delete
tbreak sgffile_decidestring
run
tbreak attack
continue
finish
next 3
jt $arg0
end

```

11.10 联络识别

GNU Go 识别确定棋串是否联络，算法在“readconnect.c”。象识别代码一样，联络代码不是基于模式的。

联络代码由引擎通过以下函数启动：

- `int string_connect(int str1, int str2, int *move)`

如果 `str1` 和 `str2` 联络返回 `WIN`。

- `int disconnect(int str1, int str2, int *move)`

如果 `str1` 和 `str2` 分断返回 `WIN`。

为观察联络代码实际工作情况，可用以下例子。

```
gnugo --quiet -l connection3.sgf --decide-connection M3/N7 -o vars.sgf
```

（文件“connection3.sgf”在“regression/games”。）检查生成的 sgf 文件察看函数 `decide_connection` 调用 `string_connect()` 和 `string_disconnect()` 进行了哪些识别。

识别代码的一个用途是在联络数据库的自动辅助函数宏 `oplay_connect` 、 `xplay_connect` 、 `oplay_disconnect` 和

xplay_disconnect.

12 基于模式的识别

战术识别代码存储在“reading.c”中，为效率缘故，生成尝试着手的代码都是手写的 C 代码。需要多说一点的是另一种识别类型，它的着手取自模式数据库。

GNU Go 有三种主要的模式识别。首先是 OWL(Optics with Limit Negotiation) 代码试图识别出可供评估的一点，其代码在“engine/optics.c”中（参见 8 眼和后手眼），象战术识别代码一样，使用了一个持续缓存来保持着手之间的 owl 数据，这是一个必要的加速手段，否则 GNU Go 对弈会很慢。

其次是“engine/combination.c”尝试寻找组合情形，即通过一连串的攻击达成不可避免的结果。

最后是对杀模块。对杀是两个状态为 DEAD（死棋）或 CRITICAL（危险）相邻的黑白棋块互相比拼谁先杀死对方。主要函数 owl_analyze_semeai() 包含在“owl.c”中。由于对杀的复杂性，该函数比普通的 owl 代码更容易出错。

12.1 Owl 代码

另外章节（8 眼和后手眼）描述的“optics.c”中死活代码在定型局面上工作良好。所谓定型局面，指在没有着手能够扩展或缩小眼位。棋块被包围但尚有一点空间做眼时，停止基于图形的分析，代之为少量趋向定型局面的识别。

防守方尝试扩展眼位，而攻击方则缩小眼位，如果双方都没有找到有效的着手，立即评估该局面。我们称此类死活识别为，有限算路支持的图形 Optics With Limit-negotiation (OWL)，在“engine/owl.c”中实现

该模块。

有两个相当小的数据库 “patterns/owl_defendpats.db” 和 “patterns/owl_attackpats.db” 存储扩展和缩小眼位的着手。“owl.c” 中的代码生成一个小的着手树，攻着取自 “owl_attackpats.db” 守着取自 “owl_defendpats.db”。除模式库建议的着手外，还需测试眼位分析得到的致命着手。

第三个数据库 “owl_vital_apats.db” 模式用来取代图形识别代码做出的眼位分析。由于眼位图形忽视了气紧和包围圈的突破点的复杂性，所以静态眼位分析有时会出错。问题在于当图形代码说一棋块确定有 2 眼时，实际由于气紧的原因并非如此，所以 owl 模式不进入对局，在这种情况下，只有用 “owl_vital_apats.db” 来度量和修正图形代码的错误。目前 “owl_vital_apats.db” 中的模式仅在级别 9 以上匹配。

调整 Owl 代码的方法是编辑这三个模式数据库，主要是前两个。

如果从 “owl_attackpats.db” 或 “owl_defendpats.db” 找不到更多的着手，或者函数 `compute_eyes_pessimistic()` 报告该棋群已经确定活棋，则可确认着手树这个节点为叶节点。在这点可以评估棋群的状态。函数 `owl_attack()` 和 `owl_defend()` 与 `attack()` 和 `find_defense()` 用法类似，利用模式库生成着手树并确定棋群状态。

Owl 代码中的 `compute_eyes_pessimistic()` 函数采用非常保守的算法，仅当眼位完全封闭（即没有临界交点）的情况下确定成眼。

“engine/owl.c” 开头的参数 `MAX_MOVES` 定义每个节点尝试的最多着手数，根据以下规则优先尝试最有价值的着手。

- 如果 `stackp > owl_branch_depth` 则每个变着只尝试一个着手：
- 如果 `stackp > owl_reading_depth` 结束识别，判定守方胜出（因

为进一步识别可能是逃出的表示)

- 如果节点数超过了 `owl_node_limit` 也停止识别并判定守方胜出
- 估值为 99 的模式确定为打将, 不再尝试其他着手, 假如发现两个这样的着手, 函数返回 `false`, 这只针对攻方而言。
- `"patterns/owl_attackpats.db"`

和 `"patterns/owl_defendpats.db"` 中所有估值为 100 的模式确定为胜出: 如果 `owl_attack` 或 `owl_defend` 发现这样的着手即返回 `true`。这个特性必须慎用。

函数 `owl_attack()` 和 `owl_defend()` 类似 `attack()` 和 `find_defense()` 可能通过指针类型参数返回一个攻着或守着。如果该局面已经胜出 `owl_attack()` 可能返回一个攻着也可能不返回, 如果它找不到有价值的着手就返回 `PASS`, 就是 0。`owl_defend()` 也类似这样。

调用 `owl_attack()` 或 `owl_defend()` 后, 在数组 `goal` 中标记了受攻的棋块, 最初属于该棋块的棋子标记 `goal=1`, 后来由 `owl_defend()` 加入的棋子标记 `goal=2`。如果最初属于该棋块的所有棋串都被提掉, `owl_attack()` 就确定该棋块败北, 即使后加入的一些棋子可以构成一块活棋群。

从 `make_dragons()` 调用该函数时只研究出路狭窄的棋块。

使用 `"--decide-owl location"` 选项可以方便地测试 Owl 代码, 检查位置 `location` 上的棋块死活时应使用选项 `"-t"` 选项生成有用的跟踪、`"-o"` 生成变着的 SGF 文件, 使用选项 `"--decide-position"` 可对所有出路狭窄的棋块执行同样的分析。

12.2 组合识别

可能会发生这样的情况，没法全部杀掉一组棋串，但有一着手可以保证杀掉其中至少一个棋串，这最简单的例子就是双叫。“combination.c”中的代码就是寻找这样的着手。

作为例子，考虑以下情况

```
+-----  
|...OOOOX  
|...OOXXX  
|..O.OXX..  
|.OXO.OX..  
|.OX..OO..  
|.XXOOOXO.  
|..*XXOX..  
|...XOX..  
|.XX..X..  
|X.....
```

这个棋盘局面中所有“X”都是活的，但在“*”处着手形成这样的局面，即四个棋串中至少有一个要被提掉，这就是组合。

典型的组合包括一连串的叫吃最后提吃，所以引擎在这里的驱动称为 `atari_atari`，尽管有时包含的着手不是打将。如上面的例子，在“*”的最初着手并不是打将，但如果“O”进行防守，随即就有一连串的叫吃最后提掉部分棋串。

象 `owl` 函数一样，`atari_atari` 基于模式识别。生成攻击着手的数据库是 `aa_attackpats.db`，该函数的一个危险在于它尝试的第一个打吃着手可能与实际的组合无关。为发现这种可能性，我们一旦找到一个组合，就将第一个着手标记为禁止，然后重新尝试。如果没发现更大规模的组合，则第一个着手即与实际一致。

- `void combinations(int color)`

生成组合攻防的着手目标。是由 `genmove()` 调用的着手生成器之一。

- `int atari_atari(int color, int *attack_move, char`


```
defense_moves[BOARDMAX], int save_verbose)
```

为 color 寻找一个组合。为着手生成返回最小受攻棋串的大小。

- ```
int atari_atari_confirm_safety(int color, int move, int *defense, int
minsize, const char saved_dragons[BOARDMAX], const char
saved_worms[BOARDMAX])
```

试图确定一着手是否错误，围绕 atari\_atari\_blunder\_size。在着手 move 后检查不小于 minsize 的攻击组合。数组 saved\_dragons[] 和 saved\_worms[] 为由 move 存储的，属于相应棋块或棋串的棋子。

- ```
int atari_atari_blunder_size(int color, int move, int *defense, const  
char safe_stones[BOARDMAX])
```

该函数检查着手后是否产生了新的组合攻击并返回其大小（分数）。为着手后认定安全的棋子标记 safe_stones 。

13 影响函数

13.1 影响的概念

13.2 地、模样和区域

13.3 引擎中采用影响函数的地方

13.4 影响和实地

13.5 实地值的细节

13.6 影响函数的核心

13.7 影响算法

13.8 渗透度

13.9 逃跑

13.10 打入

13.11 包围

13.12 影响模块所用的模式

13.13 影响的彩色显示和调试

13.14 使用 `view.pike` 调整影响

13.1 影响的概念

不能被战术攻击，或有战术防守并轮到先手的棋子称为战术活。类似地，不能被战略攻击（死活分析的意义），或有战略防守并轮到先手的棋子称战略活棋。如果要在确定战略状态前使用影响函数，则战术活棋都视为战略活棋。

棋盘上所有战略活棋都作为影响源，其影响向各方向辐射。影响力随距离指数下降。

影响仅当棋盘空时无阻碍扩张，所有战术活棋（无论何方）都作为影响障

碍，如不能穿越对方棋子之间的联络。如一间跳除非其中一子可被吃掉都作为障碍。注意两子之间的联络是否能被打穿并没有多少影响，因为两个方向上都有影响。

从双方的影响计算出每点介于-1.0 和 +1.0 之间的实地值，可视为各方成实地的可能性。

为避免出现假实地，在可能的侵入点加入额外的影响源，如在座子下的 3-3 、在宽边缘扩展的中间和任何大的开放空间的中心。类似地在看起来是实地的但可能侵入的地方如小飞增加额外的影响源。这些侵入依赖于何方先手。

所有额外的影响源，和联络一样由模式数据库的两个文件 `patterns/influence.db` 和 `patterns/barriers.db` 控制。细节在 13.12 Patterns used by the Influence module 描述。

13.2 地、模样和区域

使用影响代码棋盘上的空区域按三种方式分区。一点可分为白或黑的实地、模样和区域。函数 `whose_territory()`、`whose_moyo()` 和 `whose_area()` 可返回按该类归属的一方或空。

- 实地

棋盘上预期终局时成为一方或另一方实际点数的部分视为实地。

- 模样

棋盘上已成实地或如果对方不做侵消容易成实地的部份视为模样。

- 区域

棋盘上一方影响强于另一方的部份视为区域。

一般地实地是模样而模样是区域。为取得这些概念的感性认识，可用选项 `"-m 0x0180"` 装入一个中盘的 `sgf` 文件并检查其结果图形（参见 13.13 Colored display and debugging of influence）。

13.3 引擎中采用影响函数的地方

从影响函数得到的信息在引擎中被多处使用，影响某块在着手生成过程中也多次被调用。影响函数计算细节与调用函数的需要相关。

GNU Go 确定棋串的战术稳定性后，影响模块首次被调用，此时所有活子作为影响源赋强度值 100，结果存储在变量 `initial_influence` 和 `initial_opposite_influence` 中，作为估测棋块强度的重要信息。例如，棋块作为 25 目模样的部分时就立即被估测为活棋，对于更小的模样，棋块要通过 owl 代码进行死活识别（参见 12 介绍基于模式的识别）。模样为 5 目的棋块被估测为弱棋，即使 owl 代码确定它不可能被吃。

为 owl 代码和棋块强弱估算所需的工具，对各棋块计算“逃跑”影响（参见 13.9 Escape）。

所有棋块计算结束后，再次调用影响模块覆盖变量 `initial_influence` 和 `initial_opposite_influence`。当然现在要考虑棋块状态变量。属于死棋块的棋子不作为影响源，且其他棋子根据其所属棋块的强弱调整。

运行的结果是着手评估的最重要的工具。所有模式的辅助函数描述在 9. The Pattern Code 参照影响结果（如 `olib(*)` 等。）实际使用这些结果。进而，`initial_influence` 作为计算着手实地价值的参考。这样每点从存储在 `initial_influence` 的影响强度赋给实地价值。该值意在估计这点成为白或黑地的可能性。

接着，对于函数 `value_moves` 考虑的着手所有着手，再次由函数 `compute_move_influence` 调用影响模块访问该着手后的实地平衡，结果与着手之前的状态相比较。

为计算着手后续价值做一附加的影响计算，描述在 13.5 Details of the Territory Valuation。

引擎使用“influence.c”中一些公共函数列表在 18.4 Utilities “engine/influence.c”。

13.4 影响和实地

本节描述在 `estimate_territorial_value()` 中使用影响函数估计实地。

○ 走“*”价值为一点：

```
OXXX.  
OX.XX  
O*a.X  
OX.XX  
OXXX.
```

由影响函数计算如下：先计算 X 先走在这个局部的任何一处的价值，再计算 ○ 走“*”后在这个局部落子的价值。这两个值的差即该着手的价值。

技术上假定 X 先走是通过对称模式库 `barriers.db` 实现的。安全联络中断影响通道对 ○ 和 X 是有差别的；当然对于 ○ 这样的联络要求更紧密。同样对于 X 有自然后续着手的地方要增加额外的影响源。

在上面的例子中，对称性（着手前）如下：如果 X 是着手方，白影响在“*”阻挡停止，X 实地四点，而如果 ○ 是着手方，则可以假定左面的后续着手为“*”的影响源，在“a”停止，实地三点。

○ 在“*”价值返回 1。当然前提是着手值得使用，并取得价值为 2。GNU Go 3.0 用 `patterns.db` 中的模式来处理，强制明确的后续价值。版本 3.2 则对每一考虑的着手计算独立的后续影响。在上述例子中侵消源作为“*”后续着手在“a”加入，破坏了黑棋所有的实地，故后续价值为 3。

当然某些地方仍然要用到基于模式的后续着手价值。

另一个例子考虑一下局面估计 O 在“*”的着手：

```
OOOXXX
..OX..
..OX..
...*..
-----
```

着手前假定 X 局部先走（O 必须联络），实地如下（小写字母表示各方实地）：

```
OOOXXX
ooOXxxx
o.Oxxx
o...xx
-----
```

O 在“*”着手并假定 X 再走。实地变为（X 假定必须联络）：

```
OOOXXX
ooOXxxx
ooOX.x
oo.O.x
-----
```

实地差别为 4， `influence_delta_territory()` 即返回。接着还有后续价值和反向后续价值。反向后续价值在这里很高，着手视为先手，由确定的模式加入。后续或反向后续价值另一个来源是威胁吃掉并解救棋串。参考函数 `value_move_reaons` 注释使用后续和反向后续价值调整有效着手价值。

下面给出一个吃住棋子的实地价值例子，考虑 O 在“*”着手：

```
XXXXXXXXO
X.OOOOXO
X.O..O*O
-----
```

如前所述假定 X 先走，即吃住白棋时确定的实地价值：

```
XXXXXXXXO
XxyyyyXO
Xxyxxy.O
-----
```

“y”表示“x”实地且为白死子，即需要双倍计点的位置。而 “O”在 “*”着

手则有：

```
XXXXXXXXO
X.OOOOXO
X.OooOOO
-----
```

“X”少了 16 点实地而“O”多了 2 点，合计 19 点。

影响函数计算吃子价值是在 GNU Go 3.2 引入的，之前版本使用有效尺寸启发。有效尺寸是指棋子数加上周围距离该棋串或棋块较其它更近的空点数。这里 “O” 棋串有效尺寸为 6（棋子数）+ 2（内眼）+ 2*0.5（左面两个空点与 X 棋串平分）+ 1*0.33（联络点，三个棋子平分）= 9.33。这里价值是双计的，故为 18.67 接近正确值 18。有效尺寸启发在着手评估中仍被使用，此时影响函数不能容易地给出精确值（如依赖于劫的攻击、攻击威胁等）。

注意本节只描述着手的实地价值，除此之外，GNU Go 使用各种启发确定着手的战略价值（对其他棋子削弱或加强）。另外，影响函数并没有象例子所说调整得很好。但这给出了一个相当好的设计思想。

另一个问题是目前只考虑了安全实地的变化。GNU Go 3.2 和以后版本使用了下节描述的修订启发，确定双方可能的实地。

13.5 实地值的细节

本节讨论黑白影响计算完成后 GNU Go 如何将实地值赋给一点。关键点在于一点有 xx% 的机会成为白地则称白有 xx 点，黑类似。

函数 `new_value_territory` 中的算法大致如下：

设 w_i 是白在一点的影响， b_i 是黑影响，则 $value = ((w_i - b_i) / (w_i + b_i))^3$ （正值表示白实地负值表示黑实地）可以简单衡量，尽管还要差距，但已经可用。

该值跟着要经过几次修正，假定第一次估计是正值。

如果 b_i 和 w_i 都小，就降低之。小的意思取决于该点是否靠近边角，因为边角更容易成地。

接点各点的值降低到其邻点的最小值。可以想知这又是一次中央难成空的实现。这步进一步减少了多面漏风的区域的大小。

最后一些模式确切禁止 GNU Go 在一些点计算实地。例如最后必须回填的假眼。另外还要加上死子的点数。

为调好该方案对与实地评估相关的影响计算进行了修订。包括默认的衰减和一些模式操作的修订。

13.6 影响函数的核心

基本的影响辐射过程可以作为对于邻点和更远点的宽度优先搜索使用一个队列有效地实现。

影响障碍可由模式匹配辅以条件属性和/或辅助函数发现。墙结构、侵入点和侵消点同样可由模式匹配发现。

计算影响点的基本点是棋盘上有许多影响源，其贡献累加得出影响值。目前可以假定棋盘上的活子是影响源，但这并非全部。

函数 `compute_influence()` 包括了一个围绕棋盘的循环，对于每个影响源调用函数 `accumulate_influence()`。这是影响函数的核心。在进入细节前，以下是一强度 100 的独立影响源形成的影响域（衰减为 3.0）：

```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 1 2 3 2 1 0 0 0
0 0 1 3 5 11 5 3 1 0 0
0 1 2 5 16 33 16 5 2 1 0
0 1 3 11 33 x 33 11 3 1 0
0 1 2 5 16 33 16 5 2 1 0
0 0 1 3 5 11 5 3 1 0 0
0 0 0 1 2 3 2 1 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

这些值实际上是浮点数为便于表示取整。这里的概念是值为零时影响域并

未停止。

内部的 `accumulate_influence()` 从影响源开始宽度优先扩散，以队列形式实现。影响的传播顺序和影响只向外扩展的条件保证了所有点只被访问一次并最终结束。在上述例子中，各点访问顺序为：

```
+ + + + + + + + + + +
+ 78 68 66 64 63 65 67 69 79 +
+ 62 46 38 36 35 37 39 47 75 +
+ 60 34 22 16 15 17 23 43 73 +
+ 58 32 14 6 3 7 19 41 71 +
+ 56 30 12 2 0 4 18 40 70 +
+ 57 31 13 5 1 8 20 42 72 +
+ 59 33 21 10 9 11 24 44 74 +
+ 61 45 28 26 25 27 29 48 76 +
+ 77 54 52 50 49 51 53 55 80 +
+ + + + + + + + + + +
```

访问到标记“+”后继续并到棋盘以外。在实际棋盘上有棋子或紧密联络停止影响扩散。这样会中断上图，但传播的主要属性仍保持，即每点只访问一次，且之后不再有影响传播到这点。

13.7 影响算法

采用 `accumulate_influence()` 函数中使用的变量，命 (m, n) 是影响源坐标， (i, j) 是传播时访问的一点。影响按下述方案传播到其八个最近的邻点，和对角点：

对于八个方向 (di, dj) ：

1. 计算向量 di, dj 和 $(i, j) - (m, n)$ 的积 $di * (i - m) + dj * (j - n)$ 。
2. 如果是负数或零，则方向在外部，继续另一个方向。除非在访问影响源，即第一点时要访问所有方向。
3. 如果 $(i + di, j + dj)$ 在棋盘外部或已占用则继续下一方向。
4. 命 S 是 (i, j) 的影响强度，影响从该点传播到 $(i + di, j + dj)$ 为

$P * (1/A) * D * S$, 三类阻尼为:

- 渗透度“P”, 是棋盘点属性, 通常为一, 即无限制传播, 但为使传播停止, 如遇到在如一间跳, 则须在模式匹配时将该类点的渗透度赋值为零。下面详述。
- 衰减度“A”, 是影响源属性, 在不同方向不同。默认为 3 除非对角线上为双倍。修改衰减度可以调整影响源的影响范围大小。
- 方向阻尼“D”, 是 (di, dj) 和 $(i, j) - (m, n)$ 夹角余弦的平方。意义在于阻止影响绕过干扰棋子并在该方向继续作用。这里选择余弦平方是强制性的, 好处在于可以给出一个合理的函数表示“m”、“n”、“i”、“j”、“di”和“dj”, 而无需使用任何三角或平方根计算。访问影响源时按惯例命此因子为一。

影响的典型贡献来自该点和影响源之间最多三个邻点, 三个值简单相加。如前指出的, 访问该点前所有贡献已经计算好。

函数 `compute_influence()` 计算整个棋盘的影响时, 对每一影响源调用 `accumulate_influence()` 一次, 这是独立的, 各影响源的影响贡献相加。

13.8 渗透度

各空点渗透度初始值为一, 占用的为零, 然而为获取有用的影响需要修正之。考虑一下局面:

```
|.....  
|0000..  
|...O..  
|...a.X  (“a”是空点)  
|...O..  
|...000  
|.....O  
+-----
```

角上当然是“O”安全实地, “X”子对其内部影响可以忽略。为阻止“X”影

响泄漏到角部，使用模式匹配（“barriers.db”中模式Barrier1/Barrier2）修改“x”在此点的渗透度为零。“o”在此联络则仍可传播影响。

需要注意的另一情况是对角线影响辐射时渗透度阻尼的计算。对于横纵辐射可以使用辐射通过点（对应方）的渗透度。对角线辐射要额外乘以要穿越的两点渗透度最大值。其原因分析如下：

...X	...X
00..	0da.
..0.	.bc.
..0.	..0.
+-----	+-----

不希望“X”影响由“a”传播到“b”，而 c 和 d 的渗透度都为零，故可实现阻挡。

13.9 逃跑

影响代码的一个应用是计算 `dragon.escape_route` 域，函数 `compute_escape()` 计算过程如下。首先所有点根据对方影响值赋给一个 0 到 4 的逃跑值。

escape_route 域由“surround.c”中的代码修改(参见 13.11 包围)。

对于弱包围棋块除 2，对于普通包围赋值零。

除赋给空点逃跑值外，还赋给同方棋块逃跑值。该值根据棋块状态为 0 到 6，活棋为 6。

接着将距离超过 4 的点（包括同方棋块）带来的影响逃跑值累加，这些点可能由一个长度 4 且不通过对方棋块的路径（且没有更短路径）连接到该棋块。在下例中，累加标记“4”的四个点的影响逃跑值。

$$\begin{array}{cccccccc} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & X & \cdot & \cdot & O \\ \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot & O \end{array}$$

```

X . . . . . . . . . . X . . 1 1 2 3 4 .
X O . O . . . . . O X O 1 O 1 2 3 4 O
X O . O . . . . . X O 1 O 1 . 4 . .
X O . . . X . O O X O 1 . . X . . O
. . . X . . . . . . 1 . X . . . . .
X . . . . X . . . . X . . . . X . . .
. . . . . . . . . . . . . . . . . .

```

由于棋块试图得到安全，会疑惑为什么要由对方棋块调用 `compute_influence()` 关注逃跑。为解释此点，注意 `compute_influence()` 的方向参数，考虑以下局面例子：

```

...XX...
000..000
O.....O
O.....O
-----

```

下面是否成为 `O` 实地取决于何方着手。本例由 `barriers.db` 中的模式辅助实现，如果 `X` 先手则允许 `X` 影响漏进下面，而 `O` 先手则不能。这也是在足够开放的棋盘部分增加影响源的“侵入”模式，根据先手方不同而有不同的处理。

为确定上面 `O` 在三线缺口的实地价值，首先计算初始局面对方（即 `X`）先手的影响。接着 `O` 着手：

```

...XX...
000..0000
O.....O
O.....O
-----

```

再次计算影响，也令 `X` 先手。实地的差别（影响值计算结果）即着手实地价值。

影响计算用于估计逃跑路径实在是特别的，但假定对方先手就不会高估逃跑的可能性，这是有意义的。在逃跑方向上着手之后是对方的先手。

目前逃跑路径技术看起来足够好用但并不能完全依赖。多数错误是过于乐观。

13.10 打入

“breakin.c” 中的代码打入实地需要更深的战术识别是影响模块无法检测的。在影响模块以后运行并对实地评估进行修正。

打入代码使用了“readconnect.c” 中两个公共函数。

- `int break_in(int str, const char goal[BOARDMAX], int *move)`

如果棋串 `str` 方先手且能够联络到区域 `goal[]`（可能有子也可能没子）

返回 WIN。

- `int block_off(int str, const char goal[BOARDMAX], int *move)`

如果棋串 `str` 对方先手且不能联络到区域 `goal[]`（可能有子也可能没子）

返回 WIN。

这些函数是其同类 `recursive_break_in` 和 `recursive_block_off` 的公共前端，互相递归调用。

过程如下：查看所有影响函数检测到的大块（ ≥ 10 ）实地），对于其附近的所有点使用 `readconnect.c` 中的联络距离计算。寻找属于对方的最近的安全棋子。

对于每个这样的棋串 `str` 调用

- `break_in(str, territory)` 如果对方先手，
- `block_off(str, territory)` 如果实地方先手

如果打入成功，即包围失败，就缩小实地，看对方是否还能打入。重复该过程直至实地缩小到对方无法进入。

为 考 察 打 入 代 码 的 操 作 ， 对 于 文 件 “regression/games/break_in.sgf” 带选项 `-d0x102000` 运行 GNU Go，跟踪如下：

```
Trying to break in from D7 to:
E9 (1) F9 (1) G9 (1) E8 (1) F8 (1) G8 (1)
```

```

H8 (1)  G7 (1)  H7 (1)  J7 (1)  H6 (1)  J6 (1)
H5 (1)  J5 (1)  H4 (1)  J4 (1)  H3 (1)  J3 (1)
H2 (1)  J2 (1)
block_off D7, result 0 PASS (355, 41952 nodes, 0.73 seconds)
E9 (1)  F9 (1)  G9 (1)  E8 (1)  F8 (1)  G8 (1)
H8 (1)  G7 (1)  H7 (1)  J7 (1)  H6 (1)  J6 (1)
H5 (1)  J5 (1)  H4 (1)  J4 (1)  H3 (1)  J3 (1)
H2 (1)  J2 (1)
B:F4
  Erasing territory at E8 -b.
  Erasing territory at G3 -b.
  Now trying to break to smaller goal:
F9 (1)  G9 (1)  F8 (1)  G8 (1)  H8 (1)  G7 (1)
H7 (1)  J7 (1)  H6 (1)  J6 (1)  H5 (1)  J5 (1)
H4 (1)  J4 (1)  H3 (1)  J3 (1)  H2 (1)  J2 (1)

```

这表明函数 `break_in` 调用目标为下图标记“a”。该代码试图找出其是否
 联络到 D7 的棋串。

```

  A B C D E F G H J
9 . . . . a a a . . 9
8 . . . . a a a a . 8
7 . . . X O O a a a 7
6 . . . X X X O a a 6
5 . . . . + . . a a 5
4 . . . X . . O a a 4
3 . . . . X . . a a 3
2 . . . . . . O a a 2
1 . . . . . . . . . 1
  A B C D E F G H J

```

找到打入，目标即去除 E9 和 J2 而收缩，再次调用打入代码。

为观察打入的识别过程，可以 `gtp` 模式装载 GNU Go，使用命令：

```

loadsgf break_in.sgf
= black

start_sgftime
=

break_in D7 E9 F9 G9 E8 F8 G8 H8 G7 H7 J7 H6 J6 H5 J5 H4 J4 H3 J3 H2 J2
= 1 E8

finish_sgftime vars.sgf

```

```

=

start_sgfttrace
=

break_in D7 F9 G9 F8 G8 H8 G7 H7 J7 H6 J6 H5 J5 H4 J4 H3 J3 H2 J2
= 1 G7

finish_sgfttrace vars1.sgf

```

这将生成两个包含调用打入代码后变化的 `sgf` 文件。第二个文件“vars1.sgf”中变化较少。

打入代码列出一些找到的打入，结束时调用函数 `add_expand_territory_move` 加入着手目标。

打入代码较慢，每局只改变引擎的几个着手。不管怎样是对程序实力的充分贡献。打入代码在 GNU Go 3.6 级别 10 默认使能，级别 9 默认禁止，实际上这是 GNU Go 3.6 级别 10 与级别 9 的唯一差别。

13.11 包围

棋块怎样算作被包围？

象 Bruce Wilcox 指出的那样，联络对方棋群的几何线经常很重要。很难防止“O”棋块逃跑：

```

.....
.....O....
.X.....X
.X...O...X
.....
.....
-----

```

另一方面，下面棋块很危险：

```

.....
.....
.X.....X
.....O....
.X.....X

```

```
.X...O...X
.....
.....
-----
```

这两个局面的差别是前一个”O” 棋块穿越了顶上两 “X”子的联络线。

“surround.c” 中代码实现了棋块被包围的测试。思想是计算 surround set 即对方棋块包围圈的突出外壳。如果是，则称被包围。

实践中这个方案略有修改。实现中使用各种算法计算距离，对方的棋子中如果找到另一对使其作用消失则不考虑。如以下局面中底下的 “O”子不考虑：

```
O.X.O
.....
.O.O.
.....
..O..
```

另外，给包围圈下第二第三行棋子加点，这考虑了边缘是自然障碍。

为计算角部突出外壳之间距离实现了角度排序的算法。如果一对封闭棋子之间距离较大，则包围状态减低到 WEAKLY_SURROUNDED，如果很大则为 0。

必须解释角度排序。如下小图可能有助：

```
.O.O.
O...O
..X..
O...O
.O.O.
```

排序算法生成：

```
.4.5.
3...6
..X..
2...7
.1.8.
```

即各点按 S-G-O 升序排列。S 为南，G 是（大致的）包围中心，O 是对方棋子位置。

这样排序的必要性在于如果不加排序的话，要估计封闭棋子间的距离只能使用直接的左右角数组，在一些局面结果不一致。想象局面例如点 1、2、3、4、6 和 7 在左数组，只有 5 和 8 在右数组。由于 5 和 8 之间距离大，故棋块识

别为弱包围或未包围。这样情形虽然少，但也需要采用角度排序。

下面局面：

```
O.X.O
.....
.O.O.
```

包围较下局面强：

```
O.XXXXXX.O
.....
.O.....O.
```

在后一情形，包围状态降为 WEAKLY_SURROUNDED。

包围代码用以修改 `dragon2data` 数组的 `escape_route` 域。棋块是 WEAKLY_SURROUNDED 时，`escape_route` 除 2，棋块是 SURROUNDED，`escape_route` 简单赋值为 0。

13.12 影响模块所用的模式

本节解释影响计算所用的模式数据库细节。

首先 “`influence.db`”，中数据库对双方对称匹配。

- “E”

这些模式在给墙这样的形增加额外的影响源。尝试反应其额外的力量。这些模式不用于与实地估计相关的影响计算，但对于更好地估计棋群力量很有用。

- “I”

这些模式在典型的侵入点增加额外的影响源。如果另有类 “s”，则为双方增加额外的影响源，否则只为先手方增加。

“`barriers.db`”中的模式仅匹配“O”先手。

- “A”

“X”子之间的联络停止“O”的影响，必须足够紧密使得“O”即使先手也无法打破。

- “D”

“O”子之间的联络停止“X”的影响，可以较“A”模式松散。

- “B”

指示模式中“O”方标记“Q”的后续着手局面，用于减少实地，如可能的小飞。此外，用于计算“Q”的后续影响（或着手救出的棋子）。

- “t”

这些模式指示一方不能城市迪的点，如假眼。这些点调用模式操作属性中的辅助函数 `non_oterritory` 或 `non_xterritory` 获取。

侵消模式（“B”）较上述描述的更为有力，在识别弱形（为对方在可能打入的地方加入一个侵消源）。它的一个负面影响是一个坏的“B”模式，如条件属性错，会在回归测试集中产生 5 到 10 个失败。

影响模式一般可以使用自动辅助函数属性。条件属性（在一般的“O”、“o”、“X”和“x”以外）有“Y”和“FY”。标记“Y”的模式只能用于与实地估计相关的影响计算，“FY”模式只用于其他影响计算。

目前影响模式的操作属性只用于如上所述的非实地模式，作为解决“B”模式中后续影响的一个手段。

考察这个手段的必要性，考虑以下情形：

```
..XXX
.a*.O
.X.O.
..XXO
```

（想象左面是“X”的实地。）

“O”在“*”着手有一自然后续着手“a”，因此计算“*”后续影响时，“O”在“a”有额外的影响源可破坏左面许多黑实地。这可以给出一个大的后续价值，因此着手“*”被视为先手。

“X”在“a”应，即阻止侵入又威胁吃掉“*”。这样的情形很普通。

但是需要一个额外的条件确定何时可以使用侵入模式，这由错用的操作行完成： 额外行

```
>return <condition>;
```

加入模式。如果侵入不能先手阻止时为 `true`。在上例，相关的侵入模式操作行形式为：

```
>return (!xplay_attack(a,b));
```

“b”表示“*”子。实际上，几乎所有的后续条件都类似这样。

13.13 影响的彩色显示和调试

有许多方式可以获得影响计算的详细信息，彩色 `xterm` 或 `rxvt` 窗口可以显示影响的彩图。

生成图形有两个选项：

- “-m 0x08” or “-m 8”

显示出使影响计算的图形。执行两次，第一次是在 `make_dragons()` 运行前，第二次是在其后。差别在于第二次考虑了死棋块。被战术吃住的棋串两次都考虑。

- “--debug-influence location”

显示给定为之着手后的影响图形。该选项的一个重要限制是它仅对于着手生成器考虑的着手有效。

其他选项控制在这些情形下生成哪个图形。至少要指定以上选项之一和下属选项之一才能生成输出。

下述选项必须与上述两选项之一组合使用，否则不能打印图形。如打印影响图形，要组合 `0x08` 和 `0x010`，并使用选项“-m 0x018”。

- “-m 0x010” or “-m 16”

显示实地/模样/区域彩色显示。

实地：橙色

模样：黄色

区域：红色

该特性获取 GNU Go 对于影响区域的第一印象非常有用。

- `"-m 0x20" or "-m 32"`

显示白黑数字化影响值。这分为两个独立的图形，第一个为白棋，第二个为黑棋。注意影响值由浮点数表示，在图中取整。

- `"-m 0x40" or "-m 64"`

生成两个图形显示黑白在棋盘影响的渗透度。

- `"-m 0x80" or "-m 128"`

显示黑白通过棋盘影响源的力量。可以看到每个活子（力量取决于这些棋子）和模式贡献的源。

- `"-m 0x100" or "-m 256"`

显示影响源通过棋盘传播的衰减度。低衰减意味着影响较远。

- `"-m 0x200" or "-m 512"`

显示 GNU Go 的实地评估。每个点显示-1.0 到+1.0 的值（或-2 或+2 如果该点是死子）。正值表示白实地，这样-0.5 表示黑有 50%机会成实地。

最后，调试选项 `"-d 0x1"` 在 `DEBUG_INFLUENCE` 打开。对每一匹配的影响模式给出信息。不幸的是很多时候信息太多并过于冗长不适合浏览。然而，一旦通过 `"-m 0x80"` 发现一个影响源看起来错误，调试信息有助于快速找出相应的模式。

13.14 使用 view.pike 调整影响

回归目录中一个有用的程序是 `view.pike`。运行时需要 Pike，可以在 <http://pike.ida.liu.se/> 下载。

测试项 `"endgame:920"` 在 GNU Go 3.6 失败，下面解释如何修正。

开始在测试项 `endgame:920` 启动 `view.pike`，如在回归目录运行 `pike view.pike endgame:920`。

第一次可以看到 P15 单官价值最高为 0.17 点而正确着手 C4 略低为

0.16。实际问题当然是 C4 是满点价值应为 1.0。

点击 C4 获取着手目标和着手评估信息列表。所有都正确除了实地变化 0.00 而实际应为 1.00。

选择"delta territory for..." 按钮并点击 C4 确认。现在 B5 应标记实地变化，但没有。

下一步进入影响调试工具，按"influence"按钮，接着"black influence, dragons known,"和"territory value." 这显示如果黑局部先手（这样“黑影响”）预期的实地。这里 B5 错为 1.0 点的白实地。

可以按"after move influence for..." 点击 C4 比较白 C4 着手（仍假设之后黑局部先手）后的实地。看起来和预期一样，实地差别为 0，但这里是正确的，B5 是 1.0 点白实地。

这个问题最直接的解决方法是增加一个非实地模式，表示如果黑先手白在 B5 不能成实地。非实地模式在"barriers.db"。

```
Pattern Nonterritory56
```

```
...
```

```
X.O
```

```
?O.
```

```
:8,t
```

```
eac
```

```
XbO
```

```
?Od
```

```
;oplay_attack(a,b,c,d,d)
```

```
>non_xterritory(e);
```

在这些模式中都假设“O”先手这样“X”在 B5 不能成实地（模式中的“e”）。

现在需要更小心因为“O”在“a”着手和“X”在“b”切断，会出现“O”在“d”防

守，允许“x”在“c”分断，可能使得非实地假设失效。这样做到完全精确很难，但以上条件相当直观并能保证“a”大多数时候安全，尽管不是所有时候。

14 模样的另一方案：Bouzy 的 5/21 算法

14.1 模样历史

14.2 Bouzy 的 5/21 算法

文件“score.c”包含实地和模样的另一种计算方法。这些算法在 `estimate_score()` 使用但在引擎的其他部分一般不使用，实地、模样和区域的概念使用影响代码重新实现（参见 13.2 Territory, Moyo and Area）。引擎中唯一使用此代码的函数 `estimate_score()` 可以很容易地用基于影响代码的函数如 `influence_score()` 替代。

14.1 模样历史

在 GNU Go 2.6 扩展使用了 Bruno Bouzy 的论文中的算法，论文在 <ftp://www.joy.ne.jp/welcome/igs/Go/computer/bbthese.ps.Z>。该算法特征函数从盘上活棋开始，执行“n”次扩张操作，再执行“m”次侵蚀操作。n=5、m=21 时称 5/21 算法。

Bouzy 5/21 算法有趣之处在于它与人类的实地概念相当吻合。该算法在 GNU Go 3.4 的函数 `estimate_score` 中仍然使用，故将 5/21 算法与实地一词关联，类似地将模样和区域 n 分别与 5/10 和 4/0 算法关联。

该算法的主要问题在于它无法调整。目前确定模样和实地的方法在“influence.c”（参见 13. Influence Function）。实地、模样和区域的概念已经用影响代码重新实现了。

Bouzy 算法在文件“scoring.c”简单地重新实现并为 GNU Go 3.4 在

估计评分时使用。

GNU Go 2.6 的原 “moyo.c” 中所有特性并未全都重新——一部分增量未实现——但重实现后可读性更强。

14.2 Bouzy 的 5/21 算法

Bouzy 算法源于 Zobrist 关于确定实地的先期成果和计算机视觉的思路。该算法基于两个简单操作：扩张 (DILATION) 和侵蚀 (EROSION)。执行 5 次扩张和 21 次侵蚀确定实地。

为获得该算法的一点感性认识，取一中盘前期的棋局局面，在 RXVT 窗口用 “-m 1” 选择显示彩色，算法认定的实地趋同于人类高手的判断。

执行算法前须“去除”死子 (`dragon.status==0`)。

参考 Bouzy 论文的 86 页，在棋盘上从获取高值（如黑 +128，白 -128 for white）的函数开始，从 0 到空点重复以下操作：

扩张：对于每个棋盘点，如果该点 ≥ 0 ，且不与 < 0 的点相邻，则为增加 > 0 的邻点数。另一方也如此：如果该点 ≤ 0 ，且不与 > 0 的点相邻，则减去 < 0 的邻点数。

侵蚀：对于每个 > 0 （或 < 0 ）的点，减去（或增加） ≤ 0 （或 ≥ 0 ）的邻点数。直至为零。算法为：5 次扩张，接着 21 次侵蚀。设 n 是扩张次数，则侵蚀次数应为 $1+n(n-1)$ 以保证隔离的棋子不产生实地。这样 4/13 也能工作，但经常不是很好，如 6 路上的实地。

例如从单关开始

128 0 128

1 次扩张：

1 1

1 128 2 128 1

1 1

2 次扩张:

1 1

2 2 3 2 2

1 2 132 4 132 2 1

2 2 3 2 2

1 1

3 次扩张:

1 1

2 2 3 2 2

2 4 6 6 6 4 2

1 2 6 136 8 136 6 2 1

2 4 6 6 6 4 2

2 2 3 2 2

1 1

以此类推

下一个，同样的例子

3 次扩张 1 次侵消:

2 2 2

0 4 6 6 6 4

0 2 6 136 8 136 6 2

0 4 6 6 6 4

2 2 2

3 次扩张 2 次侵消：

1
2 6 6 6 2
6 136 8 136 6
2 6 6 6 2
1

3 次扩张 3 次侵消：

5 6 5
5 136 8 136 5
5 6 5

3/4 :

3 5 3
2 136 8 136 2
3 5 3

3/5 :

1 4 1
136 8 136
1 4 1

3/6 :

3
135 8 135
3

3/7 :

132 8 132

解释为 1 点实地。

15 棋盘库

15.1 棋盘数据结构

15.2 棋盘数组

15.3 增量棋盘数据结构

15.4 棋盘函数

GNU Go 引擎的基础一个包含了许多高效的棋盘操作例程的函数库。棋盘库称“libboard”，c 可以为仅需要一个基本棋盘而不需要人工智能能力的程序使用，如“模式 s/joseki.c”可从 SGF 文件编译定式模式数据库。包括以下文件：

- “board.c”

基本棋盘代码。使用增量算法保持对棋串和气数的跟踪。

- “hash.c”

位置 hash 代码。

- “globals.c”

其他文件中使用的全局变量。也包含了引擎其他中部分的全局变量。

- “sgffile.c”

实现输出 SGF 文件格式。

- “showbord.c”

打印棋盘。

- “printutils.c”

打印棋盘和其他的实用工具

使用棋盘库象使用整个引擎一样必须包含“liberty.h”，但不能使用那里声明的全部函数，也就是说那些函数是引擎的部分，而不是棋盘库。心须使用 libboard.a 连接。

15.1 棋盘数据结构

棋盘的基本数据结构与 17.3 棋盘状态结构中的 `board_state` 紧密相关，为效率原因都存储在全局变量中，最重要的如下：

```
int          board_size;
Intersection board[MAXSIZE];
int          board_ko_pos;

float        komi;
int          white_captured;
int          black_captured;
```

下面的声明已转移到“hash.h”

```
Hash_data    hashdata;
```

结构 The description of the Position struct is applicable to these variables also, so we won't duplicate it here. 所有这些变量为效率原因都为全局变量。在这些变量后面，还有许多私有数据结构，实现了棋串的更多的操作、气和其他属性（参见 15.3 增量棋盘数据结构）。变量 `hashdata` 包含了当前位置的 hash 值（参见 11.2 位置 Hash 排序）。

这些变量是递增产生的，不能直接操作。它们是可读的，但只能用下节描述的函数写。如果直接写，增量数据结构就不能同步，会导致崩溃。

15.2 棋盘数组

GNU Go 用一个一维数组 `board` 表示棋盘。为一些用途使用了二维的参数 (i, j) 索引。

数组 `board` 包括棋盘周围的盘外标志。下图以 `MAX_BOARD = 7` 为例示意

1 维和 2 维索引的对应关系。

i\j	-1	0	1	2	3	4	5	6
-1	0	1	2	3	4	5	6	7
0	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31
3	32	33	34	35	36	37	38	39

```

4    40  41  42  43  44  45  46  47
5    48  49  50  51  52  53  54  55
6    56  57  58  59  60  61  62  63
7    64  65  66  67  68  69  70  71  72

```

为转换 1 维指针 pos 和 2 维指针 index (i, j) , 提供了宏 POS、 I 和 J

如下:

```

#define POS(i, j)    ((MAX_BOARD + 2) + (i) * (MAX_BOARD + 1) + (j))
#define I(pos)       ((pos) / (MAX_BOARD + 1) - 1)
#define J(pos)       ((pos) % (MAX_BOARD + 1) - 1)

```

所有不在棋盘内的 1 维指针赋给盘外标志 GRAY。这样如果 board_size

和 MAX_BOARD 都为 7 时, 如下所示:

```

i\j   -1  0  1  2  3  4  5  6
-1    #  #  #  #  #  #  #  #
0     #
1     #
2     #
3     #
4     #
5     #
6     #
7     #  #  #  #  #  #  #  #

```

标注“#”的点赋值 GRAY。如果 MAX_BOARD = 7, board_size = 5:

```

i\j   -1  0  1  2  3  4  5  6
-1    #  #  #  #  #  #  #  #
0     #                #  #
1     #                #  #
2     #                #  #
3     #                #  #
4     #                #  #
5     #  #  #  #  #  #  #  #
6     #  #  #  #  #  #  #  #
7     #  #  #  #  #  #  #  #

```

棋盘上的移动方向定义为宏 SOUTH、WEST、NORTH 和 EAST:

```

#define NS          (MAX_BOARD + 1)
#define WE          1
#define SOUTH(pos)  ((pos) + NS)
#define WEST(pos)   ((pos) - 1)
#define NORTH(pos)  ((pos) - NS)
#define EAST(pos)   ((pos) + 1)

```

还包括缩写的宏 SW、NW、NE、SE、SS、WW、NN、EE 表示两步移动。

棋盘上一点移动到相邻点和尖点保证产生有效的索引值, 如果是落在盘外

则颜色值为 GRAY。盘外检查的宏有两个:

```

#define ON_BOARD(pos) (board[pos] != GRAY)
#define ON_BOARD1(pos) (((unsigned) (pos) < BOARDSIZE) && board[pos] != GRAY)

```

前一个在算法中使用，后一个在异常测试中使用。

一维棋盘数组的好处在于性能，使用一个变量表示棋盘位置，许多函数的参数减少了。另外，采用 1 维位置 1 步计算的采用 2 维位置就需要 2 步计算。

例如，需要 pos 右上位时只需要 NORTH(EAST(pos)) 而不是 (i+1, j-1)。

注意：2 维位置 (-1, -1) 表示放弃或无点时，映像到 1 维位置 0 而不是 -1，且用 NO_MOVE 或 PASS_MOVE 表示。

多向的循环可以直接写成：

```
for (k = 0; k < 4; k++) {
    int d = delta[k];
    do_something(pos + d);
}
```

以下常量在全盘和数组与全盘位置 1-1 对应时常用。

```
#define BOARD_SIZE ((MAX_BOARD + 2) * (MAX_BOARD + 1) + 1)
#define BOARD_MIN (MAX_BOARD + 2)
#define BOARD_MAX (MAX_BOARD + 1) * (MAX_BOARD + 1)
```

BOARD_SIZE 是 1 维棋盘数组的实际大小，BOARD_MIN 是棋盘上点的一个索引，BOARD_MAX 是棋盘上点最后一个索引加一。

需要转换棋盘，在每个方向执行某些函数时使用以下两个函数。

```
int m, n;
for (m = 0; m < board_size; m++)
    for (n = 0; n < board_size; n++) {
        do_something(POS(m, n));
    }
```

或：

```
int pos;
for (pos = BOARD_MIN; pos < BOARD_MAX; pos++) {
    if (ON_BOARD(pos))
        do_something(pos);
}
```

15.3 增量棋盘数据结构

除了全局棋盘状态，“board.c”中的算法还实现了保持追踪每个棋串以下

信息的增量更新方法：

- 对局方
- 棋子数量
- 起点，定义为最小的 1 维参考点。
- 棋子列表
- 气数
- 气点，如果气太多这个表是截断的。
- 相邻棋串数量
- 相邻棋串列表

基本数据结构：

```
struct string_data {
    int color;                /* Color of 棋串, BLACK or WHITE */
    int size;                 /* Number of stones in 棋串. */
    int origin;               /* Coordinates of "origin", i.e. */
                              /* "upper left" stone. */
    int liberties;            /* Number of liberties. */
    int libs[MAX_LIBERTIES];  /* Coordinates of liberties. */
    int neighbors;            /* Number of neighbor 棋串 s */
    int neighborlist[MAXCHAIN]; /* List of neighbor 棋串 numbers. */
    int mark;                 /* General purpose mark. */
};
```

```
struct string_data string[MAX_STRINGS];
```

应当清楚几乎所有信息都存储在数组 `string` 中，为获取棋盘的映像：

```
static int string_number[BOARDMAX];
```

包括了指向 `string` 数组的索引，其中的信息只对非空的节点有效，所以有必要先验证 `board[pos] != EMPTY`。

`string_data` 结构中没有棋子坐标，在另一个数组中存储：

```
static int next_stone[BOARDMAX];
```

该数组实现了棋子的循环链表。每个节点包含了一个指向其他节点的指针（也可能指向自身）。从棋盘上一个棋子开始，循着这些指针可以遍历整个棋串。数组“`string_number`”中该信息对于棋盘上的空点无效。这个数据结构

所需内存是固定的，加入一个棋子或者连接两个棋串都非常简单。

代码还用到了临时变量：

```
static int ml[BOARDMAX];
static int liberty_mark;
static int string_mark;
static int next_string;
static int strings_initialized = 0;
```

数组和 `liberty_mark` 用作标记棋盘上的气，如为了避免重复计算气数。原理是如果 `ml[pos]` 与 `liberty_mark` 相同则该 `pos` 标记。将 `liberty_mark` 增量可操作新串，该值不允许递减。

数据结构 `string_data` 中的 `mark` 与 `string_mark` 的关系也如此。这是用来标记单独的棋串。

`next_string` 给出数组 `string` 下一可用入口的数字。接着 `strings_initialized` 设定为 1，提示更新所有数据结构。对于数组 `board` 中的棋盘位置，可调用 `incremental_board_init()` 完成。没有必要每次都完全完成，因为其他函数也会需要这些信息时也会去完成。

这部分代码最有趣的地方是落子和移除时数据结构的更新。为理解加入棋子的操作需要首先知道悔棋如何操作。其方法是一旦有信息改变，旧值就压入堆栈，存储其地址和数值。堆栈从如下结构中建立：

```
struct change_stack_entry {
    int *address;
    int value;
};

struct change_stack_entry change_stack[STACK_SIZE];

int change_stack_index;
```

用如下宏操作：

```
BEGIN_CHANGE_RECORD()
PUSH_VALUE(v)
POP_MOVE()
```

调用 `BEGIN_CHANGE_RECORD()` 在 `address` 域存储空指针表示一个新

着手开始。前述 `PUSH_VALUE()` 存储值及其地址。假设所有信息都压入堆栈，悔棋就只是调用 `OP_MOVE()` 了，即反向将数值赋给指针直到遇到空指针。这样的描述略微简单了，因为这样的堆栈只能存储整型值，而我们需要存储棋盘的变化。这样我们有两个并行的堆栈，一个存储整型值另一个存储 `Intersection` 值。

处理落子时，首先如果有提子的话，先提掉对方棋串。在这步我们必须压入棋盘值和指向被提棋子的 `next_stone` 指针，然后更新被提棋串相邻棋串的气和相邻列表。我们不需要压入被提棋串“string”入口的所有信息，因为着手压入后它们返回并继续被使用。

落子后我们需要处理三种情形：

1. 孤子，即没有友邻
2. 有一个友邻
3. 有两个或更多友邻

第一种情形最容易。用 `next_string` 给出的数值建立一个新棋串，然后递增改变量。棋串大小为 1，`next_stone` 直接指向自身，气数可以通过察看四个方向上的空点确定。同样也可能发现相邻棋串，减少一气并增加一个邻串。

在二种情形下不建立新串而是在邻串上扩展新子。这包括把新子链接到环链，如果需要的话更新串头，更新气数和邻串。新子的邻串的气数和邻串同样需要更新。

最后第三种情形，需要链接已存在的棋串。为不再存储过多的信息，建立一个新棋串并让其链接相邻棋串。这样所有信息都保留在数组“string”中，同样包括移除的棋串。这里保持跟踪气数和邻串有些复杂，因为这由多个连接的棋串共享。更好地利用标记可使其变得更直接。

常用构造：

```
pos = FIRST_STONE(s);
do {
    ...
```

```
pos = NEXT_STONE(pos);
} while (!BACK_TO_FIRST_STONE(s, pos));
```

恢复一次标记“s”的棋串 pos 存储坐标。一般地 pos 用作棋盘坐标，“s”用作数组 string 的索引或指向数组 string 中一个入口的指针。

15.4 棋盘函数

识别，经常在计算机游戏中成为**搜索**，是 GNU Go 的基础过程。这个过程中生成假想的棋盘以确定一个问题的答案，诸如：“这些子是否活子”。因为是假想的位置，所以重要的是能够恢复之最终回到现在的棋盘。在识别中要维护一个堆栈。尝试一个着手使用 trymove 或其变体 tryko。这个函数将当前棋盘压入堆栈并落子。堆栈指针 stackp 递增存储保持位置的跟踪。函数 popgo() 弹出着手堆栈，递减 stackp 并恢复上一手的变化。

每个成功的 trymove() 都需要匹配一个 popgo()。这样正确使用该函数的方法：

```
if (trymove(pos, color, ... )) {
    ... [potentially lots of code here]
    popgo();
}
```

这里 komaster 仅当前一手提劫时使用，以避免 GNU Go 中的战术和 owl 别代码进行多余的识别。（参见 11.4 劫争操作）。

- `int trymove(int pos, int color, const char *message, int str, int komaster, int kom_pos)`

如果 (pos) 对于 color. 是有效着手返回 TRUE。此时，将棋盘压入堆栈并落子，递增 stackp。如果识别代码记录了识别变化（选项“--decide-string”或“-o”），则在 SGF 文件中加入 *message 注释。注释同样引用 str 如果非零。Komaster 和劫争位置变量在其他地方描述（参见

11.4 劫争操作)。

- `int tryko(int pos, int color, const char *message, int komaster, int kom_pos)`

`tryko()` 将当前位置压入堆栈, 按 `pos` 和 `color` 落子。即使是非法的提劫也允许。这是假定本方寻劫对方应劫后的继续。在上述前提下着手合法有效返回 1, 因自杀为非法则返回 0。

- `void popgo()`

弹出着手堆栈。本函数在成功的 `trymove` 或 `tryko` 后都最终要调用重装棋盘位置, 从而恢复了调用 `trymove/tryko` 产生的变化并使棋盘恢复到调用前的状态。**注意:** 如果 `trymove/tryko` 返回 0, 即所尝试的着手非法, 则不应调用 `popgo`。

- `int komaster_trymove(int pos, int color, const char *message, int str, int komaster, int kom_pos, int *new_komaster, int *new_kom_pos, int *is_conditional_ko, int consider_conditional_ko)`

`trymove/tryko` 的变体, 提劫(条件的和无条件的)必须跟一个 `komaster` 组织(参见 11.4 劫争操作)

可以看到, `trymove()` 落子很容易(用 `popgo()`)回溯, 在 GNU Go 分析棋盘位置时每次对奕会调用上千次。相比之下函数 `play_move()` 落子是固定的, 但也可以恢复, 如对手悔棋。

- `void play_move(int pos, int color)`

落子, 如果要测试合法性要先调用 `is_legal()`。该函数严格遵循下述算法:

1. 在棋盘上落子。
2. 如果对方相邻棋串没有气就提掉并增加提子数。
3. 如果落子所在棋串没有气就提掉并增加提子数。

尽管落子是“固定的”但一般也可以 `undo_move()` 恢复，但开销要比恢复一个临时落子多得多。棋谱有一些限制，在一些情况下落子后无法恢复。

- `int undo_move(int n)`

恢复“n”个固定落子，成功返回 1，失败返回 0。如果不能恢复“n”手，则一手也不恢复。

其他棋盘函数在 18.3 棋盘工具描述。

16 在内存中操作 SGF 树

SGF (Smart Game Format) 是一种用来存储许多种游戏对局的文件格式，包括国际象棋和围棋。这里对文件格式标准不做介绍，关于文件格式的确切定义可参考：<http://www.red-bean.com/sgf/>。

GNU Go 包含了一个库以 SGF 格式来处理内存中的围棋对局记录，和读写 SGF 文件。这个库 libsgf.a 在 sgf 下。使用 SGF 例程，要引用“sgftree.h”。

所有对局记录都按树型存储，每个节点代表对局的一个状态，一般是每手棋之后的状态。每个节点包括 0 或多个属性。对局树中的变化用子节点表示。第一个字节节点是主变化。

用来编码对局树的 SGFNode 和 SGFProperty 类型结构定义如下：

```
typedef struct SGFProperty_t {
    struct SGFProperty_t *next;
    short name;
    char value[1];
} SGFProperty;
typedef struct SGFNode_t {
    SGFProperty *props;
    struct SGFNode_t *parent;
    struct SGFNode_t *child;
    struct SGFNode_t *next;
} SGFNode;
```

SGF 树的每个节点都存储在一个 SGFNode 结构中，其中有一个指针指向名为 props 的属性链表，一个指针指向 child 子节点，其中包括了所有的变化，并由指针 next 链接。parent 指向其父节点。

SGF 属性为 SGFProperty 结构，由 next 链接，每个属性都有一个属性 name 类型为 short int。属性名称列表在文件“sgf_properties.h”中。

一些属性可能包括了一个参数，可以是整型、浮点、字符或者字符串。这些值可通过特殊的函数存取。

16.1 SGF 树数据类型

有时我们想记录正在进行的棋局或同样简单的记录而不需要高级的树结构，这时可以用下面简化的 SGFTree 接口：

```
typedef struct SGFTree_t {  
    SGFNode *root;  
    SGFNode *lastnode;  
} SGFTree;
```

SGFTree 包含指向 SGF 树根节点和最后访问的节点的指针，大多数时候这时棋局最后一手。

大多数操作 SGFTree 的函数可以同样操作 SGFNode，但只操作树的当前节点。

带参数 tree 和 node 的函数的操作顺序：

1. node 非空
2. tree->lastnode 非空
3. 对于当前节点

17 GNU Go 的 API

如果需要自行编写 GNU Go 接口，或者利用 GNU Go 引擎创建自己的程序，本章是非常有趣的。

纵观 GNU Go，其包括两部分：GNU Go 引擎和使用引擎的程序（用户接口）。两者链接为一个库，目前的程序实现了下列用户模式：

- 在 ASCII 终端上可操作的交互式棋盘
- 自对弈——GNU Go 自己用自己对弈
- 复盘——用户可研究 SGF 文件中的棋谱
- GMP — Go Modem Protocol，双机自动对弈的协议
- GTP — Go Text Protocol，一个更为通用的围棋协议，参见 19GTP 协议

GNU Go 引擎可在其它程序中使用，如与 GNU Go 一同发布的“interface/debugboard/”中的“debugboard”，提供用户调用 SGF 文件并交互地查看某局面的不同属性，如棋群状态和眼状态。

本章的目的在于说明象 debugboard 这样的程序是如何与 GNU Go 引擎交互的。

图 1 描述了一个使用 GNU Go 引擎的程序。

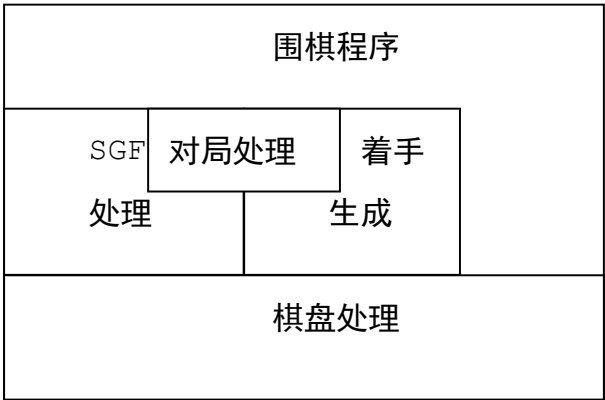


图 1：使用 GNU Go 引擎的程序结构

基础是一个名为 `libboard.a` 的库，提供有效的围棋棋盘操作，包括着手规则检查、棋串增量处理和高效的围棋局面 `hash` 方法。

在此之上，有一个名为 `libsgf.a` 的库辅助实现 `smart go` 文件、SGF 文件，在内存和文件中操作对局树。

GNU Go 代码中的主要部分是着手生成库，对给定的局面生成一个着手。这部分引擎还可以用来操作局面、增删棋子、做战术和战略识别以及查询有效着手等，编译在 `libengine.a`。

对局操作代码帮助程序员保持着手的跟踪。对局可以存储到 SGF 文件并在以后读取，这些代码编译在 `libengine.a`。

围棋程序的任务则是提供一个图形化或者非图形化的用户接口，使用户得以与引擎交互。

17.1 如何在你自己的程序中使用引擎：开始

17.2 引擎中的基本结构

17.3 棋盘状态结构

17.4 局面函数

17.5 对局操作

17.1 如何在你自己的程序中使用引擎：开始

为在程序中使用 GNU Go 引擎，要包含文件“`gnugo.h`”，它描述了所有的公共 API，另一个文件“`liberty.h`”描述了引擎的内部接口。如果要在引擎中创建一个新模块，如着手建议，必须同时包含这个文件。本节只描述公共接口。

使用引擎前必须先调用函数 `init_gnugo()` 初始化引擎。它带有一个参

数：引擎可以用做内部 hash 表的兆字节内存。此外引擎要使用几兆字节做其他用途，包括描述棋群（气、死活状态等）、眼位等等。

17.2 引擎中的基本结构

gnugo.h 中有几个所有地方都用到的基本定义。其中最重要的就是棋子颜色定义，棋盘上每个点都由以下状态之一表示：

颜色	取值
EMPTY	0
WHITE	1
BLACK	2

宏 `OTHER_COLOR(color)` 用来给出参数代表的棋子颜色的对手方颜色，该宏仅当颜色为 `WHITE` 或 `BLACK` 时可以使用，不能用于空点或边界点。

GNU Go 使用棋盘的两种不同的表示方法，大多数情况都使用一维表示，少数情况使用二维表示（参见 15 棋盘库）。一维棋盘在 GNU Go 3.2 前引入，二维棋盘可以追溯到 Man Lung Li 1995 前写的旧程序。API 仍然使用一维棋盘，所以 API 函数自 GNU Go 3.0 后没有多少变化。

17.3 棋盘状态结构

引擎中一个基本的数据结构是 `board_state` 结构，在引擎内部“liberty.h”定义。

```
typedef unsigned char Intersection;

struct board_state {
    int board_size;

    Intersection board[BOARDSIZE];
    int board_ko_pos;
    int black_captured;
    int white_captured;
```

```

Intersection initial_board[BOARDSIZE];
int initial_board_ko_pos;
int initial_white_captured;
int initial_black_captured;
int move_history_color[MAX_MOVE_HISTORY];
int move_history_pos[MAX_MOVE_HISTORY];
int move_history_pointer;

float komi;
int move_number;
};

```

Intersection 存储 EMPTY、WHITE 或 BLACK，声明为 unsigned char 使其在存储和访问时间上更有效。棋盘状态包含了 Intersection 数组表示棋盘。棋谱存储在该结构中，同时还包括劫争点（如果上一手不是提劫则为空）、贴目、提子数和棋谱开始的初始状态。

17.4 局面函数

引擎中所有处理局面的函数带有前缀 gnugo_，这些函数仍使用棋盘的二维表示（参见 15.2 棋盘数组）。下面是一个来自“gnugo.h”的完整列表。

- void init_gnugo(float memory)

初始化 gnugo 引擎，只需要调用一次。

- void gnugo_clear_board(int boardsize)

清除棋盘。

- void gnugo_play_move(int i, int j, int color)

落子并启动时钟

- int gnugo_play_sgfnodetree(SGFNode *node, int to_move)

在棋盘上从 SGF 节点开始生成着手并落子。返回下步的颜色。

- int gnugo_play_sgftree(SGFNode *root, int *until, SGFNode **curnode)

从根节点开始直到所有着手。返回下一手的颜色。

- int gnugo_sethand(int handicap, SGFNode *node)

sethand，设置让子并返回让子数，更新 sgf 文件

- `float gnugo_genmove(int *i, int *j, int color, int *resign)`

`genmove()` 的接口 (7)

- `int gnugo_attack(int m, int n, int *i, int *j)`

`attack()` 的接口

- `int gnugo_find_defense(int m, int n, int *i, int *j)`

`find_defense()` 的接口

- `float gnugo_estimate_score(float *upper, float *lower)`

将估算的最高和最低评分存储到 `*upper`、`*lower` 并返回平均值。正值白领先。在计算最高值时所有争议的棋串算作白方，计算最低值时归为黑方。

17.5 对局操作

上节所描述的函数已经满足编写一个全功能的围棋程序的所需了，但为使操作更为简单，还定义了一套处理对局的函数。

描述对局的数据结构是：

```
typedef struct {
    int      handicap;

    int      to_move;          /* whose move it currently is */
    SGFTree  game_record;      /* Game record in sgf format. */

    int      computer_player;  /* BLACK, WHITE, or EMPTY (used as BOTH) */

    char      outfilename[128]; /* Trickle file */
    FILE      *outfile;
} Gameinfo;
```

让子的意义明显。`to_move` 是下一手颜色。

SGF 树 `game_record` 用来存储整个棋局的所有着手，包括头节点，其中有贴目和让子数。

如果一方或双方的对手是计算机，使用 `computer_player`，否则忽略。

GNU Go 可以使用一个累积文件连续存储进行中的对局棋谱，该文件可以包

含引擎的内部状态如各局面着手目标和估值。文件名存储在 `outfilename` 中，文件指针存储在 `outfile`。如果不使用累积文件 `outfilename[0]` 为空字符，`outfile` 为空。

17.5.1 Gameinfo 函数

引擎所有操作对局信息的函数前缀为 `gameinfo_`，以下是一个完整列表来自“gnugo.h”：

- `void gameinfo_clear(Gameinfo *ginfo, int boardsize, float komi)`

初始化 `Gameinfo` 结构

- `void gameinfo_print(Gameinfo *ginfo)`

打印 `gameinfo`

- `void gameinfo_load_sgfheader(Gameinfo *gameinfo, SGFNode *head)`

从 `sgf` 结构读取头信息并设置适当变量。

- `void gameinfo_play_move(Gameinfo *ginfo, int i, int j, int color)`

对局落子。如果着手合法返回 1，并落子，否则返回 0。

- `int gameinfo_play_sgftree_rot(Gameinfo *gameinfo, SGFNode *head, const char *untilstr, int orientation)`

在 `SGF` 树落子。循着主变化操作棋盘。返回下一手颜色。`Head` 是一个 `sgf` 树。`Untilstr` 是可选的棋串，格式如“L12”或“120”表示操作到该着手或着手数为止。在测试时这点就是需要开始检查的着手。

- `int gameinfo_play_sgftree(Gameinfo *gameinfo, SGFNode *head, const char *untilstr)`

与上一函数相同，使用标准的方位。

18 工具函数

18.1 通用工具

18.2 打印工具

18.3 棋盘工具

18.4 “engine/influence.c” 中的工具

本章描述 GNU Go 引擎调用的一些工具。

18.1 通用工具

“engine/utils.c”中的工具函数许多是自动辅助函数的基础（参见 9.7 自动辅助函数）

- `void change_dragon_status(int dr, int status)`

修改 `dr` 棋块所有棋子的状态。

- `int defend_against(int move, int color, int apos)`

检查 `move` 着手是否阻止对方在 `apos` 着手。

- `int cut_possible(int pos, int color)`

如果 `color` 可在 `pos` 分断或禁止通过 `pos` 联络返回 `true`。该信息由 `find_cuts()` 在联络数据库使用 B 模式收集。

- `int does_attack(int move, int str)`

如果 `move` 着手攻击 `str` 返回 `true`。即可吃住尚未净死的棋串 `str`。

- `int does_defend(int move, int str)`

如果 `move` 着手防守 `str` 返回 `true`。即如果不防守 `str` 可被吃住。

- `int somewhere(int color, int last_move, ...)`

例子：`somewhere(WHITE, 2, apos, bpos, cpos)`。如果点列表中有一个满足

`board[pos]==color` 返回 `true`。这里 `num_moves` 是着手总数减一，棋块不允许是死棋。仅当 `stackp==0` 有效。

- `int visible_along_edge(int color, int apos, int bpos)`

沿边缘寻找第一个棋子。从 `apos` 开始向 `bpos` 的方向。如果第一个棋子为给定方返回 1。要求 `apos` 和 `bpos` 到边缘距离相等。

- `int test_symmetry_after_move(int move, int color, int strict)`

着手后棋盘是否对称（或反对称）？如果着手是 `PASS_MOVE`，检查当前棋盘。如果设置了 `strict` 要求每个棋子与对方棋子对称，否则只要求有棋子。

- `int play_break_through_n(int color, int num_moves, ...)`

函数 `play_break_through_n()` 从 `color` 方开始连续交替着手。按顺序着手完毕，最后三手的局面由 `break_through()` 分析，检查双方是否包围和/或联络。如果最后三个局面为空，则为失败。如果一个或更多的着手因某些原因为非法，其他顺序仍执行，并调用 `break_through()`，就像什么也没发生。像 `break_through()` 一样，如果尝试打入成功函数返回 1，如果只分断返回 2。

- `int play_attack_defend_n(int color, int do_attack, int num_moves, ...)`

- `int play_attack_defend2_n(int color, int do_attack, int num_moves, ...)`

函数 `play_attack_defend_n()` 从 `color` 方开始连续交替着手。按顺序着手完毕，最后着手根据 `do_attack` 值给出攻防目标。如果没有攻防点，则假定它已被提掉。如果一个或更多的着手因某些原因为非法，其他顺序仍执行，并测试攻防，就像什么也没发生。相应地 `play_attack_defend2_n()` 从 `color` 方开始连续交替着手。按顺序着手完毕，最后两个着手根据 `do_attack` 值给出两个攻防目标。如果没有攻防点，则假定它已被提掉。如果一个或更多的着手因某些原因为非法，其他顺序仍执行，并测试攻防，就像什

么也没发生。这些函数典型用法是在自动辅助函数中建立征子并判断效果。

```
● int play_connect_n(int color, int do_connect, int num_moves, ...)
```

从 color 方开始连续交替着手。按顺序着手完毕，最后两个着手根据 do_connect 值给出联络目标。如果没有联络点，则假定联络失败。如果一个或更多的着手因某些原因为非法，其他顺序仍执行，并测试联络，就像什么也没发生。最终联络由函数 string_connect 和 disconnect 确定（参见

```
0#####
```

```
#####      .gdbinit file      #####
```

```
#####
```

```
# this command displays the stack
```

```
define dump
```

```
set dump_stack()
```

```
end
```

```
# display the name of the move in ascii
```

```
define ascii
```

```
set gprintf("%o%m\n", $arg0, $arg1)
```

```
end
```

```
# display the all information about a dragon
```

```
define dragon
```

```
set ascii_report_dragon("$arg0")
```

```
end
```

```
define worm
```

```
set ascii_report_worm("$arg0")
```

```
end
```

```
# move to the next variation
```

```
define nv
```

```
tbreak trymove
```

```
continue
```

```
finish
```

```
next
```

```
end
```

```
# move forward to a particular variation
```

```
define jt
```

```

while (count_variations < $arg0)
nv
end
nv
dump
end

# restart, jump to a particular attack variation

define avar
delete
tbreak sgffile_decidestring
run
tbreak attack
continue
jt $arg0
end

# restart, jump to a particular defense variation

define dvar
delete
tbreak sgffile_decidestring
run
tbreak attack
continue
finish
next 3
jt $arg0
end

```

联络识别)。

- void set_depth_values(int level)

stackp >= depth 时识别征子，假定征子方一棋子一旦被叫吃，则被征棋串安全。同样地可用于其他层参数，如 backfill_depth 等。简明而言，stackp 大的时候使用简化的假定。不幸的是这样的方案会引进“水平线效应”，即如果将“水平线”——识别假定所基于的 stackp 值一旦超出，就可能错误地认定征子成功。为避免这个深度，有时必须增加深度参数。这个函数可用于各种识别深度参数。如果 mandated_depth_value 不是-1 就使用该值；否

则深度值设定为 level 的函数。参数 mandated_depth_value 可在命令行设置强制深度值；一般是-1。

- `void modify_depth_values(int n)`

修改各种战术识别深度参数。典型由于避免水平线效应。尝试一些着手时临时增加深度值，可避免识别相关着手时仅因为识别过早达到深度限制而将其视为有效。

- `void increase_depth_values(void)`
`modify_depth_values(1).`

- `void decrease_depth_values(void)`
`modify_depth_values(-1).`

- `void restore_depth_values()`

根据存储值设置深度。

- `void set_temporary_depth_values(int d, int b, int b2, int bc, int ss,
int br, int f, int k)`

确切设定深度值。目前未使用。

- `int confirm_safety(int move, int color, int *defense_point, char
safe_stones[BOARDMAX])`

检查 color 方 move 着手没有错误，不考虑大小。

- `float blunder_size(int move, int color, int *defense_point, char
safe_stones[BOARDMAX])`

检查错误。如果着手减少相邻同方棋串气数，有可能是坏棋，函数确认同方围受攻棋串是否因此受攻以及对方不可防守的棋串是否因此可防守。返回错误大小的估计，或 0.0 如果没有错误。数组 safe_stones[] 包含着手后确定安全的棋子，可以是 NULL。从 fill_liberty() 调用时，该函数可选返回防守点，可在后续着手中使得 move 着手安全。

- `int double_atari(int move, int color, float *value, char
safe_stones[BOARDMAX])`



如果 color 方着手 move 满足下形返回 true:



```
X*      (O=color)
OX
```



吃住两个“X”棋串之一。名字有些不符因为其中包括并非双叫吃的攻击，当然双叫吃是最重要的特例。如果 `safe_stones != NULL`，只尝试攻击标记 `safe` 的棋子。双叫吃攻击价值在 `value` 中返回（除非为 `NULL`）受攻棋子标记 `unsafe`。



```
● void unconditional_life(int unconditional_territory[BOARDMAX], int
    color)
```

对给定 `color` 方寻找即使对方连走也不能被吃住的棋串。结果棋串的首子写入 `worm` 数组并返回不能吃住的棋串数目。算法是由对方尝试收气，如有可提棋串则提掉，循环判别直至没有可提掉的。如果攻击失败，回溯着手，这样所有棋串都不能被提时，剩下的就是无条件活棋。然后识别出对方无条件死棋串和无条件实地，为此从前面操作得到的局面（`color` 方只剩下无条件活棋）继续以下步骤：

  对方在无条件活棋串收气除非非法。（着手次序是否决定着手合法无关紧要，次序任意）。

  递归扩展对方被叫吃的棋串，除非是自杀。

  对方在可取得的两空邻点的位置着手。（即大眼分小）。

  对方在有空邻点的位置着手（即两空减为一空。）剩下的被叫吃的对方棋串和无条件活棋串的气组成无条件实地。初始局面中处于无条件实地的对方棋串为无条件死。`Color` 有无条件活子时 `unconditional_territory[][]` 返回 1，有无条件实地时返回 2，否则返回 0。

```
● void who_wins(int color, FILE *outfile)
```

棋局评分确定胜方。

```
● void find_superstring(int str, int *num_stones, int *stones)
```

寻找扩展棋串的棋子，扩展超串（参见 11.8 超串）。

- `void find_superstring_liberties(int str, int *num_libs, int *libs, int liberty_cap)`

计算 `str` 超串, 忽略第 5 类 (参见 11.8 超串)。建立不是 `str` 气但是超串气的列表。如果 `liberty_cap` 非零, 则只生成超串中气数小于 `liberty_cap` 的子串的气。

- `void find_proper_superstring_liberties(int str, int *num_libs, int *libs, int liberty_cap)`

与 `find_superstring_liberties` 相同, 但忽略棋串 `str` 的气, 因为已经处理过。如果 `liberty_cap` 非零, 则只生成超串中气数最多为 `liberty_cap` 的子串的气。

- `void find_superstring_stones_and_liberties(int str, int *num_stones, int *stones, int *num_libs, int *libs, int liberty_cap)`

计算 `str` 超串, 忽略第 5 类 (参见 11.8 超串)。建立不是 `str` 气但是超串气的列表。如果 `liberty_cap` 非零, 则只生成超串中气数小于 `liberty_cap` 的子串的气。

- `void superstring_chainlinks(int str, int *num_adj, int adjs[MAXCHAIN], int liberty_cap)`

类似 `chainlinks`, 寻找 `str` 超串的边界链, 包括 `str` 自身的边界链。如果 `liberty_cap != 0`, 只报告 \leq `liberty_cap` 气数。

- `void proper_superstring_chainlinks(int str, int *num_adj, int adjs[MAXCHAIN], int liberty_cap)`

类似 `chainlinks`, 寻找 `str` 超串的边界链, 包括 `str` 自身的边界链。如果 `liberty_cap != 0`, 只报告 \leq `liberty_cap` 气数。

- `void start_timer(int n)`

开始计时器。GNU Go 内部有四个计时器可存取各种任务用时。

- `double time_report(int n, const char *occupation, int move, double mintime)`

报告用时并重启计时器。如果间隔小于 `mintime` 不报告。

18.2 打印工具

“`engine/printutils.c`” 中函数提供类似 `printf` 和同类的格式化打印。识别以下格式：

- `%c, %d, %f, %s, %x`

通常意义的格式输出，打印字符、整数、浮点数、字符串或 16 进制数。

- `%o`

“`Outdent`” 通常输出是 `2*stackp` 空格，跟踪时可一眼看出到层数。`%o` 在格式开始禁止缩进。

- `%H`

打印 Hash 值。

- `%C`

打印棋手方字符串。

- `%m, %2m (synonyms)`

取 2 个整数输出着手，使用二维棋盘表示（参见棋盘数组 15.2）。

- `%1m`

取 1 个整数输出着手，使用一维棋盘表示（参见棋盘数组 15.2）。

- `%r`

取 1 个整数输出着手，使用一维打印识别结果字符串。可能的输出：

- 0
- KO_B
- LOSS
- GAIN

- KO_A
- WIN
- SEKI
- ERROR

在“printutils.c” 中的静态声明函数：

- void gfprintf(FILE *outfile, const char *fmt, ...)

格式化输出到 “outfile”。

- int gprintf(const char *fmt, ...)

格式化输出到 stderr。总返回 1 允许短路逻辑表达式。

- int mprintf(const char *fmt, ...)

格式化输出到 stdout。

- DEBUG(level, fmt, args...)

如果 level & debug，格式化输出到 stderr。否则忽略。

- void abortgo(const char *file, int line, const char *msg, int pos)

错误位置打印调试输出并退出。

- const char * color_to_string(int color)

对局方转换为字符串。

- const char * location_to_string(int pos)

位置转换为字符串。

- void location_to_buffer(int pos, char *buf)

位置转换为字符串输出到缓冲。

- int string_to_location(int boardsize, char *str, int *m, int *n)

以标准 GNU Go 坐标系从字符串 str 中取 (m, n)。“m”是从顶数第 m 行

“n”是列。坐标为 0 到 boardsize-1，成功返回 1，否则返回 0。

- int is_hoshi_point(int m, int n) 如果是星位返回 True。
- void draw_letter_coordinates(FILE *outfile) 用坐标打印一行。
- void simple_showboard(FILE *outfile)

showboard(0) 的骨架。没有特殊选项，没有对局方，可以选择输出。

以下函数在“showbord.c”中，这里没有列出所有公共函数。

- void showboard(int xo)

显示棋盘。

```
xo=0:    black and white XO board for ascii game
xo=1:    colored dragon display
xo=2:    colored eye display
xo=3:    colored owl display
xo=4:    colored matcher status display
```

- const char * status_to_string(int status)

状态转换为字符串。

- const char * safety_to_string(int status)

安全值转换为字符串。

- const char * routine_to_string(int routine)

例程转换为字符串。

- const char * result_to_string(int result)

识别结果转换为字符串。

18.3 棋盘工具

本节所列函数来自“board.c”。“board.c”中其他函数在 15.4 棋盘函数。

- void store_board(struct board_state *state)

存储棋盘状态。

- void restore_board(struct board_state *state)

恢复存储的棋盘状态。

- void clear_board(void)

清除内部棋盘。

- void dump_stack(void)

在 gdb 下打印着手堆栈。

- `void add_stone(int pos, int color)`

在棋盘放置棋子并更新 `board_hash`，该操作破坏着手历史。

- `void remove_stone(int pos)`

在棋盘移除棋子并更新 `board_hash`，该操作破坏着手历史。

- `int is_pass(int pos)`


测试着手是否为放弃，是返回 1。


- `int is_legal(int pos, int color)`


确定 `color` 方在 `pos` 着手是否合法。

- `int is_suicide(int pos, int color)`

确定 `color` 方在 `pos` 着手是否是自杀。具体情形：

 没有相邻空点。

 没有只有一气的对方相邻棋串。

 没有大于一气的本方相邻棋串。

- `int is_illegal_ko_capture(int pos, int color)`

确定 `color` 在 `pos` 着手是否非法提劫。

- `int is_edge_vertex(int pos)`

确定点是否在边缘。

- `int edge_distance(int pos)`

到边缘的距离。

- `int is_corner_vertex(int pos)`

确定点是否在角部。

- `int get_komaster()`

- `int get_kom_pos()`

存取变量 `komaster` 和 `kom_pos` 的公共函数，在“`board.c`”静态声明。

下面是 `countlib()` 系。针对确定棋串气数问题。`countlib()` 针对基本问题，其他函数经常需要快速获取信息，故该系有一些不同的函数。

- `int countlib(int str)`

计算 pos 棋串气数，该位置必须有子。

- `int findlib(int str, int maxlib, int *libs)`

寻找 str 棋串气点，位置不能为空。最多 maxlib 个气点输出到 libs[]。返回所有气数。如果不管气数多少而需要所有气点，应将 MAXLIBS 作为 maxlib 值并为 libs[] 分配相应空间。

- `int fastlib(int pos, int color, int ignore_captures)`

计算如果 color 在 pos 着手棋子所能得到的气数。该函数意图是尽可能快而不必要完整。如果返回正值（成功），该值必须保证正确。如果 ignore_captures 非零不考虑提子。位置 pos 必须为空。如果有两个以上同方相邻棋串返回失败值 -1。提子以非常有限的方式操作，如果 ignore_capture 为 0，且需要提子，经常返回 -1。

- `int approxlib(int pos, int color, int maxlib, int *libs)`

寻找如果 color 在 pos 着手棋子所能得到的气数，忽略可能的提子。位置 pos 必须为空。如果 libs != NULL，最多 maxlib 个气点输出到 libs[]。到达 maxlib 后数气可能停止也可能不停止。返回找到的气数，如果 maxlib 小返回可能小于实际气数。如果不管气数多少需要所有的气点，应将 MAXLIBS 作为 maxlib 并为 libs[] 分配相应空间。

- `int accuratelib(int pos, int color, int maxlib, int *libs)`

寻找寻找如果 color 在 pos 着手棋子所能得到的气数。考虑所有提子。返回值考虑所有气数，除非 maxlib 允许提前停止。位置必须为空。如果 libs != NULL，最多 maxlib 个气点输出到 libs[]。到达 maxlib 后数气可能停止也可能不停止。返回找到的气数。该函数保证非由提子产生的气点在 libs[] 数组中前位。检查是否从给定位置开始所有气数都是提子结果可用 (board[libs[k]] != EMPTY)。如果不管气数多少需要所有的气数或气

点，应将 MAXLIBS 作为 maxlib 并为 libs[] 分配相应空间。

下面是一些通用工具函数。

- `int count_common_libs(int str1, int str2)`

寻找两个棋串公气。

- `int find_common_libs(int str1, int str2, int maxlib, int *libs)`

寻找两个棋串公气。最多 maxlib 个公气输出到 libs[]。返回公气数。如果不管气数多少需要所有的公气点，应将 MAXLIBS 作为 maxlib 并为 libs[] 分配相应空间。

- `int have_common_lib(int str1, int str2, int *lib)`

确定两棋串是否至少有一公气。如果有且 lib != NULL，在 *lib 中返回一个公气点。

- `int countstones(int str)`

报告棋串中棋子数。

- `int findstones(int str, int maxstones, int *stones)`

寻找 str 棋串的棋子。位置必须非空。最多 maxstones 个棋子位置输出到 stones[]。返回棋子总数。

- `int chainlinks(int str, int adj[MAXCHAIN])`

非常有用的函数。（在 adj 数组）返回包围棋串 str 的链。返回链数。

- `int chainlinks2(int str, int adj[MAXCHAIN], int lib)`

（在 adj 数组）返回包围棋串 str，且气数为 lib 的链。返回链数。

- `int chainlinks3(int str, int adj[MAXCHAIN], int lib)`

（在 adj 数组）返回包围棋串 str，且气数小于等于 lib 的链。返回链数。

- `int extended_chainlinks(int str, int adj[MAXCHAIN], int both_colors)`

（在 adj 数组）与 str 直接相邻或有一公气的对方棋串。如果 both_colors 参数为 true，同时返回本方有公气的棋串。

- `int find_origin(int str)`

寻找棋串首子，即 1D 棋盘坐标最小的棋子。为建立棋串正则索引。

- `int is_self_atari(int pos, int color)`

确定 `color` 在 `pos` 着手是否自叫吃，即只有一气。自杀情形也返回 `true`。

- `int liberty_of_string(int pos, int str)`

如果 `pos` 是 `str` 棋串的气点返回 `true`。

- `int second_order_liberty_of_string(int pos, int str)`

如果 `pos` 是 `str` 棋串第二顺序气点返回 `true`。

- `int neighbor_of_string(int pos, int str)`

如果 `pos` 与 `str` 棋串相邻返回 `true`。

- `int has_neighbor(int pos, int color)`

如果 `pos` 有 `color` 方邻串返回 `true`。

- `int same_string(int str1, int str2)`


如果 `str1` 和 `str2` 属于同一棋串返回 `true`。


- `int adjacent_strings(int str1, int str2)`

如果 `str1` 和 `str2` 棋串相邻返回 `true`。

- `int is_ko(int pos, int color, int *ko_pos)`

如果 `color` 方在 `pos` 着手是提劫（无论合法与否）返回 `true`。如果是，且 `ko_pos` 不是 `NULL` 指针，`*ko_pos` 返回提劫点位置。如果不是，`*ko_pos` 为 0。着手为提劫当且仅当：

-  邻点都是对方棋子。

-  提子数为一。

- `int is_ko_point(int pos)`

如果 `pos` 是棋子被提即是劫，或 `pos` 是与提劫点相邻的空点返回 `true`。

- `int does_capture_something(int pos, int color)`

如果 color 方在 pos 着手至少提掉一个棋串返回 1。

- `void mark_string(int str, char mx[BOARDMAX], char mark)`

对于 pos 处棋串所有棋子，设定 mx 为值 mark。如果棋串中一些棋子早先调用本函数已经标记，则只保证标记相连的为标记棋子。棋串中其他棋子可能标记也可能不标记。（目前实现中标记）。

- `int move_in_stack(int pos, int cutoff)`

如果在识别树深于 cutoff 级 pos 位置至少有一个着手返回 true。

- `int stones_on_board(int color)`

返回棋盘上 color 方棋子数。只在固定局面数子，而不用于 trymove() 或 tryko() 产生的局面。使用 stones_on_board(BLACK | WHITE) 获取棋盘上棋子总数。

18.4 “engine/influence.c” 中的工具

在此仅列出 influence.c 中部分公共函数。影响代码由函数 compute_influence 启动(参见 13.3 引擎中采用影响函数的地方)。如下：

- `void compute_influence(int color, const char safe_stones[BOARDMAX], const float strength[BOARDMAX], struct influence_data *q, int move, const char *trace_message)`

计算双方影响值。调用方必须：

- 设置 board[] 状态
- 用 INFLUENCE_SAFE_STONE 标记安全棋子，0 标记死子
- 用 INFLUENCE_SAVED_STONE 标记最新救出的棋子（以备 influence_data *q 计算本着手后续价值时再用）。

结果存储在 q。除非切换调试否则 move 没有作用。设定为 -1 没有任何调试输出(否则由命令行选项“-m”控制)。假定 color 先手。（影响障碍模式(A、D 类)和侵入模式(B 类)。

“influence.c”中其他函数是引擎有用的工具。最有用的列表如下：

- `void influence_mark_non_territory(int pos, int color)`

为“barriers.db”中“t”模式操作调用。标记 pos 为 color 不能成实地。

- `int whose_territory(const struct influence_data *q, int pos)`

返回在 pos 有实地的对局方。如果没有任一方，返回 EMPTY。

- `int whose_moyo(const struct influence_data *q, int pos)`

返回在 pos 有模样的对局方。如果没有任一方，返回 EMPTY。由影响定义模样非常特别。

- `int whose_area(const struct influence_data *q, int pos)`

返回在 pos 有优势影响（“区域”）的对局方。如果没有任一方，返回 EMPTY。由影响定义区域非常特别。

19 GTP 协议

19.1 GTP 协议

19.2 GTP 方式启动 GNUGO

19.3 协议应用

19.4 Metamachine

19.5 增加新 GTP 命令

19.6 GTP 命令参考

19.1 GTP 协议

GNU GO 3.0 引入了一个新接口——围棋文本协议，英文缩写：GTP，其目的是建立一个比 GMP (Go Modem Protocol) 协议的 `ascii` 接口更合适于双机通信、更简单、更有效、更灵活的接口。

该协议有两种版本。GNU GO 3.0 和 3.2 使用了版本 1，GNU GO 3.4 及以后的版本使用协议版本 2。规格定义在 <http://www.lysator.liu.se/~gunnar/gtp/>。GNU GO 3.4 参考实现了 GTP 版本 2，除了多数共同的命令也对这个协议进行了私有扩展。

GTP 有很多用途。对 GNU GO 来说，第一个用途是回归测试 (20 回归测试)，其次是利用 GTP 与 NNGS 围棋服务器连接，同自身或其它程序对弈进行自动测试。目前也有许多支持 GTP 的图形用户界面，用来桥接与 NNGS 类似的围棋服务器。

19.2 GTP 方式启动 GNUGO

以 GTP 方式启动 GNU GO, 只要用“--mode gtp”参数启动 GNUGO 就可以。

接着你不会得到提示或什么其它输出, 但是 GNU GO 已经在默默地等待 GTP 命令。

一个 GTP 会话如下例:

```
virihure 462% ./gnugo --mode gtp
1 boardsize 7
=1

2 clear_board
=2

3 play black D5
=3

4 genmove white
=4 C3

5 play black C3
?5 illegal move

6 play black E3
=6

7 showboard
=7
  A B C D E F G
7 . . . . . 7
6 . . . . . 6
5 . . + X + . . 5
4 . . . + . . . 4
3 . . O . X . . 3
2 . . . . . 2   WHITE (O) has captured 0 stones
1 . . . . . 1   BLACK (X) has captured 0 stones
  A B C D E F G

8 quit
=8
```

命令行输入, 用可选的数字标识起始, 后跟命令名和它的参数。

命令成功时返回等号(=)开始,接着是命令的数字标识(如果有的话)和结果。例子中多数结果都是空字符串,只有命令 4 的返回白着手 C3、命令 7 的返回当前棋盘局面图,其它所有结果是空串。返回串由两个连续的换行结束。

失败的命令由问号“?”代替等号,如命令 5 的返回。

该协议的详细规格在 <http://www.lysator.liu.se/~gunnar/gtp/>。可用的所有命令在 GNU GO 中以命令 `list_commands` 列出。文档参见。

19.3 协议应用

GTP 是非对称协议,包括控制端和引擎两方面。控制端发送命令,引擎响应之。GNU GO 实现了这个协议的引擎端。

随 GNU GO 源代码一起也发布几个实现了控制端的应用。其中最有趣的一些:

- `regression/regress.awk`

回归测试脚本,该脚本向引擎发送 GTP 命令设定和评估局面,然后分析引擎的回应。关于 GTP 基于回归测试的更多信息参见回归一章(20 回归测试)。

- `regression/regress.pl`

回归测试的 Perl 脚本,与 CGI 脚本“`regression/regress.plx`”一起生成回归测试的 HTML 文件。

- `regression/regress.pike`

回归测试的 Pike 脚本。比“`regress.awk`”更丰富、更有效。

- `regression/view.pike`

通过图形棋盘检查单个回归测试项的 Pike 脚本。提供了一个检查 GNU GO 内核的简便方法。

- `interface/gtp_examples/twogtp`

使两个引擎互相对弈的 Perl 脚本。这个脚本实际上为两个引擎设置了棋盘尺寸、让子数和贴目，然后在引擎之间来回传送着手。

- `interface/gtp_examples/twogtp-a`

可与 twogtp 互换的另一 Perl 脚本。

- `interface/gtp_examples/twogtp.py`

twogtp 的 Python 实现脚本。比其 Perl 实现功能更多。

- `interface/gtp_examples/twogtp.pike`

twogtp 的 pike 实现脚本。比 Python 实现功能更多。

- `interface/gtp_examples/2ptkgo.pl`

包括一个图形棋盘的 twogtp 变体。

更多 GTP 应用列表，包括桥接到围棋服务器和图形用户界面：
<http://www.lysator.liu.se/~gunnar/gtp/>。

19.4 Metamachine

GTP 的一个有趣的应用是让 GNU GO 作为权威，指导其它程序。另一个计算机程序可以请求 GNU GO 生成后续局面并进行评估。

Santa Cruz 的加州大学的 David Doshay 用基于 GNU GO 开发的 SlugGo 并行引擎做过一个有趣的实验。详细描述在
<http://lists.gnu.org/archive/html/gnugo-devel/2004-08/msg00060.html>。

" Metamachine " 实验是一个更为普通的尝试，采用 GTP 与被作为权威使用的 GNU GO 程序通信。原理如下：

- 使用 GTP `top_moves` 命令请求 GNU GO“权威”生成第一步着手。
- 尝试双方着手并对当前盘面评分。
- 选择评分较高局面作为引擎的着手。

该方案不会产生一个更强的引擎，但具有启发性，并且 SlugGo 实验展示了一个较为精致的方案，沿此路线可能会产生较强的引擎。

两种实现都随同 GNU GO 发布。都采用 fork 和 pipe 系统调用，因此需要一个类 Unix 环境。Metamachine 已在 GNU/Linux 之下测试。

注意：如果 Metamachine 正常终止，GNU GO 进程将被杀掉，但总会有出错的可能，所以运行完 Metamachine 后最好运行 `ps -A|grep gnugo` 或 `ps -aux|grep gnugo` 确认没有未终止的进程。(如果有，就杀掉。)

19.4.1 独立的 Metamachine

“`interface/gtp_examples/metamachine.c`”是独立实现的 Metamachine 程序。用 `cc -o metamachine metamachine.c` 编译运行。它通过 GTP 与一个 GNUGO 进程沟通，并把 GNUGO 作为权威。

采用以下原理：

```
stdin          pipe a
GTP client ----> Metamachine -----> GNU Go
               <-----
               <-----
stdout         pipe b
```

大多数由 client 发布的命令通过 Metamachine 逐个传送给 GNU GO。唯一例外是 `gg_genmove`，该命令被截获并转发。Client 并不知道这种情况，它只知道发出了一个 `gg_genmove` 命令并收到了回应。因此 Metamachine 更像一个普通的 GTP 引擎。

用法：无参数给出标准的 GTP 行为。`metamachine --debug` 将调试信息发送到 `stderr`。

19.4.2 GNU GO 作为 Metamachine

另外，也可以使用配置选项 “`--enable-metamachine`”编译 GNU GO。这编译了包含 Metamachine 代码的文件 `oracle.c`。除非运行 GNU GO 时使用了实时选项 “`--metamachine`”，否则这对引擎没有影响。因此需要同时使用编译选

项和实时选项获取 Metamachine。

这个方法比独立程序好，因为可以使用 GNU GO 的工具。例如这样可以用 CGoban 或 Ascii 方式运行 Metamachine。

增加命令行“-d0x1000000”可以进行跟踪。在调试 Metamachine 时，危险在于编程中的一个小失误就可能导致通信进程和控制器挂起，彼此都在等待对方的回应。如果这看来发生了，你可以把 gdb 放入运行过程，察看发生了什么，这很有用。

19.5 增加新 GTP 命令

在 GNU GO 中 GTP 的实现分发三个文件，“interface/gtp.h”、“interface/gtp.c”和“interface/play_gtp.c”。前两个实现了一个小的辅助函数库，供其它程序使用。为了推广的需要，以最低的限制条件发放许可（参见 [A.3 The Go Text Protocol License](#) 部分）。实际的 GTP 命令在“play_gtp.c”中实现，与引擎内核交互。

来看一个简单而相当典型的命令实现：gtp_countlib()（一个 GNUGO 的私有扩展命令）：

```
static int
gtp_countlib(char *s)
{
    int i, j;
    if (!gtp_decode_coord(s, &i, &j))
        return gtp_failure("invalid coordinate");

    if (BOARD(i, j) == EMPTY)
        return gtp_failure("vertex must not be empty");

    return gtp_success("%d", countlib(POS(i, j)));
}
```

命令参数通过字符串 s 传递，在这里参数是一个棋盘点，故此调

用“gtp.c”的 `gtp_decode_coord()` 读取。

一个正确格式化的回应应以“=”或“?”开始，接着是数字标识（如果传递了的话）和实际结果，最后是两个连续的换行符结束。正确地获取这种格式非常重要，因为对端的控制器依赖于它。当然结果本身不能包含两个连续的换行符，但可以用单个换行符分成多行。

假设要格式化的文本没用换行符结束，要生成正确格式化的回应，最容易的方法是使用函数 `gtp_failure()` 和 `gtp_success()` 之一。

有时输出会过于复杂不适合使用 `gtp_success`，如我们想打印棋盘点

`gtp_success()` 就不支持。这时只有回到构造，如 `gtp_genmove()`

```
static int
gtp_genmove(char *s)
{
    [...]
    gtp_start_response(GTP_SUCCESS);
    gtp_print_vertex(i, j);
    return gtp_finish_response();
}
```

这里 `gtp_start_response()` 输出写等号和数字标识，
`gtp_finish_response()` 增加最后的两个换行符。下个例子来自

```
gtp_list_commands()
static int
gtp_list_commands(char *s)
{
    int k;
    UNUSED(s);

    gtp_start_response(GTP_SUCCESS);

    for (k = 0; commands[k].name != NULL; k++)
        gtp_printf("%s\n", commands[k].name);


    gtp_printf("\n");
    return GTP_OK;
}
```

刚才说到，回应必须以两个换行符结束。这里我们只好自己来完成回应，

因为在循环中最后一个命令已经输出了一个换行符。

为了增加一个新 GTP 命令至 GNU GO，需要在“play_gtp.c”中插入以下代码片断：

 在第 68 行开始用 DECLARE 宏指令声明函数。

 在第 200 行开始加入 commands[] 数组。

 操作命令的代码实现。

“gtp.c”/“gtp.h”中有用的辅助函数：

- gtp_printf() 基本格式打印
- gtp_mprintf() 棋盘点和棋子格式编码打印
- gtp_success() 和 gtp_failure() 简单回应
- gtp_start_response() 和 gtp_end_response() 较复杂的应答
- gtp_print_vertex() 和 gtp_print_vertices() 打印一个和多个棋盘点
- gtp_decode_color() 从命令参数中读出棋子
- gtp_decode_coord() 从命令参数中读出棋盘点
- gtp_decode_move() 从命令参数中读出着手，即棋子加棋盘点

19.6 GTP 命令参考

本节列出 GNU GO 实现命令及其参考，每个命令入口包括以下的域：

- Function: 该命令的功能
- Arguments: 命令需要的其它信息，典型值包括 none、vertex、integer 等
- Fails: 导致命令失败的详情
- Returns: 在 = 和连续两个换行符间显示的内容，典型值包括 nothing、a move coordinate、some status string 等。
- Status: 该命令与标准 GTP 第 2 版的关系，如果没有关联则是 GNUGO 的私有扩展。

细节不再赘述，所列为所有命令（没有特别顺序）：

Argument:

N-None

V-Vertex

I-Integer

F-Float

C-Color "black", "white", or "empty"

NH- number of handicap stones

LVH-list of vertices with handicap stones

LVH-list of vertices

F-FileName

MN-Move Number

M-move (color + V)

Fails:

N-Never

IL-Illegal

IC-Incorrcect

IV- invalid

IM- illegal move

BNE-board not empty

BOL-Board outside limit

FF-missing filename or failure to open or parse file

EV-empty vertex

OV-occupied vertex

NV-Not Available

SE-stack empty

Returns:

N-Nothing

DSstatus ("alive", "critical", "dead", or "unknown")

US unconditional status ("undecided", "alive", "dead", "white_territory", "black_territory").

ES Status in the form of one of the strings "alive", "dead", "seki", "white_territory", "black_territory", or "dame"

着手列表 color move (one move per line)

white_influence (float)
black_influence (float)
white_strength (float)
black_strength (float)
white_attenuation (float)
black_attenuation (float)
white_permeability (float)
black_permeability (float)
territory_value (float)
influence_regions (int)
non_territory (int)

The encoding of influence_regions is as follows:

4 white stone
3 white territory
2 white moyo
1 white area
0 neutral
-1 black area
-2 black moyo
-3 black territory
-4 black stone

Worm data formatted like A19:

color	black
size	10
effective_size	17.83
origin	A19
liberties	8
liberties2	15
liberties3	10

```

liberties4      8
attack          PASS
attack_code     0
lunch           B19
defend          PASS
defend_code     0
cutstone        2
cutstone2       0
genus           0
inessential     0

B19:
color           white
.
.
.
inessential     0

C19:

```

Name	Function	Argument	Fails	Returns	Status
Quit	退出	N	N	N	G2S
Protocol_version	报告协议版本号.	N	N	协议版本号	G2S
Name	报告程序名称.	N	N	程序名称	G2S
version	报告程序版本号.	N	N	版本号	G2S
Boardsize	设置路数并清盘	I	BOL	N	G2S
equery_boardsiz	询问当前棋盘路数	N	N	路数	G2S
clear_board	清盘面.	N	N	N	G2S
Orientation	设置方向并清盘	I	IL	N	
query_orientation	查询当前方向	N	N	方向	
Komi	设置贴目	F	IC	N	G2S
get_komi	取贴目数	N	N	贴目	
Black	给定节点落黑子	V	IV/IM	N	OG1
Playwhite	给定节点落白子.	V	IV/IM	N	OG1
Play	给定节点给定落子	C/V	IV/IM	N	G2S
fixed_handicap	设置固定位置让子	NH	IV	LVH	G2S
place_free_handicap	设置自由位置让子	NH	IV	LVH	G2S
set_free_handicap	放置自由位置让子	LVH	BNE/IV	N	G2S
get_handicap	取让子数	N	N	NH	
Loadsgf	打开一个 SGF 文件	F+MN	FF	C	G2S

	至所给的手数或第一手	/V/N V		
Color	返回端点的子色	V	IV	C
list_stones	黑子或白子的端点列表	C	IV	LV
Countlib	计算端点所在串的气数	V	IV/EV	气数
Findlib	返回端点串的气点列表	V	IV/EV	节点列表
Accuratelib	检查落子气点列表	M	IV/OV	节点列表
accurate_approxlib	检查落子气点列表	M	IV/OV	节点列表
is_legal	着手是否合法	M	IV	1/0
all_legal	所有合法着法节点列表	C	IV	节点列表
Captures	返回提子数	C	IV	提子数
last_move	上一手的子色端点	N	NV	M
move_history	招法历史反序列表	N	N	着手列表
Trymove	尝试落子	M	IV/IM	N
Tryko	忽略劫争尝试落子	M	IV/IM	
Popgo	取消 trymove 或 tryko	N	SE	N
clear_cache	清高速缓存	N	N	N
Attack	试吃一串	V	IV/EV	攻击代码和评分
attack_either	试吃两串之一	2V	IV/EV	攻击代码
Defend	试防守一串	V	IV/EV	防守代码和评分
ladder_attack	试征吃一串	V	IV/EV	攻击代码
increase_depths	增加深度	N	N	N
decrease_depths	减少深度	N	N	N
owl_attack	试吃龙	V	IV/EV	攻击代码
owl_defend	试防守龙	V	IV/EV	防守代码
owl_threaten_attack	试两步吃龙	V	IV/EV	防守代码
owl_threaten_defense	试两步防守龙	V	IV/EV	防守代码
owl_does_attack	确定指定着法对指定的龙有攻击	V,V	IV/EV	攻击代码
owl_does_defend	确定指定着法对指定的龙有防守	V,V	IV/EV	防守代码
owl_connection_defends	确定指定着法连接指定的两条龙	V,V	IV/EV	防守代码
defend_both	试图防守两串	2V	IV/EV	防守代码
owl_substantial	确定是否吃住一串给活龙	V	IV/EV	1/0
analyze_semeai	对杀分析	D,D	IV/EV	对杀防守、攻击结果、着手
analyze_semeai_after_move	下过一手后的对杀分析	C,V,D,D	IV	对杀防守、攻击结果、着手
tactical_analyze_semeai	不使用 OWL 的对杀分析	D,D	IV/EV	棋块状态
Connect	试连接两串	V, V	IV/EV	联络结果
Disconnect	试断开两串	V, V	IV/EV	分断结果
break_in	试以串破空	V, Vs	IV/EV	结果和评分

Block_off	试对串守空	V, Vs	IV/EV	结果和评分	
Eval_eye	眼空间评价	V	IV	眼最大最小数	
dragon_status	龙状态	OV	IV/EV	DS、攻击、防守评分	
same_dragon	确定两子属于同一条龙	V, V	IV/EV	1/0	
unconditional_status	确定端点状态	V	IV	US	
Combination_attack	按给定子色寻找通过组合攻击吃住某些子的着法	C	IV	推荐着手	
combination_defend	如给定子色能通过组合攻击吃住对方，列表对方对此攻击的防守着法	C	IV	推荐着手	
aa_confirm_safety	运行 atari_atari_confirm_safety()	M,OI	IM	成功代码	
genmove_black	产生并下出认为最好的黑着法	N	N	着手或 "PASS"	OG1
genmove_white	产生并下出认为最好的白着法	N	N	着手或 "PASS"	OG1
Genmove	以给定子色产生并下出认为最好的着法	C	IV	着手或 "PASS"	G2S
reg_genmove	以给定子色产生并下出认为最好的着法	C	IV	着手或 "PASS"	G2S
gg_genmove	以给定子色产生并下出认为最好的着法	C,OI	IV	着手或 "PASS"	
restricted_genmove	只在给定的端点中选择，并以给定子色产生并下出认为最好的着法	C,Vs	IV	着手或 "PASS"	
kgs-genmove_cleanup	以给定子色产生并下出认为最好的着法，在对手死子未清除前不 PASS	C	IV	着手或 "PASS"	KGS
Level	设置棋力	Int	IC	N	
Undo	悔一手	N	IV	N	G2S
gg-undo	悔 N 手	OI	IV	N	
time_settings	设时间	I,I,I	IV	N	G2S
time_left	报告剩余时间	C,C,I,I	IV	N	G2S
Final_score	记谱完成的棋局	OI	N	SGF 评分	G2S
Final_status	报告下完棋局的端点最终状态	V,OI	IV	ES	
final_status_list	按指定最终状态报告下完棋局的端点列表	ES,OI	IV	节点列表	G2S
estimate_score	记录评比	N	N	评分上下限	
experimental_score	重视下一手谁走的估计评分	C	IV	评分	

reset_life_node_counter	重设活的节点计数	N	N	N	
get_life_node_counter	取活的节点数	N	N	活节点数	
reset_owl_node_counter	重设 OWL 节点计数	N	N	N	
get_owl_node_counter	取 OWL 节点数	N	N	OWL 节点数	
reset_reading_node_counter	重设计算节点数	N	N	N	
get_reading_node_counter	取计算节点数	N	N	识别节点数	
reset_trymove_counter	重 设 trymoves/trykos 计数	N	N	N	
get_trymove_counter	取 trymoves/trykos 计数	N	N	trymoves/ trykos 数	
reset_connection_node_counter	重设连接节点计数	N	N	N	
get_connection_node_counter	取连接节点计数	N	N	联络节点数	
Test_eyeshape	为矛盾的评估测试 眼形	Vs	IV	故障保护	
analyze_eyegraph	对眼图计算眼值和 要点	S	IV	眼值	
Cputime	以秒返回共用 CPU 时间	N	N	秒	
Showboard	向 stdout 写盘面	N	N	N	G2S
dump_stack	向 stderr 转储堆栈	N	N	N	
move_influence	落子后返回有关初 始的影响函数信息	M, Inf	N	影响数据	
move_probabilities	列出欲下的每着及 概率，如前面没有 genmove 命令，则 此命令结果无效	N	N	着手配对	
move_uncertainty	返回着法的不确定 成份	N	N	不确定位	
followup_influence	返回一着后产生的 影响信息	M, Inf	N	影响数据	
worm_data	返回串数据结构中 的信息	OV	N	棋串数据	
worm_stones	列表串的子	V	EV	棋子列表	
worm_cutstone	返回串数据结构中 的 cutstone 域	V	N	分断子	
dragon_data	返回龙数据结构中 的信息	OV	N	棋块数据	
dragon_stones	列表龙的子	V	IV	棋子列表	
Eye_data	返回眼数据结构中 的信息	C, V	N	眼数据	
half_eye_data	返回后手眼数据结 构中的信息	V	N	后手眼数据	
start_sgftrace	开始在内存中以 SGF 树存储计算期的招 法	N	N	N	
finish_sgftrace	完成存储着法的 SGF 树，并写入文件	F	N	N	
Printsgf	当前局面输出到 SGF	F	MF	N	
tune_move_ordering	调谐局部计算的着	Is	IV	N	

	法命令参数				
Echo	显示参数	S	N	N	
echo_err	显示参数至 stdout 及 stderr	S	N	N	
Help	列表所支持的命令	N	N	命令列表	G2S
Known_command	报告一条命令是否支持	CMD	N	"true" /"false"	G2S
report_uncertainty	从 owl_attack and owl_defend 调谐不 确定报告开或关	"on" / "off"	IV	N	
get_random_seed	取随机种子	N	N	随机数	
set_random_seed	设随机种子	I	IV	N	
is_surrounded	确定龙是否被围	V	IV/EV	0/1/2	
does_surround	确定一着是否能围住龙	V,V	IV/EV	1	
surround_map	报告给定端点的龙的包围图	V,V	IV/EV	包围值	
set_search_diamond	限制搜索，建立搜索菱形	Pos	IV	N	
reset_search_mask	为限定搜索清盘面标志	N	N	N	
Limit_search	设置限定搜索全局变量	Value	IV	N	
set_search_limit	为限定搜索标志一个端点	Position	IV	N	
draw_search_area	画搜索区，写至 stder	N	N	N	

20 回归测试

经典的回归测试，其目的是避免重复的程序错误。找到一个错误后，程序员在修正该错误的同时向测试包中加入一个测试项，该测试项在修正前失败而在修正后通过。在发布一个新版本前，执行测试包中所有测试项即可及时发现重复出现的旧错误，因该错误无法通过相应的测试项。

GNU Go 中的回归测试略有差别。一个典型的测试项包括给出一个局面并请求引擎下一着手。将该着手与一个或多个正确着手相比较，以判定该测试项通过与否。同时还存储该测试项预期通过还是失败，以及在此状态下的偏差，以表明该修正是否解决了问题，在解决问题的同时是否又带来了新问题。这样，回归测试既包括了有待修正的引擎错误局面，也包括了此修正可能影响到的正确局面。

20.1 GNU Go 中的回归测试

20.2 测试包

20.3 性能测试

20.4 运行 regress.pike

20.5 用 Emacs 查看测试

20.6 HTML 回归查看

20.1 GNU Go 中的回归测试

执行回归测试的文件包含在 `regression/` 目录。测试项是以 GTP 指令存储在后缀为 `.tst` 的文件中，并以 GTP 注释形式存储了相应的正确结果和预期的通过/失败状态。可以使用 shell 脚本 `test.sh`、`eval.sh` 或 `regress.sh` 运行测试包。也可以使用 Makefile 目标文件，如果运行 `make all_batches`

可以执行大多数的测试包。Pike 脚本 `regress.pike` 可以执行所有的测试项或测试项的子集。

回归测试所使用的棋谱存储在目录 `regression/games/` 及其子目录中。

20.2 测试包

所有测试项以 GTP 指令形式分组存储于各测试包，测试包的内容片段如下：

下：

```
# Connecting with ko at B14 looks best. Cutting at D17 might be
# considered. B17 (game move) is inferior.
loadsgf games/strategy25.sgf 61
90 gg_genmove black
#? [B14|D17]

# The game move at P13 is a suicidal blunder.
loadsgf games/strategy25.sgf 249
95 gg_genmove black
#? [!P13]

loadsgf games/strategy26.sgf 257
100 gg_genmove black
#? [M16]*
```

以索引符（#）起始的行或通常情况下索引符后面所有的内容都解释为 GTP 注释并为引擎所忽略。GTP 指令按显示的顺序执行，但只有带数字编号的行才用作测试。以#?起始的注释行是回归测试脚本的幻数，指示正确结果和预期的通过/失败状态。方括号中的字符串按正则表达式与前面的数字编号行 GTP 指令的结果匹配。正则表达式特别有用的一个特性是可以使用|表示变着。这样，上面的 `B14|D17` 就表示对于测试项 90，生成的结果是 B14 或 D17 都为通过。还有一个重要的特殊情况需要注意，如果正确结果字符串以惊叹号起始，则表示该结果在正则表达式中排除，即是匹配是否定的。这样，对于测试项 95，`!P13` 表示除了 P13 外都是正确结果。

对于测试项 100，#?行的方括号后跟了一个星号（*），表示该测试预期失

败，如果没有星号则表示预期成功。方括号还可以后跟 & 表示忽略其结果，这主要用作统计报告，测试包运行中经过了多少战术识别节点。

20.3 性能测试

`./test.sh blunder.tst` 执行 `blunder.tst` 中的测试项，并按数字

编号打印测试结果：

```
1 E5
2 F9
3 O18
4 B7
5 A4
6 E4
7 E3
8 A3
9 D9
10 J9
11 B3
12 C6
13 C6
```

这样产生的信息不是很充分，更有趣的是 `./eval.sh blunder.tst` 还将

结果与测试文件比较，并按如下形式打印：

```
1 failed: Correct "!E5", got "E5"
2 failed: Correct "C9|H9", got "F9"
3 PASSED
4 failed: Correct "B5|C5|C4|D4|E4|E3|F3", got "B7"
5 PASSED
6 failed: Correct "D4", got "E4"
7 PASSED
8 failed: Correct "B4", got "A3"
9 failed: Correct "G8|G9|H8", got "D9"
10 failed: Correct "G9|F9|C7", got "J9"
11 failed: Correct "D4|E4|E5|F4|C6", got "B3"
12 failed: Correct "D4", got "C6"
13 failed: Correct "D4|E4|E5|F4", got "C6"
```

测试结果为如下四种之一：

- passed: 预期通过

最理想的结果。

- PASSED: 意外通过

我们做修正时所希望的结果。是一个过去失败现在通过的旧测试项。

- failed: 预期失败

测试如我们预期那样失败，除非该测试项是用来验证修补。该测试项显示 GNU Go 引擎的弱点，如果你想搜寻发现一个有待改进的领域，这是一个很好的地方。

- FAILED: 意外失败

顾名思义这是在修正中破坏了他东西。有时 GNU Go 在带有一个或多个错误的情况下通过了一项测试，其中一个错误被修正后，其他错误显露出来从而使得测试失败。当测试出现意外失败时，有必要做仔细的检查已确定是否有一个修正破坏了他东西。

如果需要一个更简短的报告，可以使用 `./regress.sh .`

`blunder.tst`，只报告意外结果。上面的例子就压缩为：

```
3 unexpected PASS!
5 unexpected PASS!
7 unexpected PASS!
```

F 为方便起见，测试可以采用 `makefile` 目标文件方式。例如，`make blunder` 用 `eval.sh blunder.tst` 执行测试包 `blunder`。`make test` 用 `regress.sh` 脚本按顺序执行所有测试包。

20.4 运行 regress.pike

一个更有效的执行回归测试的方法是使用“`regress.pike`”脚本。这需要先安装 Pike (<http://pike.ida.liu.se/>)。

不带参数运行 `regress.pike`，可以执行 `make all_batches` 执行的所有测试包，区别是发现意外结果后立刻报告（而不是执行完整个文件后），同时为每个测试文件和整个测试统计所消耗的时间和节点。

执行一个测试包，如 `./regress.pike nicklas3.tst`
或 `./regress.pike nicklas3`，结果如下：

```
nicklas3                2.96    614772    3322    469
Total nodes: 614772 3322 469
Total time: 2.96 (3.22)
Total uncertainty: 0.00
```

数字表示测试包花费处理器时间 2.96 秒、实际时间 3.22 秒，识别节点开销，战术识别用了 614772 个，OWL 识别用了 3322 个，连接识别用了 469。最后一行与测试包产生着手法的变化相关，0 表示着手评估中没有随机性贡献。

可以执行多个测试包，如：`./regress.pike owl ld_owl owl1`。

可以执行单个测试项如 `./regress.pike strategy:6`，多个测试项如 `./regress.pike strategy:6,23,45`，测试项范围如 `./regress.pike strategy:13-15`，以及更为复杂的组合如 `./regress.pike strategy:6,13-15,23,45 nicklas3:602,1403`。

可以通过命令行选项选择引擎、引擎选项、打开详细输出，也可以用文件指定需要执行的测试项。运行 `./regress.pike -help` 可以查看完整的最新选项列表。

20.5 用 Emacs 查看测试

为快速地察看测试，可以使用 Emacs 的图形显示模式（参见 3.5 Emacs 的 GNU Go 模式）。进入 `M-x gnugo` 时可能需要回归缓存的游标，以使 GNUGO 在正确的目录打开。进入正确目录的一个好办法是打开想要研究的测试窗口，从 Emacs 用：命令直接将 GTP 命令由测试窗口剪切粘贴到微型缓存。尽管 Emacs 模式没有坐标网格，可用 `:showboard` 命令得到带坐标网格的 `ascii` 棋盘。

20.6 HTML 回归查看

可以使用两个 perl 脚本 `regression/regress.pl` 、
`regression/regress.plx` 生成非常有用的 HTML 回归视图。

驱动程序 (`regress.pl`) :

调用 GNU Go 执行回归测试

捕获跟踪输出、棋盘局面、通过/失败状态、sgf 输出和棋块 状态信息。

查看捕获到的输出 (`regress.plx`) 的接口:

并不调用 GNU Go

以直观的方式 (即 HTML) 显示捕获的输出。

20.6.1 配置 HTML 回归视图

本文档假设在 per Debian 的 Apache 1.3 版本配置 apache, 但对于其他版本的配置非常相似。

首先, 配置 Apache 在你需要的提供 html 视图的目录上运行 CGI 脚本, 为此, 将下列内容加入 `"/etc/apache/httpd.conf"` (或可用的用户指定配置文件):

```
<Directory /path/to/script/>
Options +ExecCGI
</Directory>
```

这使得 CGI 脚本可以在 `regress.plx` 使用的目录下运行。接着, 配置 Apache 指明 `".plx"` 是 CGI 脚本端接。你的 `"httpd.conf"` 文件要包含一节

`<IfModule mod_mime.c>`, 如果没有下述内容, 应加入在最后:

```
AddHandler cgi-script ....
```

再将 `".plx"` 加入扩展列表。

需要确认装载了运行 CGI 所必需的模块；`mod_cgi` 和 `mod_mime` 足够了。“`httpd.conf`”中应有相关的 `LoadModule` 行，如果必要为它们取消注释。

然后将 “`regress.plx`”复制到计划提供 html 视图服务的目录。

还需要安装 Perl 模块 GD，该模块可从 CPAN 或通过 Debian 上的 `apt-get install libgd-perl` 获取。

最后，运行 `regression/regress.pl` 以建立用于生成 html 视图的 XML 数据，将 `html/` 目录复制到 `regress.plx` 所在的同名目录中。

到这里，你就有了一个 html 回归视图的工作拷贝。