Documentation for the GNU Go Project
Edition 3.0.0
August, 2001

**By Daniel Bump, David Denholm, Jerome Dumonteil,
Gunnar Farnebäck, Thomas Traber, Tanguy Urvoy, Inge Wallin**

GNU GO 3.0

This is Edition 3.0.0 of *The GNU Go Project documentation*,
for the 3.0 version of the GNU GO program.

Published by the Free Software Foundation
675 Massachusetts Avenue
Cambridge, MA 02139-3309 USA
Phone: +1-617-876-3296

1 Introduction

This is GNU Go 3.0, a Go program. Development versions of GNU Go may be found at `http://www.gnu.org/software/gnugo/devel.html`. Contact us at `gnugo@gnu.org` if you are interested in helping.

1.1 About GNU Go and this Manual

The challenge of Computer Go is not to **beat** the computer, but to **program** the computer.

In Computer Chess, strong programs are capable of playing at the highest level, even challenging such a player as Garry Kasparov. No Go program even as strong as amateur shodan exists. The challenge is to write such a program.

To be sure, existing Go programs are strong enough to be interesting as opponents, and the hope exists that some day soon a truly strong program can be written.

GNU Go is getting stronger. For one thing, we've paid a lot of attention to life and death. GNU Go 3.0 can consistently give GNU Go 2.6 a four stone handicap. In a four stone game against GNU Go 2.6, GNU Go 3.0 very often kills a group.

Until now, Go programs have always been distributed as binaries only. The algorithms in these proprietary programs are secret. No-one but the programmer can examine them to admire or criticise. As a consequence, anyone who wished to work on a Go program usually had to start from scratch. This may be one reason that Go programs have not reached a higher level of play.

Unlike most Go programs, GNU Go is Free Software. Its algorithms and source code are open and documented. They are free for any one to inspect or enhance. We hope this freedom will give GNU Go's descendents a certain competetive advantage.

Here is GNU Go's Manual. There are doubtless inaccuracies. The ultimate documentation is in the commented source code itself.

The first three chapters of this manual are for the general user. Chapter 3 is the User's Guide. The rest of the book is for programmers, or persons curious about how GNU Go works. Chapter 4 is a general overview of the engine. Chapter 5 introduces various tools for looking into the GNU Go engine and finding out why it makes a certain move, and Chapters 6–7 form a general programmer's reference to the GNU Go API. The remaining chapters are more detailed explorations of different aspects of GNU Go's internals.

1.2 Copyrights

Copyright 1999, 2000, 2001 by the Free Software Foundation except for the files '`gmp.c`' and '`gmp.h`', which are copyrighted by Bill Shubert (`wms@igoweb.org`).

All files are under the GNU General Public License (see Section A.1 [GPL], page 173), except '`gmp.c`', '`gmp.h`', '`gtp.c`', '`gtp.h`', the files '`interface/html/*`' and '`win/makefile.win`'.

The two files '`gmp.c`' and '`gmp.h`' were placed in the public domain by William Shubert, their author, and are free for unrestricted use.

The files 'gtp.c' and 'gtp.h' are copyright the Free Software Foundation. In the interests of promoting the Go Text Protocol these two files are licensed under a less restrictive license than the GPL and are free for unrestricted use (see Section A.3 [GTP License], page 185).

The files 'interface/html/*' are not part of GNU Go but are a separate program and are included in the distribution for the convenience of anyone looking for a CGI interface to GNU Go. They were placed in the public domain by their author, Douglas Ridgway, and are free for unrestricted use. The file 'win/makefile.win' is also in the public domain and is free for unrestricted use.

1.3 Authors

GNU Go maintainers are Daniel Bump and Gunnar Farnebäck. GNU Go authors (in chronological order of contribution) are Man Li, Daniel Bump, David Denholm, Gunnar Farnebäck, Nils Lohner, Jerome Dumonteil, Tommy Thorn, Nicklas Ekstrand, Inge Wallin, Thomas Traber, Douglas Ridgway, Teun Burgers, Tanguy Urvoy, Thien-Thi Nguyen, Heikki Levanto, Mark Vytlacil, Adriaan van Kessel, Wolfgang Manner, Jens Yllman and Don Dailey.

1.4 Thanks

We would like to thank Arthur Britto, Tim Hunt, Piotr Lakomy, Paul Leonard, Jean-Louis Martineau, Andreas Roever and Pierce Wetter for helpful correspondence. Thanks to everyone who stepped on a bug (and sent us a report)!

Thanks to Gary Boos, Peter Gucwa, Martijn van der Kooij, Michael Margolis, Trevor Morris, Mans Ullerstam, Don Wagner and Yin Zheng for help with Visual C++.

And thanks to Alan Crossman, Stephan Somogyi, Pierce Wetter and Mathias Wagner for help with Macintosh.

Special thanks to Ebba Berggren for creating our logo, based on a design by Tanguy Urvoy and comments by Alan Crossman. The old GNU Go logo was adapted from Jamal Hannah's typing GNU: http://www.gnu.org/graphics/atypinggnu.html. Both logos can be found in 'doc/newlogo.*' and 'doc/oldlogo.*'.

We would like to thank Stuart Cracraft, Richard Stallman and Man Lung Li for their interest in making this program a part of GNU, William Shubert for writing CGoban and gmp.c, Rene Grothmann for Jago and Erik van Riper and his collaborators for NNGS.

1.5 The GNU Go Task List

You can help make GNU Go the best Go program.

This is a task-list for anyone who is interested in helping with GNU Go. If you want to work on such a project you should correspond with us until we reach a common vision of how the feature will work!

A note about copyright. The Free Software Foundation has the copyright to GNU Go. For this reason, before any code can be accepted as a part of the official release of GNU Go, the Free Software Foundation will want you to sign a copyright assignment.

Of course you could work on a forked version without signing such a disclaimer. You can also distribute such a forked version of the program so long as you also distribute the source code to your modifications under the GPL (see Section A.1 [GPL], page 173). But if you want your changes to the program to be incorporated into the version we distribute we need you to assign the copyright.

Please contact the GNU Go maintainers, Daniel Bump (`bump@math.stanford.edu`) and Gunnar Farnebäck (`gf@isy.liu.se`), to get more information and the papers to sign.

Below is a list of things YOU could work on. We are already working on some of these tasks, but don't let that stop you. Please contact us or the person assigned to task for further discussion.

1. Report and fix bugs.

   Bugs are an important cause of weakness in any Go program! If you can, send us bug FIXES as well as bug reports. If you see some bad behavior, figure out what causes it, and what to do about fixing it. And send us a patch! If you find an interesting bug and cannot tell us how to fix it, we would be happy to have you tell us about it anyway. Send us the sgf file (if possible) and attach other relevant information, such as the GNU Go version number. In cases of assertion failures and segmentation faults we probably want to know what operating system and compiler you were using, in order to determine if the problem is platform dependent.

2. Extend the regression test suites.

   See the texinfo manual in the doc directory for a description of how to do this. In particular it would be useful with test suites for common life and death problems. Currently second line groups, L groups and the tripod shape are reasonably well covered, but there is for example almost nothing on comb formations, carpenter's square, and so on. Other areas where test suites would be most welcome are fuseki, tesuji, and endgame.

3. Tune the pattern databases.

   This is a sort of art. It is not necessary to do any programming to do this since most of the patterns do not require helpers. We would like it if a few more Dan level players would learn this skill.

4. Extend and tune the Joseki database.

5. Rewrite the semeai module

   The semeai module is vastly in need of improvement. In fact, semeai can probably only be analyzed by reading to discover what backfilling is needed before we can make atari.

6. Write a connection analysis module.

   The connection analysis is today completely static and has a hard time identifying mutually dependent connections or moves that simultaneously threatens two or more connections. This could be improved by writing a connection reader, which like the owl code uses pattern matching to find a small amount of key moves to try.

7. Speed up the tactical reading.

   GNU Go is reasonably accurate when it comes to tactical reading, but not always very fast. The main problem is that too many ineffective moves are tested, leading to strange variations that shouldn't need consideration. To improve this the move generation heuristics in the reading code needs to be refined. Some improvements should also be possible to obtain by tuning the move ordering.

8. Automatically search for errors.

   In some positions GNU Go may report a group as alive or connected with a living group. But after the opponent has placed one stone GNU Go may change the status to dead, without going through a critical status. It would be nice if these positions could be automatically identified and logged for later analysis.

2 Installation

You can get the most recent version of GNU Go ftp.gnu.org or a mirror (see `http://www.gnu.org/order/ftp.html` for a list). You can read about newer versions and get other information at `http://www.gnu.org/software/gnugo/`.

2.1 GNU/Linux and Unix

Untar the sources, change to the directory gnugo-3.0.0. Now do:

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

The most important configure options, cache size, default level and dfa will be explained in detail in the next section (see ⟨undefined⟩ [Configure Options], page ⟨undefined⟩). Probably you do not need to set these unless you are dissatisfied with GNU Go's performance for any reason.

As an example,

```
./configure --enable-cache-size=32 --enable-level=8
```

creates a 48 Mb cache in RAM and sets the level to 8. Both these defaults can be overridden at run time.

If you do not specify any options, the default level is 10 (highest supported), and the default cache size is 16, and the DFA is not enabled.

If you have a slow machine or find GNU Go too slow you may want to decrease the default level. At level 8 the engine is playing about 1.6 times faster than at level 10.

Increasing the cache size will improve performance up to a point — once the cache is so large that it cannot be kept in RAM, GNU Go will start swapping (characterized by frequent hard drive accesses) and performance will degrade.

You have now made a binary called '`interface/gnugo`'. Now (running as root) type

```
make install
```

to install gnugo in '`/usr/local/bin`'.

There are different methods of using GNU Go. You may run it from the command line by just typing:

```
gnugo
```

but it is nicer to run it using CGoban 1 (under X-Windows) or Jago (on any platform with a Java runtime environment).

You can get the most recent version of CGoban 1 from Bill Shubert's web site: `http://www.igoweb.org/~wms/comp/cgoban/index.html` The CGoban version number MUST be 1.9.1 at least or it won't work. CGoban 2 will not work.

See Section 3.2 [CGoban], page 10, for instructions on how to run GNU Go from Cgoban, or See Section 3.5 [Jago], page 12, for Jago.

There are three options which you should consider configuring, particularly if you are dissatisfied with GNU Go's performance.

2.1.1 Ram Cache

By default, GNU Go makes a cache of 16 Megabytes in RAM for its internal use. The cache is used to store intermediate results during its analysis of the position.

Increasing the cache size will often give a modest speed improvement. If your system has lots of RAM, consider increasing the cache size. But if the cache is too large, swapping will occur, causing hard drive accesses and degrading performance. If your hard drive seems to be running excessively your cache may be too large. On GNU/Linux systems, you may detect swapping using the program 'top'. Use the 'f' command to toggle SWAP display.

You may override the size of the default cache at compile time by running one of:

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

to set the cache size to `n` megabytes. For example

```
./configure --enable-cache-size=48
```

creates a cache of size 48 megabytes. If you omit this, your default cache size will be 16 MB. You must recompile and reinstall GNU Go after reconfiguring it by running `make` and `make install`.

You may override the compile-time defaults by running gnugo with the option '`--cache-size n`', where `n` is the size in megabytes of the cache you want, and '`--level`' where n is the level desired. We will discuss setting these parameters next in detail.

2.1.2 Default Level

GNU Go can play at different levels. Up to level 10 is supported. At level 10 GNU Go is much more accurate but takes an average of about 1.6 times longer to play than at level 8.

The level can be set at run time using the '`--level`' option. If you don't set this, the default level will be used. You can set the default level with the configure option '`--enable-level=n`'. For example

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

sets the default level to 9. If you omit this parameter, the compiler sets the default level to 10. We recommend using level 10 unless you find it too slow. If you decide you want to change the default you may rerun configure and recompile the program.

2.1.3 DFA Configure Option

If you `./configure --enable-dfa` you get the experimental DFA (Discrete Finite-State Automata) pattern matcher. This will result in a larger but somewhat faster engine. The option is considered experimental because it is new and harder to debug but sufficiently tested that it is probably safe.

2.2 Compiling GNU Go on Microsoft platforms

GNU Go is being developed on Unix variants. GNU Go is easy to build and install on those platforms. GNU Go 3.0 has support for building on MS-DOS, Windows 3.x, Windows NT/2000 and Windows 95/98.

There are two approaches to building GNU Go on Microsoft platforms.

1. The first approach is to install a Unix-like environment based on ports of GCC to Microsoft platforms. This approach is fully supported by the GNU Go developers and works well. Several high quality free Unix-environments for Microsoft platforms are available.

   One benefit of this approach is that it is easier to participate in Gnu Go's development. These unix environments come for instance with the 'diff' and 'patch' programs necessary to generate and apply patches.

   Another benefit of the unix environments is that development versions (which may be stronger than the latest stable version) can be built too. The supporting files for VC are not always actively worked on and consequently are often out of sync for development versions, so that VC will not build cleanly.
2. The second approach is to use compilers such as Visual C developed specially for the Microsoft platform. GNU Go 2.6 and later support Visual C. Presently we support Visual C through the project files which are supplied with the distribution.

The rest of this section gives more details on the various ways to compile GNU go for Microsoft platforms.

2.2.1 Windows 95/98, MS-DOS and Windows 3.x using DJGPP

On these platforms DJGPP can be used. GNU Go installation has been tested in a DOS-Box with long filenames on Windows 95/98. GNU Go compiles out-of-the box with the DJGPP port of GCC using the standard Unix build and install procedure.

Some URLs for DJGPP:

DJGPP home page: `http://www.delorie.com/djgpp/`

DJGPP ftp archive on simtel:

`ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/`

`ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/`

Once you have a working DJGPP environment and you have downloaded the gnugo source available as gnugo-3.0.0.tar.gz you can build the executable as follows:

```
tar zxvf gnugo-3.0.0.tar.gz
cd gnugo-3.0.0
./configure
make
```

Optionally you can download glib for DJGPP to get a working version of snprintf.

2.2.2 Windows NT, Windows 95/98 using Cygwin

On these platforms the Cygwin environment can be installed. Recent versions of Cygwin install very easily with the setup program available from the cygwin homepage. `<http://sourceware.cygnus.com/cygwin/`. GNU Go compiles out-of-the box using the standard Unix build procedure on the Cygwin environment. After installation of cygwin and fetching '`gnugo-3.0.0.tar.gz`' you can type:

```
tar zxvf gnugo-3.0.0.tar.gz
cd gnugo-3.0.0
```

```
./configure
make
```

The generated executable is not a stand-alone executable: it needs cygwin1.dll that comes with the Cygwin environment. cygwin1.dll contains the emulation layer for Unix.

Cygwin Home page: `http://sourceware.cygnus.com/cygwin/`

Optionally you can use glib to get a working version of snprintf. Glib builds out of the box on cygwin.

2.2.3 Windows NT, Windows 95/98 using MinGW32

The Cygwin environment also comes with MinGW32. This generates an executable that relies only on Microsoft DLLs. This executable is thus completely comparable to a Visual C executable and easier to distribute than the Cygwin executable. To build on cygwin an executable suitable for the win32 platform type the following at your cygwin prompt:

```
tar zxvf gnugo-3.0.0.tar.gz
cd gnugo-3.0.0
env CC='gcc -mno-cygwin' ./configure
make
```

2.2.4 Windows NT, Windows 95/98 using Visual C and project files

We assume that you do not want to change any configure options. If you do, you should edit the file '`config.vc`'. Note that when `configure` is run, this file is overwritten with the contents of '`config.vcin`', so you may also want to edit '`config.vcin`', though the instructions below do not have you running `configure`.

1. Open the VC++ 6 workspace file gnugo.dsw
2. Set the gnugo project as the active project (right-click on it, and select "Set as Active Project". Select 'Build' from the main menu, then select 'Build gnugo.exe', this will make all of the runtime subprojects.

Notes:

- a) The build can also be done from the command line:
  ```
  msdev gnugo.dsw /make "gnugo - Win32 Release"
  ```
- b) The default configuration is 'Debug', build the optimized version by selecting 'Build' from the main menu , then select 'Set active Configuration' and click on 'gnugo - Win32 Release'. See the Visual Studio help for more on project configurations.
- c) A custom build step in the first dependent subproject (utils) copys config.vc to config.h in the root directory. If you want to modify config.h, copy any changes to config.vc. In particular if you want to change the default level or default cache size, whose significance is discussed in See Section 2.1 [GNU/Linux and Unix], page 5, you must edit this file.
- d) This project was built and tested using VC version 6.0. It has not been tested, and will most likely not work with earlier versions of VC.
- e) If for any reason some or all of the automatically built files in the patterns directory do not build you can run mkpat on the command line to make these files. For reference here are the recommended mkpat options:

```
 FILE              MKPAT OPTIONS       INPUT FILES

conn.c            mkpat -c conn          conn.db
patterns.c        mkpat -b pat           patterns.db, patterns2.db
apatterns.c       mkpat -X attpat        attack.db
dpatterns.c       mkpat defpat           defense.db
influence.c       mkpat -c influencepat  influence.db
endgame.c         mkpat -b endpat        endgame.db
owl_attackpat.c   mkpat -b owl_attackpat  owl_attackpats.db
owl_vital_apat.c mkpat -b owl_vital_apat  owl_vital_apats.db
owl_defendpat.c   mkpat -b owl_defendpat  owl_defendpats.db
fuseki9.c         mkpat -b -f fuseki9    fuseki9.db
fuseki19.c        mkpat -b -f fuseki19   fuseki19.db
josekidb.c        mkpat -b joseki        hoshi.db, komoku.db,
                                         sansan.db, takamoku.db
                                         mokuhazushi.db
```

2.2.5 Running GNU Go on Windows NT and Windows 95/98

GNU Go does not come with its own graphical user interface. The Java client jago can be used.

To run Jago you need a Java Runtime Environment (JRE). This can be obtained from `http://www.javasoft.com/`. This is the runtime part of the Java Development Kit (JDK) and consists of the Java virtual machine, Java platform core classes, and supporting files. The Java virtual machine that comes with I.E. 5.0 works also.

Jago: `http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/jago/Go.html`

1. Invoke GNU Go with `gnugo --quiet --mode gmp`
2. Run `gnugo --help` from a cygwin or DOS window for a list of options
3. optionally specify `--level <level>` to make the game faster

Jago works well with both the Cygwin and MinGW32 executables. The DJGPP executable also works, but has some problems in the interaction with jago after the game has been finished and scored.

2.3 Macintosh

If you have Mac OS X you can build GNU Go using Apple's compiler, which is derived from GCC. We recommend adding the flag -no-cpp-precom to CFLAGS.

3 Using GNU Go

3.1 Getting Documentation

You can obtain a printed copy of the manual by running `make gnugo.ps` in the ‘`doc/`’directory, then printing the resulting postscript file. The manual contains a great deal of information about the algorithms of GNU Go.

On platforms supporting info documentation, you can usually install the manual by executing ‘make install’ (running as root) from the ‘`doc/`’ directory. The info documentation can be read conveniently from within Emacs by executing the command `Control-h i`.

Documentation in ‘`doc/`’ consists of a man page ‘`gnugo.6`’, the info files ‘`gnugo.info`’, ‘`gnugo.info-1`’, ... and the Texinfo files from which the info files are built. The Texinfo documentation contains this User’s Guide and extensive information about the algorithms of GNU Go, for developers.

If you want a typeset copy of the Texinfo documentation, you can `make gnugo.dvi` or `make gnugo.ps` in the ‘`doc/`’ directory.

You can make an HTML version with the command `makeinfo --html gnugo.texi`. Better HTML documentation may be obtained using `texi2html -split_chapter gnugo.texi`. You can obtain the `texi2html` utility (version 1.61 or later) from `http://www.mathematik.uni-kl.de/~obachman/Texi2html/`. (See also `http://texinfo.org/texi2html/`.)

User documentation can be obtained by running `gnugo --help` or `man gnugo` from any terminal, or from the Texinfo documentation.

Documentation for developers is in the Texinfo documentation, and in comments throughout the source. Contact us at `gnugo@gnu.org` if you are interested in helping to develop this program.

3.2 Running GNU Go via CGoban

This is an extremely nice way to run GNU Go. CGoban provides a beautiful graphic user interface under X-Windows.

Start CGoban. When the CGoban Control panel comes up, select “Go Modem”. You will get the Go Modem Protocol Setup. Choose one (or both) of the players to be “Program,” and fill out the box with the path to gnugo. After clicking OK, you get the Game Setup window. Choose “Rules Set” to be Japanese (otherwise handicaps won’t work). Set the board size and handicap if you want.

If you want to play with a komi, you should bear in mind that the GMP does not have any provision for communicating the komi. Because of this misfeature, unless you set the komi at the command line GNU Go will have to guess it. It assumes the komi is 5.5 for even games, 0.5 for handicap games. If this is not what you want, you can specify the komi at the command line with the ‘`--komi`’ option, in the Go Modem Protocol Setup window. You have to set the komi again in the Game Setup window, which comes up next.

Click OK and you are ready to go.

In the Go Modem Protocol Setup window, when you specify the path to GNU Go, you can give it command line options, such as '`--quiet`' to suppress most messages. Since the Go Modem Protocol preempts standard I/O other messages are sent to stderr, even if they are not error messages. These will appear in the terminal from which you started CGoban.

3.3 Ascii Interface

Even if you do not have CGoban installed you can play with GNU Go using its default Ascii interface. Simply type `gnugo` at the command line, and GNU Go will draw a board. Typing `help` will give a list of options. At the end of the game, pass twice, and GNU Go will prompt you through the counting. You and GNU Go must agree on the dead groups—you can toggle the status of groups to be removed, and when you are done, GNU Go will report the score.

You can save the game at any point using the `save` *filename* command. You can reload the game from the resulting SGF file with the command `gnugo -l` *filename* `--mode ascii`. Reloading games is not supported when playing with CGoban. However you can use CGoban to save a file, then reload it in ascii mode.

3.4 GNU Go mode in Emacs

You can run GNU Go from Emacs. This has the advantage that you place the stones using the cursor arrow keys. This may require Emacs 20.4 or later—it has been tested with Emacs 20.4 but does not work with Emacs 19 or Emacs 20.2.

Load '`interface/gnugo.el`', either by `M-x load-file`, or by copying the file into your '`site-lisp`' directory and adding a line

```
(autoload 'gnugo "gnugo" "GNU Go" t)
```

in your '`.emacs`' file.

Now you may start GNU Go by `M-x gnugo`. You will be prompted for command line options see Section 3.9 [Invoking GNU Go], page 12. Using these, you may set the handicap, board size, color and komi.

You can enter commands from the GNU Go ASCII interface after typing '`:`'. For example, to take a move back, type '`:back`', or to list all commands, type '`:help`'.

Here are the default keybindings:

- '`Return`' or '`Space`'

  Select point as the next move. An error is signalled for invalid locations. Illegal locations, on the other hand, show up in the GNUGO Console buffer.
- '`q`' or '`Q`'

  Quit. Both Board and Console buffers are deleted.
- '`R`'

  Resign.
- '`C-l`'

  Refresh. Includes restoring default window configuration.
- '`M-_`'

  Bury both Board and Console buffers (when the boss is near).

- ‘`p`’

  Pass; i.e., select no location for your move.
- ‘`:`’

  Extended command. After typing the ‘`:`’ you can type a command for GNU Go. The possible commands are as in See Section 3.3 [Ascii], page 11.

3.5 Running GNU Go via Jago

Jago, like CGoban is a client capable of providing GNU Go with a graphical user interface. Unlike CGoban, it does not require X-Windows, so it is an attractive alternative under Windows. You will need a Java runtime environment. Obtain Jago at

`http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/jago/Go.html` and follow the links there for the Java runtime environment.

3.6 The Go Modem Protocol and Go Text Protocol

The Go Modem Protocol (GMP) was developed by Bruce Wilcox with input from David Fotland, Anders Kierulf and others, according to the history in `http://www.britgo.org/tech/gmp.html`.

Any Go program *should* support this protocol since it is a standard. Since CGoban supports this protocol, the user interface for any Go program can be done entirely through CGoban. The programmer can concentrate on the real issues without worrying about drawing stones, resizing the board and other distracting issues.

GNU Go 3.0 introduces a new protocol, the Go Text Protocol (see Chapter 20 [GTP], page 164) which we hope can serve the functions currently used by the GMP.

3.7 Computer Go Tournaments

Computer Tournaments currently use the Go Modem Protocol. The current method followed in such tournaments is to connect the serial ports of the two computers by a “null modem” cable. If you are running GNU/Linux it is convenient to use CGoban. If your program is black, set it up in the Go Modem Protocol Setup window as usual. For White, select “Device” and set the device to ‘`/dev/cua0`’ if your serial port is COM1 and ‘`/dev/cua1`’ if the port is COM2.

3.8 Smart Go Format

The Smart Go Format (SGF), is the standard format for storing Go games. GNU Go supports both reading and writing SGF files. The SGF specification (FF[4]) is at: `http://www.red-bean.com/sgf/`

3.9 Invoking GNU Go: Command line options

3.9.1 Some basic options

'`--help`', '`-h`'
: Print a help message describing the options. This will also tell you the defaults of various parameters, most importantly the level and cache size. The default values of these parameters can be set before compiling by `configure`. If you forget the defaults you can find out using '`--help`'.

'`--boardsize` *size*'
: Set the board size

'`--komi` *num*'
: Set the komi

'`--level` *level*'
: GNU Go can play with different strengths and speeds. Level 10 is the default. Decreasing the level will make GNU Go faster but less accurate in its reading.

'`--quiet`', '`--silent`'
: Don't print copyright and other messages. Messages specifically requested by other command line options, such as '`--trace`', are not supressed.

'`-l`', '`--infile` *filename*'
: Load the named SGF file

'`-L`', '`--until` *move*'
: Stop loading just before the indicated move is played. *move* can be either the move number or location.

'`-o`', '`--outfile` *filename*'
: Write sgf output to file

'`--mode` *mode*'
: Force the playing mode ('ascii', 'test,' 'gmp' or 'gtp'). The default is ASCII, but if no terminal is detected GMP (Go Modem Protocol) will be assumed. In practice this is usually what you want, so you may never need this option.

3.9.2 Other general options:

'`-M`', '`--cache-size` *megs*'
: Memory in megabytes used for hashing. The default size is 16 unless you configure gnugo with the command `configure --enable-cache-size=`*size* before compiling to make *size* the default (see Chapter 2 [Installation], page 5). GNU Go stores results of its reading calculations in a Hash table (see Section 14.2 [Hashing], page 125). If the Hash table is filled, it is emptied and the reading continues, but some reading may have to be repeated that was done earlier, so a larger cache size will make GNU Go run faster, provided the cache is not so large that swapping occurs. Swapping may be detected on GNU/Linux machines using the program `top`. However if you have ample memory or if performance seems to be a problem may want to increase the size of the Hash cache using this option.

'`--chinese-rules`'

Use Chinese rules. This means that the Chinese or Area Counting is followed. It may affect the score of the game by one point in even games, more if there is a handicap (since in Chinese Counting the handicap stones count for Black).

'`--japanese-rules`'

Use Japanese Rules. This is the default unless you specify '`--enable-chinese-rules`' as a configure option.

'`--copyright`': Display the copyright notice

'`--version`' or '`-v`': Print the version number

'`--printsgf` *filename*': Create an SGF file containing a diagram of the board. Useful with '`-L`' to create diagrams from games.

3.9.3 Other options affecting strength and speed

The single parameter '`--level`' is a convenient way of choosing whether to play stronger or faster. This single parameter controls a host of other parameters which may optionally be set individually at the command line. The default values of these parameters may be found by running `gnugo --help`. Unless you are working on the program you probably don't need these options. Instead, just adjust the single variable '`--level`'.

These options are of use to developers tuning the program for performance and accuracy.

- '`-D`', '`--depth` *depth*'

  Deep reading cutoff. When reading beyond this depth (default 14) GNU Go assumes that any string which can obtain 3 liberties is alive. Thus GNU Go can read ladders to an arbitrary depth, but will miss other types of capturing moves.

- '`--branch-depth`'

  This sets the `branch_depth`, typically a little below the `depth`. Between `branch_depth` and `depth`, attacks on strings with 3 liberties are considered but branching is inhibited, so fewer variations are considered.

- '`-B`', '`--backfill-depth` *depth*'

  Deep reading cutoff. Beyond this depth (default 9) GNU Go will no longer try backfilling moves in its reading.

- '`--backfill2-depth` *depth*'

  Another depth controlling how deeply GNU Go looks for backfilling moves. The moves tried below `backfill2_depth` are generally more obscure and time intensive than those controlled by `backfill_depth`, so this parameter has a lower default.

- '`-F`', '`--fourlib-depth` *depth*'

  Deep reading cutoff. When reading beyond this depth (default 5) GNU Go assumes that any string which can obtain 4 liberties is alive.

- '`-K`', '`--ko-depth` *depth*'

  Deep reading cutoff. Beyond this depth (default 8) GNU Go no longer tries very hard to analyze kos.

- '`--branch-depth` *depth*'

  Deep reading cutoff. Below this depth (default 8), GNU Go still tries to attack strings with only 3 liberties, but only tries one move at each node.

- '`--aa_depth` *depth*'

  The reading function `atari_atari` looks for combinations beginning with a series of ataris, and culminating with some string having an unexpected change in status (e.g. alive to dead or critical). This command line optio sets the parameter `aa_depth` which determines how deeply this function looks for combinations.

- '`--superstring-depth`'

  A superstring (see Section 14.5 [Superstrings], page 135) is an amalgamation of tightly strings. Sometimes the best way to attack or defend a string is by attacking or defending an element of the superstring. Such tactics are tried below `superstring_depth` and this command line option allows this parameter to be set.

The preceeding options are documented with the reading code (see Section 14.1 [Reading Basics], page 122).

- '`owl-branch`' Below this depth Owl only considers one move. Default 8.
- '`owl-reading`' Below this depth Owl assumes the dragon has escaped. Default 20.
- '`owl-node-limit`' If the number of variations exceeds this limit, Owl assumes the dragon can make life. Default 10000. We caution the user that increasing `owl_node_limit` does not necessarily increase the strength of the program.
- '`--level` *amount*'

  The higher the level, the deeper GNU Go reads. Level 10 is the default. If GNU Go plays too slowly on your machine, you may want to decrease it.

3.9.4 Ascii Mode Options

'`--color` *color*'

Choose your color ('black' or 'white')

'`--handicap` *number*'

Choose the number of handicap stones (0–9)

3.9.5 Development options:

'`--replay` *color*'

Replay all moves in a game for either or both colors. If used with the '`-o`' option the game record is annotated with move values. This option requires '`-l` *filename*'. The color can be:

white: replay white moves only

black: replay black moves only

both: replay all moves

When the move found by genmove differs from the move in the sgf file the values of both moves are reported thus:

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

This option is useful if one wants to confirm that a change such as an speedup or other optimization has not affected the behavior of the engine. Note that when the several moves have the same top value (or nearly equal) the move generated is not deterministic (though it can be made deterministic by starting with the same random seed). Thus a few deviations from the move in the sgf file are to be expected. Only if the two reported values differ should we conclude that the engine plays differently from the engine which generated the sgf file. See Chapter 21 [Regression], page 170.

'`-a`', '`--allpats`'

Test all patterns, even those smaller in value than the largest move found so far. This should never affect GNU Go's final move, and it will make it run slower. However this can be very useful when "tuning" GNU Go. It causes both the traces and the output file ('`-o`') to be more informative.

'`-T`', '`--printboard`': colored display of dragons.

Use rxvt, xterm or Linux Console. (see Section 5.8 [Colored Display], page 33)

'`-E`': colored display of eye spaces

Use rxvt, xterm or Linux Console. (see Section 5.8 [Colored Display], page 33)

'`-d`', '`--debug` *level*'

Produce debugging output. The debug level is given in hexadecimal, using the bits defined in the following table from '`engine/gnugo.h`'.

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

'`-H`', '`--hash` *level*'

hash (see '`engine/gnugo.h`' for bits).

'`-w`', '`--worms`'

Print more information about worm data.

'`-m`', '`--moyo` *level*'

moyo debugging, show moyo board. The *level* is fully documented elsewhere (see Section 5.8 [Colored Display], page 33).

'`-b`', '`--benchmark` *number*'
: benchmarking mode - can be used with '`-l`'.

'`-s`', '`--stack`'
: stack trace (for debugging purposes).

'`-S`', '`--statistics`'
: Print statistics (for debugging purposes).

'`-t`', '`--trace`'
: Print debugging information. Use twice for more detail.

'`-r`', '`--seed` *seed*'
: Set random number seed. This can be used to guarantee that GNU Go will make the same decisions on multiple runs through the same game. If `seed` is zero, GNU Go will play a different game each time.

'`--decide-string` *location*'
: Invoke the tactical reading code (see Chapter 14 [Tactical Reading], page 122 to decide whether the string at *location* can be captured, and if so, whether it can be defended. If used with '`-o`', this will produce a variation tree in SGF.

'`--decide-dragon` *location*'
: Invoke the owl code (see Section 15.1 [The Owl Code], page 141) to decide whether the dragon at *location* can be captured, and whether it can be defended. If used with '`-o`', this will produce a variation tree in SGF.

'`--score` *method*'
: Requires '`-l`'. *method* can be "end", "last", "aftermath" or a move. "end" and "aftermath" are appropriate when the game is complete, or nearly so, and both try to supply an accurate final score. The other options may be used to get an estimate during the middle of the game. Any of these options may be combined with '`--chinese-rule`' if you want to use Chinese (Area) counting.

    last
    : load the sgf file up to the last move, then estimate territory using the Bouzy 5/21 algorithm (see Chapter 17 [Moyo], page 152).

    end
    : finish the game by selfplaying from the end of the file until two passes, then estimate territory using the Bouzy 5/21 algorithm (see Chapter 17 [Moyo], page 152).

    aftermath
    : finish the game by selfplaying from the end of the file until two passes, then estimate territory using the most accurate scoring algorithm available. Slower than '`--score last`', and while these algorithms usually agree, if they differ, '`--score aftermath`' is most likely to be correct.

    move, e.g. '`--score J17`'
    : load file until move is reached and estimate territorial balance using the Bouzy 5/21 algorithm. The '`--score end`'

and '`--score aftermath`' options are only useful at or near the end of the game, so if you want an estimate of the score in the middle, use this method.

'`--printsgf` *output file*'

load SGF file, output final position (requires '`-l`') as another SGF file. Illegal moves are indicated with the private `IL` property. This property is not used in the FF4 SGF specification, so we are free to preempt it. This feature is used in the CGI interface in '`interface/html/gg.cgi`'.

4 GNU Go engine overview

This chapter is an overview of the GNU Go internals. Further documentation of how any one module or routine works may be found in later chapters or comments in the source files.

4.1 Definitions

A *worm* is a maximal set of vertices on the board which are connected along the horizontal and vertical lines, and are of the same color, which can be `BLACK`, `WHITE` or `EMPTY`. The term `EMPTY` applied to a worm means that the worm consists of empty (unoccupied) vertices. It does **not** mean that that the worm is the empty set. A *string* is a nonempty worm. An empty worm is called a *cavity*. If a subset of vertices is contained in a worm, there is a unique worm containing it; this is its *worm closure*. (see Section 10.1 [Worms], page 67.)

A *dragon* is a union of strings of the same color which will be treated as a unit. If two strings are in the same dragon, it is the computer's working hypothesis that they will live or die together and are effectively connected. (see Section 10.7 [Dragons], page 76.)

A *superstring* is a less commonly used unit which is the union of several strings but generally smaller than a dragon. The superstring code is in '`engine/utils.c`'. The definition of a superstring is slightly different if the code is called from '`owl.c`' or from '`reading.c`'.

4.2 Move Generation Basics

The engine of GNU Go takes a positions and a color to move and generates the (supposedly) optimal move. This is done by the function `genmove()` in '`engine/genmove.c`'.

The move generation is done in three passes:

1. information gathering
2. different modules propose moves
3. The values of the moves are weighted together and the best move is selected

4.2.1 Information gathering

The information gathering is done by a function `examine_position()`, which will be discussed in greater detail in the next section. Such information could be life and death of the groups, information about moyos, connection of groups and so on. Information gathering is performed by `examine_position`, which in turn calls:

- `make_worms()`

  Collect information about all connected sets of stones (strings) and cavities. This information is stored in the `worm[][]` array. (see Section 10.1 [Worms], page 67)

- `compute_initial_influence()`

  Decides which areas of the board are influenced by which player. This function is run a second time later at the end of `make_dragons()`, since

GNU Go's opinion about the safety of groups may change, and it is important to have the influence function as accurate as possible. see Chapter 16 [Influence], page 145

- `make_dragons()`

  Collect information about connected strings, which are called dragons. Important information here is number of eyes, life status, and connectedness between string. (see Section 10.7 [Dragons], page 76.)

A more detailed

4.2.2 Move generation in GNU Go 3.0

Once we have found out all about the position it is time to generate the best move. Moves are proposed by a number of different modules called *move generators*. The move generators themselves do not set the values of the moves, but enumerate justifications for them, called *move reasons*. The valuation of the moves comes last, after all moves and their reasons have been generated.

The move generators in version 3.0 are:

- `fuseki()`

  Generate a move in the early fuseki.

- `semeai()`

  Find out if two dead groups of opposite colors are next to each other and, if so, try to kill the other group. This module will eventually be rewritten along the lines of the owl code.

- `shapes()`

  Find patterns from '`patterns/patterns.db`' in the current position. Each pattern is matched in each of the 8 possible orientations obtainable by rotation and reflection. If the pattern matches, a so called "constraint" may be tested which makes use of reading to determine if the pattern should be used in the current situation. Such constraints can make demands on number of liberties of strings, life and death status, and reading out ladders, etc. The patterns may call helper functions, which may be hand coded (in '`patterns/helpers.c`') or autogenerated.

  The patterns can be of a number of different classes with different goals. There are e.g. patterns which try to attack or defend groups, patterns which try to connect or cut groups, and patterns which simply try to make good shape. In addition to the large pattern database called by `shapes()`, pattern matching is used by other modules for different tasks throughout the program. See Chapter 12 [Patterns], page 92, for a complete documentation of patterns.

- `atari_atari()`

  See if there are any combination threats and either propose them or defend against them.

- `owl_reasons()`

  The Owl Code (see Section 15.1 [The Owl Code], page 141) which has been run during `examine_position`), before `owl_reasons()` executes, has decided whether different groups can be attacked. The module `review_owl_reasons` reviews the statuses of every dragon and assigns move reasons for attack and defense. Unlike the other move generation modules, this one is called from `examine_position()`.

- `endgame_shapes()`

  If no move is found with a value greater than 6.0, this module matches a set of extra patterns which are designed for the endgame. The endgame patterns can be found in '`patterns/endgame.db`'.
- `revise_semeai()`

  If no move is found, this module changes the status of opponent groups involved in a semeai from `DEAD` to `UNKNOWN`. After this, genmove runs `shapes` and `endgame_shapes` again to see if a new move turns up.
- `fill_liberty()`

  Fill a common liberty. This is only used at the end of the game. If necessary a backfilling or backcapturing move is generated.

4.2.3 Selecting the Move

After the move generation modules have run, the best ten moves are selected by the function `review_move_reasons`. This function also does some analysis to try to turn up other moves which may have been missed. The modules `revise_semeai()` and `fill_liberty` are only run if no good move has been discovered by the other modules.

4.3 Examining the Position

In this section we summarize the sequence of events when `examine_position()` is run from `genmove()`. This is for reference only. Don't try to memorize it.

```
purge persistent reading cache (see Section 14.2.5 [Persistent Cache], page 131)
make_worms() (see Section 10.1 [Worms], page 67):
  build_worms() finds and identifies the worms
  compute effective size of each worm
  unconditional_life()
  find_worm_attacks_and_defenses():
    for each attackable worm:
      set worm.attack
      add_attack_move()
    find_attack_patterns() to find a few more attacks
    for each defensible worm
      set worm.defend
      add_defense_move
      if point of attack is not adjacent to worm see if it defends
    find_defense_patterns() to find a few more defenses
    for each attackable worm try each liberty
      if it attacks add_attack_move
      if it defends add_defense_move
  find kos.
  for each worm
```

```
    find higher order liberties
  find cutting points (worm.cutstone)
  for each worm compute the genus (see Section 10.1 [Worms], page 67)
  small_semeai()
  try to improve values of worm.attack and worm.defend
  try to repair situations where adjacent worms can be
    both attacked and defended
  find worm lunches
  find worm threats
compute_initial_influence() (see Chapter 16 [Influence], page 145)
  compute_influence()
    find_influence_patterns()
  at each intersection accumulate_influence()
  segment_influence()
make_dragons() (see Section 10.7 [Dragons], page 76)
  initialize dragon data
  find the inessential worms
  make_domains()
    initialize eye data
    compute_primary_domains()
    fill out arrays black_eye and white_eye
      describing eyeshapes
    find_cuts()
    for every eyespace
      originate_eye()
    count_neighbors()
  find_connections()
  amalgamate dragons sharing an eyespace
  initialize_supplementary_dragon_data()
  find adjacent worms which can be captured (dragon lunches)
  find topological half eyes and false eyes
  modify_eye_spaces()
  for each eye space
    compute_eyes()
    store the results in black_eye, white_eye arrays
  compute the genus of each dragon
  for each dragon
    compute_escape()
  resegment_initial_influence()
  for each dragon
    influence_get_moyo_size()
  for each dragon
     compute_dragon_status()
  find_neighbor_dragons()
  purge_persistent_owl_cache()
  for each dragon which seems surrounded
     try owl_attack() and owl_defend()
     if appropriate find owl threats
  for each dragon
```

```
    set dragon.matcher_status
  for each dragon
    set dragon2.safety
  semeai()
  revise opinion of which worms are inessential
  count non-dead dragons of each color
owl_reasons() (see Section 15.1 [The Owl Code], page 141)
compute_initial_influence() again (see Chapter 16 [Influence], page 145)
```

4.4 Sequence of Events

In this section we summarize the sequence of events during the move generation and selection phases of `genmove()`, which take place after the information gathering phase has been completed.

```
fuseki()
shapes()
review_move_reasons()
  find_more_attack_and_defense_moves()
  remove_opponent_attack_and_defense_moves()
  do_remove_false_attack_and_defense_moves()
  examine_move_safety()
  induce_secondary_move_reasons()
  value_moves()
  find the ten best moves
if the value of the best move is < 6.0
  endgame_shapes()
if no move found yet
  revise_semeai()
  shapes()
  endgame_shapes()
if still no move found
  fill_liberty()
if still no move found
    pass
```

4.5 Roadmap

The GNU Go engine is contained in two directories, '`engine/`' and '`patterns/`'. Code related to the user interface, reading and writing of smart go format files and testing are found in the directories '`interface/`', '`sgf/`', and '`regression/`'. Code borrowed from other GNU programs is contained in '`utils/`'. Documentation is in '`doc/`'.

In this document we will describe some of the individual files comprising the engine code in '`engine/`' and '`patterns/`'. In '`interface/`' we mention two files:

'`gmp.c`'
: This is the Go Modem Protocol interface (courtesy of William Shubert and others). This takes care of all the details of exchanging setup and moves with Cgoban, or any other driving program recognizing the Go Modem Protocol.

'main.c'

This contains `main()`. The 'gnugo' target is thus built in the 'interface/' directory.

4.5.1 Files in 'engine/'

In 'engine/' there are the following files:

- 'aftermath.c'

  Contains algorithms which may be called at the end of the game to generate moves that will generate moves to settle the position, if necessary playing out a position to determine exactly the status of every group on the board, which GNU Go can get wrong, particularly if there is a seki. This module is the basis for the most accurate scoring algorithm available in GNU Go.
- 'board.c'

  This file contains code for the maintenance of the board. For example it contains the important function `trymove()` which tries a move on the board, and `popgo()` which removes it by popping the move stack. At the same time vital information such as the number of liberties for each string and their location is updated incrementally.
- 'clock.c'

  Clock code, including code allowing GNU Go to automatically adjust its level in order to avoid losing on time in tournaments.
- 'dragon.c'

  This contains `make_dragons()`. This function is executed before the move-generating modules `shapes()` `semeai()` and the other move generators but after `make_worms`. It tries to connect worms into dragons and collect important information about them, such as how many liberties each has, whether (in GNU Go's opinion) the dragon can be captured, if it lives, etc.
- 'fuseki.c'

  Generates fuseki (opening) moves from a database.
- 'filllib.c'

  Code to force filling of dame (backfilling if necessary) at the end of the game.
- 'genmove.c'

  This file contains `genmove()` and its supporting routines, particularly `examine_position()`.
- 'globals.c'

  This contains the principal global variables used by GNU Go.
- 'gnugo.h'

  This file contains declarations forming the public interface to the engine.
- 'hash.c' and 'cache.c'

  Hashing code implementing Zobrist hashing. (see Section 14.2 [Hashing], page 125) The code in 'hash.c' provides a way to hash board positions into compact descriptions which can be efficiently compared. The code in 'cache.c' implements a kind of database for storing reading results, so they can be quickly retrieved. The caching code uses the board hashes as keys to the database. They are split since these functionalities are sufficiently demarked that either file could be reimplemented without affecting the other one. Note also that `matchpat()` uses the hashing code without also using the caching code.

- 'hash.h' and 'cache.h'

  Header files for `hash.c` and `cache.c`.

- 'influence.c' and 'influence.h'.

  This code determines which regions of the board are under the influence of either player. (see Chapter 16 [Influence], page 145)

- 'liberty.h'

  Header file for the engine. The name "liberty" connotes freedom (see Appendix A [Copying], page 173).

- 'life.c'

  The code in this file contains an alternative approach to life and death based on reading instead of the static approach in 'optics.c'. This code is experimental. It is reasonably accurate but too slow. It is activated when gnugo is invoked with the '--life' option.

- 'matchpat.c'

  This file contains the pattern matcher `matchpat()`, which looks for patterns at a particular board location. The actual patterns are in the 'patterns/' directory. The function `matchpat()` is called by every module which does pattern matching, notably `shapes`.

- 'move_reasons.c'

  This file contains the code which assigns values to every move after all the move reasons are gen

- 'optics.c'

  This file contains the code to recognize eye shapes, documented in See Chapter 11 [Eyes], page 82.

- 'owl.c'

  This file does life and death reading. The paradigm is that moves are played by both players trying to expand and shrink the eyespace until a static configuration is reached where it can be analyzed by the code in 'optics.c' or 'life.c'.

- 'printutils.c'

  Print utilities

- 'reading.c'

  This file contains code to determine whether any given string can be attacked or defended. See Chapter 14 [Tactical Reading], page 122, for details.

- 'score.c'

  Implements the Bouzy algorithms (see Chapter 17 [Moyo], page 152) and contains code for scoring the game.

- ‘`semeai.c`’

  This file contains `semeai()`, the module which tries to win capturing races. This module does not work particularly well and will eventually be replaced.
- ‘`shapes.c`’

  This file contains `shapes()`, the module called by `genmove()` which tries to find moves which match a pattern (see Chapter 12 [Patterns], page 92).
- ‘`showbord.c`’

  This file contains `showboard()`, which draws an ASCII representation of the board, depicting dragons (stones with same letter) and status (color). This was the primary interface in GNU Go 1.2, but is now a debugging aid.
- ‘`worm.c`’

  This file contains `make_worms()`, code which is run at the beginning of each move cycle, before the code in ‘`dragon.c`’, to determine the attributes of every string. These attributes are things like liberties, wether the string can be captured (and how), etc
- ‘`utils.c`’

  An assortment of utilities, described in greater detail below.

4.5.2 Files in ‘`patterns/`’

The directory ‘`patterns/`’ contains files related to pattern matching. Currently there are several types of patterns. A partial list:

- move generation patterns in ‘`patterns.db`’ and ‘`patterns2.db`’
- move generation patterns in files ‘`hoshi.db`’ etc. which are automatically build from the files ‘`hoshi.sgf`’ etc. These comprise our small Joseki library.
- patterns in ‘`owl_attackpats.db`’, ‘`owl_defendpats.db`’ and ‘`owl_vital_apats.db`’. These generate moves for the owl code (see Section 15.1 [The Owl Code], page 141).
- Connection patterns in ‘`conn.db`’ (see Section 12.9 [Connections Database], page 104)
- Influence patterns in ‘`influence.db`’ and ‘`barriers.db`’ (see Chapter 16 [Influence], page 145)
- eye patterns in ‘`eyes.db`’ (see Chapter 11 [Eyes], page 82).

The following list contains, in addition to distributed source files some intermediate automatically generated files such as patterns.c. These are C source files produced by "compiling" various pattern databases, or in some cases (such as ‘`hoshi.db`’) themselves automatically generated pattern databases produced by "compiling" joseki files in Smart Go Format.

- ‘`conn.db`’:

  Database of connection patterns.
- ‘`conn.c`’:

  Automatically generated file, containing connection patterns in form of struct arrays, compiled by `mkpat` from ‘`conn.db`’.

- ‘eyes.c’:

  Automatically generated file, containing eyeshape patterns in form of struct arrays, compiled by mkpat from ‘eyes.db’.
- ‘eyes.h’:

  Header file for ‘eyes.c’.
- ‘eyes.db’:

  Database of eyeshape patterns. See Chapter 11 [Eyes], page 82, for details.
- ‘helpers.c’:

  These are helper functions to assist in evaluating moves by matchpat.
- ‘hoshi.sgf’:

  Smart Go Format file containing 4-4 point openings
- ‘hoshi.db’:

  Automatically generated database of 4-4 point opening patterns, make by compiling ‘hoshi.sgf’
- ‘joseki.c’:

  Joseki compiler, which takes a joseki file in Smart Go Format, and produces a pattern database.
- ‘komoku.sgf’:

  Smart Go Format file containing 3-4 point openings
- ‘komoku.db’:

  Automatically generated database of 3-4 point opening patterns, make by compiling ‘komoku.sgf’
- ‘mkeyes.c’:

  Pattern compiler for the eyeshape databases. This program takes ‘eyes.db’ as input and produces ‘eyes.c’ as output.
- ‘mkpat.c’:

  Pattern compiler for the move generation and connection databases. Takes the file patterns.db together with the autogenerated Joseki pattern files hoshi.db, komoku.db, sansan.db, ‘mokuhadzushi.db’, ‘takamoku.db’ and produces ‘patterns.c’, or takes ‘conn.db’ and produces ‘conn.c’.
- ‘mokuhazushi.sgf’:

  Smart Go Format file containing 5-3 point openings
- ‘mokuhazushi.db’:

  Pattern database compiled from mokuhadzushi.sgf
- ‘sansan.sgf’:

  Smart Go Format file containing 3-3 point openings
- ‘sansan.db’:

  Pattern database compiled from ‘sansan.sgf’
- ‘takamoku.sgf’:

  Smart Go Format file containing 5-4 point openings
- ‘takamoku.db’:

  Pattern database compiled from takamoku.sgf.

- ‘`patterns.c`’:

  Pattern data, compiled from patterns.db by mkpat.
- ‘`patterns.h`’:

  Header file relating to the pattern databases.
- ‘`patterns.db`’ and ‘`patterns2.db`’:

  These contain pattern databases in human readable form.

4.6 Coding styles and conventions

4.6.1 Coding Conventions

Please follow the coding conventions at: `http://www.gnu.org/prep/standards_toc.html`

Please preface every function with a brief description of its usage.

Please help to keep this Texinfo documentation up-to-date.

4.6.2 Tracing

A function `gprintf()` is provided. It is a cut-down `printf`, supporting only `%c`, `%d`, `%s`, and without field widths, etc. It does, however, add some useful facilities:

- `%m`:

  Takes two parameters, and displays a formatted board co-ordinate.
- indentation:

  Trace messages are automatically indented to reflect the current stack depth, so it is clear during read-ahead when it puts a move down or takes one back.
- "outdent":

  As a workaround, `%o` at the beginning of the format string suppresses the indentation.

A variant `mprintf` sends output to stderr. Normally `gprintf()` is wrapped in one of the following:

`TRACE(fmt, ...)`:

  Print the message if the 'verbose' variable `>` 0. (verbose is set by `-t` on the command line)

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

  While `TRACE` is intended to afford an overview of what GNU Go is considering, `DEBUG` allows occasional in depth study of a module, usually needed when something goes wrong. `flags` is one of the `DEBUG_*` symbols in ‘`engine/gnugo.h`’. The `DEBUG` macro tests to see if that bit is set in the `debug` variable, and prints the message if it is. The debug variable is set using the `-d` command-line option.

The variable `verbose` controls the tracing. It can equal 0 (no trace), 1, 2, 3 or 4 for increasing levels of tracing. You can set the trace level at the command line by ‘`-t`’ for `verbose=1`, ‘`-t -t`’ for `verbose=2`, etc. But in practice if you want more verbose tracing than level 1 it is better to use gdb to reach the point where you want the tracing; you will often find that the variable `verbose` has been temporarily set to zero and you can use the gdb command `set var verbose=1` to turn the tracing back on.

4.6.3 Assertions

Related to tracing are assertions. Developers are strongly encouraged to pepper their code with assertions to ensure that data structures are as they expect. For example, the helper functions make assertions about the contents of the board in the vicinity of the move they are evaluating.

`ASSERT()` is a wrapper around the standard C `assert()` function. In addition to the test, it takes an extra pair of parameters which are the co-ordinates of a "relevant" board position. If an assertion fails, the board position is included in the trace output, and `showboard()` and `popgo()` are called to unwind and display the stack.

4.6.4 FIXME

We have adopted the convention of putting the word FIXME in comments to denote known bugs, etc.

4.7 Navigating the Source

If you are using Emacs, you may find it fast and convenient to use Emacs' built-in facility for navigating the source. Switch to the root directory '`gnugo-3.0.x/`' and execute the command:

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

This will build a file called '`gnugo-3.0.x/TAGS`'. Now to find any GNU Go function, type `M-.` and enter the command which you wish to find, or just `RET` if the cursor is at the name of the function sought.

The first time you do this you will be prompted for the location of the TAGS table. Enter the path to '`gnugo-3.0.x/TAGS`', and henceforth you will be able to find any function with a minimum of keystrokes.

5 Analyzing GNU Go's moves

In this chapter we will discuss methods of finding out how GNU Go understands a given position. These methods will be of interest to anyone working on the program, or simply curious about its workings.

We assume that you have a game GNU Go played saved as an sgf file, and you want to know why it made a certain move.

5.1 Interpreting Traces

A quick way to find out roughly the reason for a move is to run

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

(You may also want to add '`--quiet`' to suppress the copyright message.) In GNU Go 3.0, the moves together with their reasons are listed, followed by a numerical analysis of the values given to each move.

If you are tuning (see Section 12.11 [Tuning], page 106) you may want to add the '`-a`' option. This causes GNU Go to report all patterns matched, even ones that cannot affect the outcome of the move. The reasons for doing this is that you may want to modify a pattern already matched instead of introducing a new one.

If you use the '`-w`' option, GNU Go will report the statuses of dragons around the board. This type of information is available by different methods, however (see Section 5.6 [Debugboard], page 32, see Section 5.8 [Colored Display], page 33).

5.2 The Output File

If GNU Go is invoked with the option '`-o filename`' it will produce an output file. This option can be added at the command line in the Go Modem Protocol Setup Window of CGoban. The output file will show the locations of the moves considered and their weights. It is worth noting that by enlarging the CGoban window to its fullest size it can display 3 digit numbers. Dragons with status `DEAD` are labelled with an '`X`', and dragons with status `CRITICAL` are labelled with a '`!`'.

If you have a game file which is not commented this way, or which was produced by a non-current version of GNU Go you may ask GNU Go to produce a commented version by running:

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

Here <color> can be 'black,' 'white' or 'both'. The replay option will also help you to find out if your current version of GNU Go would play differently than the program that created the file.

5.3 Checking the reading code

The '`--decide-string`' option is used to check the tactical reading code (see Chapter 14 [Tactical Reading], page 122). This option takes an argument, which is a location on the board in the usual algebraic notation (e.g. '`--decide-string C17`'). This will tell you whether the reading code (in '`engine/reading.c`') believes the string can be captured, and if so, whether it believes it can be defended, which moves it finds to attack or defend the move, how many nodes it searched in coming to these conclusions. Note that when GNU Go runs normally (not with '`--decide-string`') the points of attack and defense are computed when `make_worms()` runs and cached in `worm.attack` and `worm.defend`.

If used with an output file ('`-o` *filename*') '`--decide-string`' will produce a variation tree showing all the variations which are considered. This is a useful way of debugging the reading code, and also of educating yourself with the way it works. The variation tree can be displayed graphically using CGoban.

At each node, the comment contains some information. For example you may find a comment:

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

This is to be interpreted as follows. The node in question was generated by the function `attack3()` in '`engine/reading.c`', which was called on the string at `D12`. Of the data in parentheses tells you the values of `count_variations` and `hashdata.hashval`.

The second value ("hash") you probably will not need to know unless you are debugging the hash code, and we will not discuss it. But the first value ("variation") is useful when using the debugger `gdb`. You can first make an output file using the '`-o`' option, then walk through the reading with `gdb`, and to coordinate the SGF file with the debugger, display the value of `count_variations`. Specifically, from the debugger you can find out where you are as follows:

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

If you place yourself right after the call to `trymove()` which generated the move in question, then the variation number in the SGF file should match the variation number displayed by `dump_stack()`, and the move in question will be the last move played (F11 in this example).

This displays the sequence of moves leading up to the variation in question, and it also prints `count_variations-1`.

The second two lines tell you that from this node, the function `break_chain()` was called at D12 and returned 0 meaning that no way was found of rescuing the string by attacking an element of the surrounding chain, and the function `defend3()` was called also at D12 and returned 1, meaning that the string can be defended, and that G12 is the move that defends it. If you have trouble finding the function calls which generate these comments, try setting `sgf_dumptree=1` and setting a breakpoint in `sgf_trace`.

5.4 Checking the Owl code

You can similarly debug the Owl code using the option '`--decide-dragon`'. Usage is entirely similar to '`--decide-string`', and it can be used similarly to produce variation trees. These should be typically much smaller than the variation trees produced by '`--decide-string`'.

5.5 GTP and GDB techniques

You can use the Go Text Protocol (see Chapter 20 [GTP], page 164) to determine the statuses of dragons and other information needed for debugging. The GTP command `dragon_data P12` will list the dragon data of the dragon at `P12` and `worm_data` will list the worm data; other GTP commands may be useful as well.

You can also conveniently get such information from GDB. A suggested '`.gdbinit`' file may be found in See Section 14.7 [Debugging], page 138. Assuming this file is loaded, you can list the dragon data with the command:

```
(gdb) dragon P12
```

Similarly you can get the worm data with `worm P12`.

5.6 Debugboard

A useful utility called `debugboard` is made in the '`interface/debugboard/`' directory. This can be run in an Xterm. Use a smaller font since it requires 50 rows and 80 columns. This runs `examine_position()`, then makes a graphical display of the board. Using the cursor movement keys, you can move around the board and find out the contents of the worm, dragon and eye arrays.

5.7 Scoring the game

GNU Go can score the game. If done at the last move, this is usually accurate unless there is a seki. Normally GNU Go will report its opinion about the score at the end of the game, but if you want this information about a game stored in a file, use the '`--score`' option.

```
gnugo --score last -l filename
```

loads the sgf file to the end of the file and estimates the winner after the last stored move by estimating the territory.

```
gnugo --score end -l filename
```

loads the sgf file and GNU Go continues to play after the last stored move by itself up to the very end. Then the winner is determined by estimating the territory.

```
gnugo --score aftermath -l filename
```

loads the sgf file and GNU Go continues to play after the last stored move by itself up to the very end. Then the winner is determined by the most accurate algorithm available. Slower but more accurate than '`--score end`'.

```
gnugo --score L10 -l filename
```

loads the sgf file until a stone is placed on L10. Now the winner will be estimated as with `gnugo --score last`.

Any of these commands may be combined with '`--chinese-rules`' if you want to use Chinese (area) counting.

```
gnugo --score 100 -l filename
```

loads the sgf file until move number 100. Now the winner will be estimated as with `gnugo --score last`.

If the option '`-o` *outputfilename*' is provided, the results will also be written as comment at the end of the output file.

5.8 Colored Display

Various colored displays of the board may be obtained in a color `xterm` or `rxvt` window. Xterm will only work if xterm is not compiled with color support. If the colors are not displayed on your xterm, try `rxvt`. You may also use the Linux console. The colored display will work best if the background color is black; if this is not the case you may want to edit your '`.Xdefaults`' file or add the options '`-bg black -fg white`' to `xterm` or `rxvt`.

5.8.1 Dragon Display

You can get a colored ASCII display of the board in which each dragon is assigned a different letter; and the different `matcher_status` values (`ALIVE`, `DEAD`, `UNKNOWN`, `CRITICAL`) have different colors. This is very handy for debugging. Actually two diagrams are generated. The reason for this is concerns the way the matcher status is computed. The dragon_status (see Section 10.7 [Dragons], page 76) is computed first, then for some, but not all dragons, a more accurate owl status is computed. The matcher status is the owl status if available; otherwise it is the dragon_status. Both the dragon_status and the owl_status are displayed. The color scheme is as follows:

```
green = alive
cyan = dead
red = critical
yellow = unknown
magenta = unchecked
```

To get the colored display, save a game in sgf format using CGoban, or using the '`-o`' option with GNU Go itself.

Open an `xterm` or `rxvt` window.

Execute `gnugo -l [filename] -L [movenum] -T` to get the colored display.

Other useful colored displays may be obtained by using instead:

5.8.2 Eye Space Display

Instead of '`-T`', try this with '`-E`'. This gives a colored display of the eyespaces, with marginal eye spaces marked '`!`' (see Chapter 11 [Eyes], page 82).

5.8.3 Moyo Display

The option '`-m` *level*' can give colored displays of the various quantities which are computed in '`engine/moyo.c`'.

The moyos found by GNU Go can be displayed from an xterm or rxvt window or from the Linux console using the '`-m`' option. This takes a parameter:

'`-m` *level*'

```
 use or (hexadecimal)   cumulative values for printing these reports :
```

1	0x01	ascii printing of territorial evaluation (5/21)
2	0x02	ascii printing of moyo evaluation (5/10)
4	0x04	ascii printing of area (4/0)
8	0x08	print initial moyo influence
16	0x10	print influence
32	0x20	numeric influence
64	0x40	moyo strength
128	0x80	moyo attenuation

The first three options are somewhat superceded because these data are no longer used by the engine.

These options can be combined by adding the levels. Levels 16, 32, 64 and 128 don't do much unless you also specify level 8. Thus one might use the hexadecimal option '`-m0x018`' if you want to see the influence function displayed graphically.

See Chapter 17 [Moyo], page 152, for the first three items.

See Section 16.7 [Influential Display], page 150, for the last five items.

6 Application Programmers Interface to GNU Go

If you want to write your own interface to GNU Go, or if you want to create a go application using the GNU Go engine, this chapter is of interest to you.

First an overview: GNU Go consists of two parts: the GNU Go *engine* and a program (user interface) which uses this engine. These are linked together into one binary. The current program implements the following user modes:

- An interactive board playable on ASCII terminals
- solo play - GNU Go plays against itself
- replay - a mode which lets the user investigate moves in an existing SGF file.
- GMP - Go Modem Protocol, a protocol for automatic play between two computers.
- GTP - Go Text Protocol, a more general go protocol currently used only for testing of the engine. However, GTP is currently being standardized and it is expected that GTP will become the main choice for tasks where currently GMP is used.

The GNU Go engine can be used in other applications. For example, supplied with GNU Go is another program using the engine, called '`debugboard`', in the directory '`interface/debugboard/`'. The program debugboard lets the user load SGF files and can then interactively look at different properties of the position such as group status and eye status.

The purpose of this Chapter is to show how to interface your own program such as `debugboard` with the GNU Go engine.

Figure 1 describes the structure of a program using the GNU Go engine.

```
+-----------------------------------+
|                                   |
|          Go application           |
|                                   |
+-----+----------+------+           |
|     |          |      |           |
|     |   Game   |      |           |
|     | handling |      |           |
|     |          |      |           |
|     +----+-----+      |           |
|   SGF    |    Move    |           |
| handling | generation |           |
|          |            |           |
+----------+------------+-----------+
|                                   |
|           Board handling          |
|                                   |
+-----------------------------------+
```

`Figure 1: The structure of a program using the GNU Go engine`

The foundation is a library called `libboard.a` which provides efficient handling of a go board with rule checks for moves, with incremental handling of connected strings of stones and with methods to efficiently hash go positions.

On top of this, there is a library which helps the application use smart go files, SGF files, with complete handling of game trees in memory and in files. This library is called `libsgf.a`

The main part of the code within GNU Go is the move generation library which given a position generates a move. This part of the engine can also be used to manipulate a go position, add or remove stones, do tactical and strategic reading and to query the engine for legal moves. These functions are collected into `libengine.a`.

The game handling code helps the application programmer keep tracks of the moves in a game, and to undo or redo moves. Games can be saved to SGF files and then later be read back again. These are also within `libengine.a`.

The resposibility of the application is to provide the user with a user interface, graphical or not, and let the user interact with the engine.

6.1 How to use the engine in your own program: getting started

To use the GNU Go engine in your own program you must include the file '`gnugo.h`'. This file describes the whole public API. There is another file, '`liberty.h`', which describes the internal interface within the engine. If you want to make a new module within the engine, e.g. for suggesting moves you will have to include this file also. In this section we will only describe the public interface.

Before you do anything else, you have to call the function `init_gnugo()`. This function initializes everything within the engine. It takes one parameter: the number of megabytes the engine can use for the internal hash table. In addition to this the engine will use a few megabytes for other purposes such as data describing groups (liberties, life status, etc), eyes and so on.

6.2 Basic Data Structures in the Engine

There are some basic definitions in gnugo.h which are used everywhere. The most important of these are the numeric declarations of colors. Each intersection on the board is represented by one of these:

color	value
EMPTY	0
WHITE	1
BLACK	2

In addition to these, the following values can be used in special places, such as describing the borders of eyes:

color	value
GRAY (GRAY_BORDER)	3
WHITE_BORDER	4
BLACK_BORDER	5

There is a macro, `OTHER_COLOR(color)` which can be used to get the other color than the parameter. This macro can only be used on `WHITE` or `BLACK`, but not on `EMPTY` or one of the border colors.

6.3 The Position Struct

The basic data structure in the interface to the engine is the `Position`. A `Position` is used to store the current position of a game including the location of all black and white stones, a possible ko, and the number of captured stones on each side. Here is the definition of `Position`:

```
typedef unsigned char Intersection;

typedef struct {
  int          boardsize;
  Intersection board[MAX_BOARD][MAX_BOARD];
  int          ko_i;
  int          ko_j;
  int          last_i[2];
  int          last_j[2];

  float        komi;
  int          white_captured;
  int          black_captured;
} Position;
```

Here `Intersection` stores `EMPTY`, `WHITE` or `BLACK`. It is currently defined as an `unsigned char` to make it reasonably efficient in both storage and access time. The position stores a two-dimensional array of Intersections with the size `MAX_BOARD`. `MAX_BOARD` is the value of the biggest board size that the engine supports; it is currently set to 21. There is also a `MIN_BOARD` which is set to 3.

To indicate what board size is actually used, there is a member, `boardsize`, which should be in the range between `MIN_BOARD` and `MAX_BOARD`.

A location on the board is represented by a pair of integers in the range `[0 ... boardsize-1]`. The convention used within GNU Go is that the first integer indicates the row number from the top and the second integer indicates the column number from the left. Thus the coordinate (2,5) is F5 (A) in the small diagram below.

```
  A B C D E F G
7 . . . . . . . 7
6 . . . . . . . 6
5 . . . . . A . 5
4 . . . . . . . 4
3 . . . . . . . 3
2 . . . . . . . 2
1 . . . . . . . 1
  A B C D E F G
```

A pass move is represented by the pair `(-1,-1)`. A convention within the code is to use the suffix '`i`' and '`j`' for the first and the last coordinate.

If there is a ko present on the board, that is if one stone was captured the last move and the capturing stone can be recaptured, the pair `(ko_i, ko_j)` points at the empty intersection where the stone was just captured ('`a`' in the diagram below).

```
  A B C D E F G
7 . . . . . . . 7
6 . . . . . . . 6
5 . . . O X . . 5
4 . . O a O X . 4
3 . . . O X . . 3
2 . . . . . . . 2
1 . . . . . . . 1
  A B C D E F G
```

If no ko is present, `ko_i` should be set to `-1`.

The last two moves played are stored in `(last_i[], last_j[])`.

As the game progresses the number of prisoners on each side are maintained in the members `white_captured` and `black_captured`.

The komi used in the ongoing game is also stored in the `Position`. The reason for this is that in some instances, GNU Go plays differently whether it is ahead, behind or the position is even. So the komi is an important input to the move generation.

6.4 Functions which manipulate a Position

All the functions in the engine that manipulate Positions have names prefixed by `gnugo_`. Here is a complete list, as defined in '`gnugo.h`':

6.4.1 Functions which manipulate the go position

`void gnugo_clear_position(Position *pos, int boardsize, float komi)`

Clear the position setting the board size to `boardsize` and the komi to `komi`.

`void gnugo_copy_position(Position *dest, Position *src)`

Copy position `src` to position `dest`. This is the same convention that is used in `memcpy(3)`.

`void gnugo_add_stone(Position *pos, int i, int j, int color)`

Add a stone of `color` at `(i,j)` to the position.

`void gnugo_remove_stone(Position *pos, int i, int j)`

Remove the stone at `(i,j)` from the position. No check is done that there actually is a stone there.

`void gnugo_play_move(Position *pos, int i, int j, int color)`

Play a stone of color color at `(i, j)` in the position removing captured stones if any. No check is done if the move is legal; to do that, call `gnugo_is_legal()`. Suicide is legal.

`int gnugo_play_sgfnode(Position *pos, SGFNode *node, int to_move)`

Place all the stones in and play all the moves in the SGF node `node` (see Chapter 7 [SGF], page 43.) Return whose turn it is to move after this is done.

`int gnugo_play_sgftree(Position *pos, SGFNode *root, int *until, SGFNode **curnode)`

Clear the position and play through the moves in SGF tree `root` until the move number `until` has been reached. Return whose turn it is to move after this is done. The parameter `curnode` will be set to the current node in the tree, i.e. the one which was played last.

`int gnugo_is_legal(Position *pos, int i, int j, int color)`

Return 1 if the move at `(i,j)` would be legal; otherwise return 0. The rule set used is standard japanese rules where suicide is illegal. If there is a ko point set (`ko_i != -1`), then the ko point is also illegal to play on.

`int gnugo_is_suicide(Position *pos, int i, int j, int color)`

Return 1 if the move at `(i,j)` would be suicide; otherwise return 0.

`int gnugo_placehand(Position *pos, int handicap)`

Sets up handicap stones, returning the number of placed handicap stones. Maximum handicap supported is 0 for board sizes below 7, 4 for board sizes 7 or 8 and 9 for board sizes from 9 and up.

`int gnugo_sethand(Position *pos, int handicap, struct SGFNode_t *root)`

Sets up handicap pieces and returns the number of placed handicap stones, updating the SGF file.

`void gnugo_recordboard(Position *pos, struct SGFNode_t *node)`

Records the position in the SGF node (see Chapter 7 [SGF], page 43).

`int gnugo_genmove(Position *pos, int *i, int *j, int color, int move_number)`

Generate a move for color `color` and return it in `(*i,*j)`. The parameter `move_number` is the number of the current move. This is mostly used for debugging reasons, as the game handling functions all work on top of the move generation part of the engine. (see Section 4.2 [Move Generation Basics], page 19.).

`float gnugo_estimate_score(Position *pos, float *upper, float *lower)`

> Evaluate the approximate score. The score is given as an interval with a lower and upper bound. A positive score means that white is leading, while a negative score is good for black. When the lower bound is estimated, CRITICAL dragons are awarded to white; when estimating the lower bound, they are awarded to black.
>
> The estimation is returned through the pointers `*upper` and `*lower`, and the mean between them is returned as the functions value.

`void gnugo_who_wins(Position *pos, int color, FILE *outfile)`

> Score the game and determine the winner.

6.4.2 Status functions

These functions examines the position in different ways and tells the status of groups and other items.

`int gnugo_attack(Position *pos, int m, int n, int *i, int *j)`

> Calls the tactical reading function `attack` to determine whether the string at `(m, n)` can be captured (see Chapter 14 [Tactical Reading], page 122).

`int gnugo_find_defense(Position *pos, int m, int n, int *i, int *j)`

> Calls the tactical reading function `find_defense` to determine whether the string at `(m, n)` can be rescued (see Chapter 14 [Tactical Reading], page 122).

6.4.3 Special functions

These functions are only used in special situations, such as when the program wants to access internal data structures within the engine. They should only be used when the programmer has a good knowledge of the internals of GNU Go.

`void gnugo_force_to_globals(Position *pos)`

> Put the values in `pos` into the global variables which is the equivalent of the `Position`.

`void gnugo_examine_position(Position *pos, int color, int how_much)`

> Calls `examine_position()`, doing much prelimary analysis of the board position (see Section 4.2 [Move Generation Basics], page 19).

6.5 Game handling

The functions (in see Section 6.4 [Positional Functions], page 38) are all that are needed to create a fully functional go program. But to make the life easier for the programmer, there is a small set of functions specially designed for handling ongoing games.

The data structure describing an ongoing game is the `Gameinfo`. It is defined as follows:

```
typedef struct {
  int       handicap;

  Position  position;
  int       move_number;
  int       to_move; /* whose move it currently is */
  SGFTree   moves; /* The moves in the game. */

  int       seed; /* random seed */
  int       computer_player; /* BLACK, WHITE, or EMPTY (used as BOTH) */

  char      outfilename[128]; /* Trickle file */
  FILE      *outfile;
} Gameinfo;
```

The meaning of `handicap` should be obvious. The `position` field is of course the current position, `move_number` is the number of the current move and `to_move` is the color of the side whose turn it is to move.

The SGF tree `moves` is used to store all the moves in the entire game, including a header node which contains, among other things, komi and handicap. If a player wants to undo a move, this can most easily be done by replaying all the moves in the tree except for the last one. This is the way it is implemented in `gameinfo_undo_move()`.

If one or both of the opponents is the computer, the fields `seed` and `computer_player` are used. Otherwise they can be ignored. `seed` is used to store the number used to seed the random number generator. Given the same moves from the opponent, GNU Go will try to vary its game somewhat using a random function. But if the random generator is given the same seed, GNU Go will always play the same move. This is good, e.g. when we debug the engine but could also be used for other purposes.

GNU Go can use a trickle file to continuously save all the moves of an ongoing game. This file can also contain information about internal state of the engine such as move reasons for various locations or move valuations for the 10 highest valued moves. The name of this file should be stored in `outfilename` and the file pointer to the open file is stored in `outfile`. If no trickle file is used, `outfilename[0]` will contain a null character and `outfile` will be set to `NULL`.

6.5.1 Functions which manipulate a Gameinfo

All the functions in the engine that manipulate Gameinfos have names prefixed by `gameinfo_`. Here is a complete list, as defined in '`gnugo.h`':

`void gameinfo_clear(Gameinfo *ginfo, int boardsize, float komi)`

Clear the Gameinfo to an empty state. The board size of the `Position` is set to `boardsize`.

`void gameinfo_print(Gameinfo *ginfo)`

Print the Gameinfo on stdout. This is mostly a debug tool.

`void gameinfo_load_sgfheader(Gameinfo *ginfo, SGFNode *head)`

Load header information from the SGF node `head` and set the appropriate variables in `ginfo`.

`void gameinfo_play_move(Gameinfo *ginfo, int i, int j, int color)`

Play a move at (`i`, `j`), record it in `moves`, print it to the trickle file if any and update `move_number` and `to_move`.

`void gameinfo_undo_move(Gameinfo *ginfo)`

Replays all the moves of the game except the last one. It also updates `move_number`, `to_move` and `moves`. If there is a trickle file, it is truncated to the second to last move.

FIXME: Not yet implemented.

`int gameinfo_play_sgftree(Gameinfo *ginfo, SGFNode *head, const char *untilstr)`

Read header information and play the main variation in the SGF tree starting with `head`. Return whose turn it is to move after this is done.

The parameter `untilstr` is an optional string of the form 'L12' (a board position) or '120' (a move number) which tells the function to stop playing at that move or move number.

7 Handling SGF trees in memory

SGF - Smart Game Format - is a file format which is used for storing game records for a number of different games, among them chess and go. The format is a framework with special adaptions to each game. This is not a description of the file format standard. Too see the exact definition of the file format, see `http://www.red-bean.com/sgf/`.

GNU Go contains a library to handle go game records in the SGF format in memory and to read and write SGF files. This library - `libsgf.a` - is in the `sgf` subdirectory. To use the SGF routines, include the file `sgftree.h`.

Each game record is stored as a tree of *nodes*, where each node represents a state of the game, often after some move is made. Each node contains zero or more *properties*, which gives meaning to the node. There can also be a number of *child nodes* which are different variations of the game tree. The first child node is the main variation.

Here is the definition of `SGFNode`, and `SGFProperty`, the data structures which are used to encode the game tree.

```
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;
```

Each node of the SGF tree is stored in an `SGFNode` struct. It has a pointer to a linked list of properties (see below) called `props`. It also has a pointer to a linked list of children, where each child is a variation which starts at this node. The variations are linked through the `next` pointer and each variation continues through the `child` pointer. Each and every node also has a pointer to its parent node (the `parent` field), except the top node whose parent pointer is `NULL`.

An SGF property is encoded in the `SGFPoperty` struct. It is linked in a list through the `next` field. A property has a `name` which is encoded in a short int. Symbolic names of properties can be found in '`sgf_properties.h`'.

Some properties also have a value, which could be an integer, a floating point value, a character or a string. These values can be accessed or set through special functions (see below).

7.1 Functions which manipulate SGF nodes and properties

All the functions which create and manipulate SGF trees are prefixed by `sgf`. The SGF code was donated to us by Thomas Traber, so they don't follow the naming conventions of GNU Go perfectly.

7.1.1 Low level functions

These functions let the caller create nodes or access nodes easier.

`SGFNode *sgfNewNode(void)`
: Allocate and return a new instance of `SGFNode`. The node is cleared.

`SGFProperty *sgfMkProperty(const char *name, const char *value, SGFNode *node, SGFProperty *last)`
: Allocate and return a new instance of `SGFProperty`. The `name` should be 1 or 2 characters long. This function should probably not be used directly. Instead, use the `sgfAddProperty` functions.

`SGFNode *sgfPrev(SGFNode *node)`
: Return the previous node in a chain. This is done by going to the parent node and then search through the children until the same node is found. If there is no previous node, `NULL` is returned.

`SGFNode *sgfRoot(SGFNode *node)`
: Return the root of the tree. If `node` already is the root, `node` itself is returned.

7.1.2 Functions which manipulate SGF properties

`int sgfGetIntProperty(SGFNode *node, const char *name, int *value)`
: Get the property `name` in `node` as an integer. The value is returned in `value`. Returns 1 if successful, otherwise returns 0.

`int sgfGetFloatProperty(SGFNode *node, const char *name, float *value)`
: Get the property `name` in `node` as a floating point value. The value is returned in `value`. Returns 1 if successful, otherwise returns 0.

`int sgfGetCharProperty(SGFNode *node, const char *name, char **value)`
: Get the property `name` in `node` as a string of characters. The value is returned in `value`. Returns 1 if successful, otherwise returns 0.

`void sgfAddProperty(SGFNode *node, const char *name, const char *value)`
: Add a new property to `node`. There is no check to see if there already is a property with the same name. The property value has to be a character string.

`void sgfAddPropertyInt(SGFNode *node, const char *name, long val)`
: Add an integer property to `node`. This function converts the value to a string and calls `sgfAddProperty`.

`void sgfAddPropertyFloat(SGFNode *node, const char *name, float val)`
: Add a floating point property to `node`. This function converts the value to a string and calls `sgfAddProperty`.

`void sgfOverwriteProperty(SGFNode *node, const char *name, const char *text)`

Overwrite the property `name` in `node` with the string `text`. If the property does not yet exist in `node`, it is added using `sgfAddProperty`.

`void sgfOverwritePropertyInt(SGFNode *node, const char *name, int value)`

Overwrite the property `name` in `node` with the integer `value`. If the property does not yet exist in `node`, it is added using `sgfAddPropertyInt`.

`void sgfOverwritePropertyFloat(SGFNode *node, const char *name, float value)`

Overwrite the property `name` in `node` with the floating point number `value`. If the property does not yet exist in `node`, it is added using `sgfAddPropertyFloat`.

7.1.3 Functions which manipulate SGF nodes

`SGFNode *sgfAddStone(SGFNode *node, int color, int movex, int movey)`

Add a stone to `node`. Properties added is either `AB` (black stone) or `AW` (white stone).

`SGFNode *sgfAddPlay(SGFNode *node, int who, int movex, int movey)`

Add a child node with a move to `node`. Properties added is either `B` (black move) or `W` (white move). A pass is coded by `(-1, -1)`.

This function does not add a property to the node itself, but adds a child node instead. If there are previous child nodes, the new node is placed before the other ones, so this function should be used if you want to add a main branch to the tree. To add a variation, use `sgfAddPlayLast` instead.

`SGFNode *sgfAddPlayLast(SGFNode *node, int who, int movex, int movey)`

Add a child node with a move to `node`. Properties added is either `B` (black move) or `W` (white move). A pass is coded by `(-1, -1)`.

If there are previous child nodes in `node`, the move is added by adding the child node last, so this function should be used when you want to add a variation to the game tree.

`int sgfPrintCharProperty(FILE *file, SGFNode *node, const char *name)`

Print the properties of type `name` in `node` on `file`.

`int sgfPrintCommentProperty(FILE *file, SGFNode *node, const char *name)`

Print the comment properties of type `name` in `node` on `file`.

`void sgfWriteResult(SGFNode *node, float score, int overwrite)`

Add a `RE` (result) property to `node`. This property will contain the game result. If `overwrite` is zero the result is written only if no previous result property exists.

`SGFNode *sgfCircle(SGFNode *node, int i, int j)`

Add a `CR` (circle) property at `(i, j)` to `node`.

`SGFNode *sgfSquare(SGFNode *node, int i, int j)`

Calls `sgfMark` to add a `MA` (mark) property at `(i, j)` to `node`.

`SGFNode *sgfTriangle(SGFNode *node, int i, int j)`
: Add a `TR` (triangle) property at `(i, j)` to `node`.

`SGFNode *sgfMark(SGFNode *node, int i, int j)`
: Add a `MA` (mark) property at `(i, j)` to `node`.

`SGFNode *sgfAddComment(SGFNode *node, const char *comment)`
: Add a `C` (comment) property to `node`.

`SGFNode *sgfBoardText(SGFNode *node, int i, int j, const char *text)`
: Add a `LB` (label) property at `(i, j)` to `node`.

`SGFNode *sgfBoardChar(SGFNode *node, int i, int j, char c)`
: Add a `LB` (label) property at `(i, j)` to `node`. This functions is a utility function that converts the character to a string and calls `sgfBoardText`.

`SGFNode *sgfBoardNumber(SGFNode *node, int i, int j, int number)`
: Add a numeric label at `(i, j)` by calling `sgfBoardText`.

`SGFNode *sgfStartVariant(SGFNode *node)`
: Start a new variation in the game tree. This means that the `next` pointer of `node` is followed to the end of the list and a new node is inserted there. A pointer to the new node is returned.

`SGFNode *sgfStartVariantFirst(SGFNode *node)`
: Same as `sgfStartVariant`, except that the node is placed first in the list. This means that the new variation will be the main variation of the game tree. Returns a pointer to the new node.

`SGFNode *sgfAddChild(SGFNode *node)`
: Adds a child node to `node`. If there already are children, the new node is placed last in the list. Returns a pointer to the new node.

7.1.4 High level functions

`SGFNode *sgfCreateHeaderNode(int boardsize, float komi)`
: Create a new SGF node with the two properties `SZ` (size) and `KM` (komi). More properties, like `HA` (handicap), can later be added to it.

  The idea with this node is to store the game info and to use as a root node for the game.

`SGFNode *readsgffile(const char *filename)`
: Read an SGF file and return the resulting tree.

`void sgf_write_header(SGFNode *root, int overwrite, int seed, float komi)`
: Write random seed, date, ruleset, komi and SGF file version to the header node `root`. If `overwrite` is non-zero, it overwrites the values in the node, otherwise it just writes those that are missing.

  Ruleset is always set to "Japanese", date is set to the current date.

`int writesgf(SGFNode *root, const char *filename)`
: Write the tree starting in `root` to the file `filename`. If `filename` is `-`, the tree is written to `stdout`. Returns 1 if successful, otherwise returns 0.

7.2 The SGFTree datatype

Sometimes we just want to record an ongoing game or something similarly simple and not do any sofisticated tree manipulation. In that case we can use the simplified interface provided by `SGFTree` below.

```
typedef struct SGFTree_t {
  SGFNode *root;
  SGFNode *lastnode;
} SGFTree;
```

An `SGFTree` contains a pointer to the root node of an SGF tree and a pointer to the node that we last accessed. Most of the time this will be the last move of an ongoing game.

Most of the functions which manipulate an `SGFTree` work exactly like their `SGFNode` counterparts, except that they work on the current node of the tree.

All the functions below that take arguments `tree` and `node` will work on:

1. `node` if non-`NULL`
2. `tree->lastnode` if non-`NULL`
3. The current end of the game tree.

in that order.

7.2.1 Functions that manipulate sgftrees

`void sgftree_clear(SGFTree *tree)`

Clear the `root` and `lastnode` pointers of `tree`. '`NOTE:`' This function does not free any memory. That has to be done separately.

`int sgftree_readfile(SGFTree *tree, const char *infilename)`

Read an SGF file with the name `infilename` and store it in `tree`. Return 1 if successful, otherwise return 0. `lastnode` will be set to `NULL`.

`SGFNode *sgftreeNodeCheck(SGFTree *tree, SGFNode *node)`

Return the node to work on as described above. This is:

1. `node` if non-`NULL`
2. `tree->lastnode` if non-`NULL`
3. The current end of the tree.

in that order.

`SGFNode *sgftreeAddPlay(SGFTree *tree, SGFNode *node, int color int movex, int movey)`

Add a move of `color` at `(movex,movey)` to the tree. See [sgfAddPlay], page 45.

`SGFNode *sgftreeAddPlayLast(SGFTree *tree, SGFNode *node, int color, int movex, int movey)`

Add a variation of `color` at `(movex,movey)` to the tree. See [sgfAddPlay-Last], page 45.

`SGFNode *sgftreeAddStone(SGFTree *tree, SGFNode *node, int color, int movex, int movey)`

Add a stone of `color` at `(movex,movey)` to the tree.

`void sgftreeWriteResult(SGFTree *tree, float score, int overwrite)`

Add the result to the tree. If there already is a result, only overwrite it if `overwrite` is non-zero.

`SGFNode *sgftreeCircle (SGFTree *tree, SGFNode *node, int i, int j)`

Add a circle property at `(i, j)` to the tree.

`SGFNode *sgftreeSquare (SGFTree *tree, SGFNode *node, int i, int j)`

Add a square property at `(i, j)` to the tree.

`SGFNode *sgftreeTriangle(SGFTree *tree, SGFNode *node, int i, int j)`

Add a triangle property at `(i, j)` to the tree.

`SGFNode *sgftreeMark(SGFTree *tree, SGFNode *node, int i, int j)`

Add a mark property at `(i, j)` to the tree.

`SGFNode *sgftreeAddComment(SGFTree *tree, SGFNode *node, const char *comment)`

Add a comment property to the tree. This is a property of the node itself, and has no position on the board.

`SGFNode *sgftreeBoardText(SGFTree *tree, SGFNode *node, int i, int j, const char *text)`

Add a text property at `(i, j)` to the tree.

`SGFNode *sgftreeBoardChar(SGFTree *tree, SGFNode *node, int i, int j, char c)`

Add a character at `(i, j)` to the tree.

`SGFNode *sgftreeBoardNumber(SGFTree *tree, SGFNode *node, int i, int j, int number)`

Add a number at `(i, j)` to the tree.

`SGFNode *sgftreeStartVariant(SGFTree *tree, SGFNode *node)`

Start a new variation in the tree. See [sgfStartVariant], page 46.

`SGFNode *sgftreeStartVariantFirst(SGFTree *tree, SGFNode *node)`

Start a new main variation in the tree. See [sgfStartVariantFirst], page 46.

`SGFNode *sgftreeCreateHeaderNode(SGFTree *tree, int boardsize, float komi)`

Add a header node first in `tree`.

`void sgftreeSetLastNode(SGFTree *tree, SGFNode *last_node)`

Explicitly set the last accessed node in `tree` to `last_node`.

8 The Board Library

The foundation of the GNU Go engine is a library of very efficient routines for handling go boards. This board library, called ‘`libboard`’, can be used for those programs that only need a basic go board but no AI capability. One such program is `patterns/joseki` subdirectory, which compiles joseki pattern databases from SGF files.

The library consists of the following files:

‘`board.c`’

The basic board code. It uses incremental algorithms for keeping track of strings and liberties on the go board.

‘`hash.c`’

Code for hashing go positions.

‘`cache.c`’

Code for caching go positions

‘`globals.c`’

Global variables needed in the rest of the files. This file also contains global variables needed in the rest of the engine.

‘`sgffile.c`’

Implementation of output file in SGF format.

‘`showbord.c`’

Print go boards.

‘`printutils.c`’

Utilities for printing go boards and other things.

To use the board library, you must include ‘`liberty.h`’ just like when you use the whole engine, but of course you cannot use all the functions declared in it, i.e. the functions that are part of the engine, but not part of the board library. You must link your application with `libboard.a`.

8.1 Board Data structures

The basic data structures of the board correspond tightly to the `Position` struct described in See Section 6.3 [The Position Struct], page 37. They are all stored in global variables for efficiency reasons, the most important of which are:

```
int           board_size;
Intersection  p[MAX_BOARD][MAX_BOARD];
int           board_ko_i;
int           board_ko_j;
int           last_moves_i[2];
int           last_moves_j[2];

float         komi;
int           white_captured;
int           black_captured;

Hash_data     hashdata;
```

The description of the `Position` struct is applicable to these variables also, so we won't duplicate it here. All these variables are globals for performance reasons. Behind these variables, there are a number of other private data structures. These implement incremental handling of strings, liberties and other properties (see Chapter 19 [Incremental Board], page 161). The variable `hashdata` contains information about the hash value for the current position (see Section 14.2 [Hashing], page 125).

These variables should never be manipulated directly, since they are only the front end for the incremental machinery. They can be read, but should only be written by using the functions described in the next section. If you write directly to them, the incremental data structures will become out of sync with each other, and a crash is the likely result.

8.2 Board Functions

These functions are all the public functions in '`engine/board.c`'.

8.2.1 Setup Functions

These functions are used when you want to set up a new position without actually playing out moves.

- `void clear_board()`

  Clears the internal board (`p[][]`), resets the ko position, captured stones and recalculates the hash value.

- `void setup_board(Intersection new_p[MAX_BOARD][MAX_BOARD], int koi, int koj, int *last_i, int *last_j, float new_komi, int w_captured, int b_captured)`

  Set up a new board position using the parameters.

- `void add_stone(int i, int j, int color)`

  Place a stone on the board and update the hashdata. No captures are done.

- `void remove_stone(int i, int j)`

  Remove a stone from the board and update the hashdata.

8.2.2 Move Functions

Reading, often called **search** in computer game theory, is a fundamental process in GNU Go. This is the process of generating hypothetical future boards in order to determine the answer to some question, for example "can these stones live." Since these are hypothetical future positions, it is important to be able to undo them, ultimately returning to the present board. Thus a move stack is maintained during reading. When a move is tried, by the function `trymove`, or its variant `tryko`. This function pushes the current board on the stack and plays a move. The stack pointer `stackp`, which keeps track of the position, is incremented. The function `popgo()` pops the move stack, decrementing `stackp` and undoing the last move made.

Every successful `trymove()` must be matched with a `popgo()`. Thus the correct way of using this function is:

```
if (trymove(i, j, color, [message], k, l, komaster, kom_i, kom_j)) {
     ...    [potentially lots of code here]
     popgo();
}
```

Here the `komaster` is only set if a conditional ko capture has been made at an earlier move. This feature of the tactical and owl reading code in GNU Go is used to prevent redundant reading when there is a ko on the board (see Section 14.3 [Ko], page 131). If komaster is not defined then take the last three parameters to be `EMPTY`, and `-1, -1`.

- `void play_move(int i, int j, int color)`

  Play a move at `(i, j)`. If you want to test for legality you should first call `is_legal()`. This function strictly follows the algorithm:
  1. Place a stone of given color on the board.
  2. If there are any adjacent opponent strings without liberties, remove them and increase the prisoner count.
  3. If the newly placed stone is part of a string without liberties, remove it and increase the prisoner count.

- `int trymove(int i, int j, int color, const char *message, int k, int l, int komaster, int kom_i, int kom_j)`

  Returns true if `(i,j)` is a legal move for `color`. In that case, it pushes the board on the stack and makes the move, incrementing `stackp`. If the reading code is recording reading variations (as with '`--decide-string`' or with '`-o`'), the string `*message` will be inserted in the SGF file as a comment. The comment will also refer to the string at `(k,l)` if these are not `(-1,-1)`.

- `int TRY_MOVE()`

  Wrapper around trymove which suppresses `*message` and `(k,l)`. Used in '`helpers.c`'

- `int tryko(int i, int j, int color, const char *message, int komaster, kom_i, kom_j)`

  `tryko()` pushes the position onto the stack, and makes a move `(i, j)` of `color`. The move is allowed even if it is an illegal ko capture. It is to be imagined that `color` has made an intervening ko threat which was answered and now the continuation is to be explored. Return 1 if the move is legal with the above caveat. Returns zero if it is not legal because of suicide.

- `void popgo()`

  Pops the move stack. This function must (eventually) be called after a succesful `trymove` or `tryko` to restore the board position. It undoes all the changes done by the call to `trymove/tryko` and leaves the board in the same state as it was before the call.

  NOTE: If `trymove/tryko` returns `0`, i.e. the tried move was not legal, you must **not** call `popgo`.

- `int komaster_trymove(int i, int j, int color, const char *message, int si, int sj, int komaster, int kom_i, int kom_j, int *new_komaster, int *new_kom_i, int *new_kom_j, int *is_conditional_ko, int consider_conditional_ko)`

  Variation of `trymove/tryko` where ko captures (both conditional and unconditional) must follow a komaster scheme (see Section 14.3 [Ko], page 131).

- `int move_in_stack(int m, int n, int cutoff)`

  Returns true if at least one move been played at `(m, n)` at deeper than level 'cutoff' in the reading tree.

- `void get_move_from_stack(int k, int *i, int *j, int *color)`

  Retrieve the move number `k` from the move stack. The move location is returned in `(*i, *j)`, and the color that made the move is returned in `*color`.

- `void dump_stack(void)`

  Handy for debugging the reading code under GDB. Prints the move stack. Usage: `(gdb) set dump_stack()`.

- `void reset_trymove_counter()`

  Reset the trymove counter. This counter is incremented every time that a variant of `trymove` or `tryko` is called.

- `int get_trymove_counter()`

  Retrieve the trymove counter.

8.2.3 Status Functions

These functions are used for inquiring about properties of the current position or of potential moves.

- `int is_pass(int i, int j)`

  Returns true if the move `(i,j)` is a pass move, i.e. `(-1, -1)`.

- `int is_legal(int i, int j, int color)`

  Returns true if a move at `(i,j)` is legal for `color`.

- `int is_ko(int m, int n, int color, int *ko_i, int *ko_j)`

  Return true if the move `(i,j)` by `color` is a ko capture whether capture is a legal ko capture on this move or not. If `(*ko_i,*ko_j)` are non-`NULL`, then the location of the captured ko stone are returned through `(*ko_i,*ko_j)`. If the move is not a ko capture, `(*ko_i,*ko_j)` is set to `(-1, -1)`.

- `int is_illegal_ko_capture(int i, int j, int color)`

  Return true if the move `(i,j)` by `color` would be an illegal ko capture. There is no need to call both `is_ko` and `is_illegal_ko_capture`.
- `int is_self_atari(int i, int j, int color)`

  Return true if a move by `color` at `(i, j)` would be a self atari, i.e. whether it would get only one liberty. This function returns true also for the case of a suicide move.
- `int is_suicide(int i, int j, int color)`

  Returns true if a move at `(i,j)` is suicide for `color`.
- `int does_capture_something(int i, int j, int color)`

  Returns true if a move at `(i,j)` does capture any stone for the other side.
- `int stones_on_board(int color)`

  Return the number of stones of the indicated color(s) on the board. This only count stones in the permanent position, not stones placed by `trymove()` or `tryko()`. Use `stones_on_board(BLACK | WHITE)` to get the total number of stones on the board.

8.2.4 String and Miscellaneous Functions

These functions are used for getting information like liberties, member stones and similar about strings. Most of these are here because they have a particularly efficient implementation through access to the incremental data structures behind the scene.

- `void find_origin(int i, int j, int *origini, int *originj)`

  Find the origin of a worm or a cavity, i.e. the point with smallest '`i`' coordinate and in the case of a tie with smallest '`j`' coordinate. The idea is to have a canonical reference point for a string.
- `int findstones(int m, int n, int maxstones, int *stonei, int *stonej)`

  Find the stones of the string at `(m, n)`. `(m, n)` must not be empty. The locations of up to `maxstones` stones are written into `(stonei[], stonej[])`. The full number of stones is returned.
- `int countstones(int m, int n)`

  Count the number of stones in a string.
- `void mark_string(int i, int j, char mx[MAX_BOARD][MAX_BOARD], char mark)`

  For each stone in the string at `(i, j)`, set `mx` to value `mark`. If some of the stones in the string are marked prior to calling this function, only the connected unmarked stones starting from `(i, j)` are guaranteed to become marked. The rest of the string may or may not become marked.
- `int liberty_of_string(int ai, int aj, int si, int sj)`

  Returns true if `(ai, aj)` is a liberty of the string at `(si, sj)`.
- `int neighbor_of_string(int ai, int aj, int si, int sj)`

  Returns true if `(ai, aj)` is adjacent to the string at `(si, sj)`.

- `int same_string(int ai, int aj, int bi, int bj)`

  Returns true if `(ai, aj)` is a stone in the same string as `(bi, bj)`.
- `int findlib(int m, int n, int maxlib, int *libi, int *libj)`

  Find the liberties of the string at `(m, n)`, which must not be empty. The locations of up to maxlib liberties are written into `(libi[], libj[])`. The full number of liberties is returned. If you want the locations of all liberties, whatever their number, you should pass `MAXLIBS` as the value for maxlib and allocate space for `libi[]`, `libj[]` accordingly.
- `int countlib(int m, int n)`

  Count the number of liberties of the string at `(m,n)`, which must not be empty.
- `int approxlib(int m, int n, int maxlib, int *libi, int *libj)`

  Find the liberties a stone of the given color would get if played at `(m, n)`, ignoring possible captures of opponent stones. `(m, n)` must be empty. If `libi!=NULL`, the locations of up to maxlib liberties are written into `(libi[], libj[])`. The counting of liberties may or may not be halted when maxlib is reached. The number of liberties found is returned. If you want the number or the locations of all liberties, however many they are, you should pass `MAXLIBS` as the value for maxlib and allocate space for `libi[]`, `libj[]` accordingly.
- `int chainlinks(int m, int n, int adji[MAXCHAIN], int adjj[MAXCHAIN]):`

  Returns (in `adji[]`, `adjj[]` arrays) the chain (strings) surrounding the string at `(m, n)`. The chain is defined as the set of strings in immediate connection to the `(m, n)` string. Return value is the number of strings in the chain.
- `void chainlinks2(int m, int n, int adji[MAXCHAIN], int adjj[MAXCHAIN], int lib)`

  Returns (in `adji[]`, `adjj[]` arrays) the strings surrounding the string at `(m, n)`, which have exactly `lib` liberties.

8.2.5 Miscellaneous Functions

- `void incremental_order_moves(int mi, int mj, int color, int si, int sj, int *number_edges, int *number_same_string, int *number_own, int *number_opponent, int *captured_stones, int *threatened_stones, int *saved_stones, int *number_open)`

  Help collect the data needed by `order_moves()` in ‘`reading.c`’. It’s the caller’s responsibility to initialize the result parameters.

8.3 Hashing of Board Positions

Hashing of go positions in a hash table (sometimes also called a transposition table) is implemented in `libboard`, in ‘`hash.c`’ and `cache.c` to be exact.

To use the hash function, you must include ‘`hash.h`’ and to use the entire hash table, you must include ‘`cache.h`’ in your program. The details are described in Section 14.2 [Hashing], page 125.

9 Move generation

9.1 Introduction

GNU Go 3.0 has a move generation scheme that is substantially different from earlier versions. In particular, it is different from the method of move generation in GNU Go 2.6.

In the old scheme, various move generators suggested different moves with attached values. The highest such value then decided the move. There were two important drawbacks with this scheme:

- Efficient multipurpose moves could only be found by patterns which explicitly looked for certain combinations, such as a simultaneous connection and cut. There was also no good way to e.g. choose among several attacking moves.
- The absolute move values were increasingly becoming harder to tune with the increasing number of patterns. They were also fairly subjective and the tuning could easily break in unexpected ways when something changed, e.g. the worm valuation.

The basic idea of the new move generation scheme is that the various move generators suggest reasons for moves, e.g. that a move captures something or connects two strings, and so on. When all reasons for the different moves have been found, the valuation starts. The primary advantages are

- The move reasons are objective, in contrast to the move values in the old scheme. Anyone can verify whether a suggested move reason is correct.
- The centralized move valuation makes tuning easier. It also allows for style dependent tuning, e.g. how much to value influence compared to territory. Another possibility is to increase the value of safe moves in a winning position.

9.2 Overview

The engine of GNU Go takes a position and a color to move and generates the (supposedly) optimal move. This is done by the function genmove() in engine/genmove.c.

The move generation is done in three steps:

1. information gathering
2. generation of moves and move reasons
3. valuation of the suggested moves

This is somewhat simplified. In reality there is some overlap between the steps.

9.3 Information gathering

First we have to collect as much information as possible about the current position. Such information could be life and death of the groups, moyo status, connection of groups and so on. Information gathering are performed by the following functions, called in this order:

- `make_worms`

  Collect information about all connected sets of stones (strings) and cavities. This information is stored in the `worm[][]` array.

- `make_dragons`

  Collect information about connected strings, which are called dragons. Important information here is number of eyes, life status, and connectedness between strings. The information is stored mainly in the array `dragon[][]` but also in `dragon2[][]`.

See Section 4.3 [Examining the Position], page 21, for a more exact itinerary of the information-gathering portion of the move-generation proces.

See Chapter 10 [Worms and Dragons], page 67, for more detailed documentation about `make_worms` and `make_dragons`.

9.4 Generation of move reasons

Each move generator suggests a number of moves. It justifies each move suggestion with one or move *move reasons*. These move reasons are collected at each intersection where the moves are suggested for later valuation. The different kinds of move reasons considered by GNU Go are:

`ATTACK_MOVE`
`DEFEND_MOVE`
: Attack or defend a worm.

`ATTACK_THREAT_MOVE`
`DEFEND_THREAT_MOVE`
: Threaten to attack or defend a worm.

`NON_ATTACK_MOVE`
`NON_DEFEND_MOVE`
: a non-attacking or non-defending move.

`ATTACK_EITHER_MOVE`
: a move that attacks either on of two worms.

`DEFEND_BOTH_MOVE`
: a move that simultaneously defends two worms.

`CONNECT_MOVE`
`CUT_MOVE`
: Connect or cut two worms.

`ANTISUJI_MOVE`
: Declare an antisuji or forbidden move.

`SEMEAI_MOVE`
`SEMEAI_THREAT`
: Win or threaten to win a semeai.

`EXPAND_TERRITORY_MOVE`
`BLOCK_TERRITORY_MOVE`
: a move that expands our territory or blocks opponents expansion.

`EXPAND_MOYO_MOVE`
: a move expanding a moyo.

`VITAL_EYE_MOVE`
: a vital point for life and death.

`STRATEGIC_ATTACK_MOVE`
`STRATEGIC_DEFEND_MOVE`
: Moves added by 'a' and 'd' class patterns (see Section 12.2 [Pattern Classification], page 93) which (perhaps intangibly) attack or defend a dragon.

`OWL_ATTACK_MOVE`
`OWL_DEFEND_MOVE`
: an owl attack or defense move.

`OWL_ATTACK_THREAT`
`OWL_DEFEND_THREAT`
: a threat to owl attack or defend a group.

`OWL_PREVENT_THREAT`
: a move to remove an owl threat.

`UNCERTAIN_OWL_ATTACK`
`UNCERTAIN_OWL_DEFENSE`
: an uncertain owl attack or defense.

`MY_ATARI_ATARI_MOVE`
: a move that starts a chain of ataris, eventually leading to a capture.

`YOUR_ATARI_ATARI_MOVE`
: a move that if played by the opponent starts a chain of ataris for the opponent, leading to capture, which is also a safe move for us. Preemptively playing such a move almost always defends the threat.

NOTE: Some of these are reasons for **not** playing a move.

More detailed discussion of these move reasons will be found in the next section.

9.5 Detailed Descriptions of various Move Reasons

9.5.1 Attacking and defending moves

A move which tactically captures a worm is called an *attack move* and a move which saves a worm from being tactically captured is called a *defense move*. It is understood that a defense move can only exist if the worm can be captured, and that a worm without defense only is attacked by moves that decrease the liberty count or perform necessary backfilling.

It is important that all moves which attack or defend a certain string are found, so that the move generation can make an informed choice about how to perform a capture, or find moves which capture and/or defend several worms.

Attacking and defending moves are first found in `make_worms` while it evaluates the tactical status of all worms, although this step only gives one attack and defense (if any) move per worm. Immediately after, still in `make_worms`, all liberties of the attacked worms are tested for additional attack and defense moves. More indirect moves are found by `find_attack_patterns` and `find_defense_patterns`, which match the A (attack) and D (defense) class patterns in '`patterns/attack.db`' and '`patterns/defense.db`' As a final step, all moves which fill some purpose at all are tested whether they additionally attacks or defends some worm. (Only unstable worms are analyzed.)

9.5.2 Threats to Attack or Defend

A threat to attack a worm, but where the worm can be defended is used as a secondary move reason. This move reason can enhance the value of a move so that it becomes sente. A threatening move without any other justification can also be used as a ko threat. The same is true for a move that threatens defense of a worm, but where the worm can still be captured if the attacker doesn't tenuki.

Threats found by the owl code are called **owl threats** and they have their own owl reasons.

9.5.3 Not working attack and defense moves

The tactical reading may come up with ineffective attacks or defenses occasionally. When these can be detected by patterns, it's possible to cancel the attack and/or defense potential of the moves by using these move reasons. This can only be done by action lines in the patterns.

9.5.4 Multiple attack or defense moves

Sometimes a move attacks at least one of a number of worms or simultaneously defends all of several worms. These moves are noted by their own move reasons.

9.5.5 Cutting and connecting moves

Moves which connect two distinct dragons are called `connecting moves`. Moves which prevent such connections are called *cutting moves*. Cutting and connecting moves are primarily found by pattern matching, the `C` and `B` class patterns.

A second source of cutting and connecting moves comes from the attack and defense of cutting stones. A move which attacks a worm automatically counts as a connecting move if there are multiple dragons adjacent to the attacked worm. Similarly a defending move counts as a cutting move. The action taken when a pattern of this type is found is to induce a connect or cut move reason.

When a cut or connect move reason is registered, the involved dragons are of course stored. Thus the same move may cut and/or connect several pairs of dragons.

9.5.6 Semeai winning moves

A move which is necessary to win a capturing race is called a *semeai move*. These are similar to attacking moves, except that they involve the simultaneous attack of one worm and the defense of another. As for attack and defense moves, it's important that all moves which win a semeai are found, so an informed choice can be made between them.

Semeai move reasons should be set by the semeai module. However this has not been implemented yet. One might also wish to list moves which increase the lead in a semeai race (removes ko threats) for use as secondary move reasons. Analogously if we are behind in the race.

9.5.7 Making or destroying eyes

A move which makes a difference in the number of eyes produced from an eye space is called an *eye move*. It's not necessary that the eye is critical for the life and death of the dragon in question, although it will be valued substantially higher if this is the case. As usual it's important to find all moves that change the eye count.

(This is part of what eye_finder was doing. Currently it only finds one vital point for each unstable eye space.)

9.5.8 Antisuji moves

Moves which are locally inferior or for some other reason must not be played are called *antisuji moves*. These moves are generated by pattern matching. Care must be taken with this move reason as the move under no circumstances will be played.

9.5.9 Territorial moves

Any move that increases territory gets a move reason. These are the block territory and expand territory move reasons. Such move reasons are added by the '`b`' and '`e`' patterns in '`patterns/patterns.db`'. Similarly the '`E`' patterns attempt to generate or mitigate an moyo, which is a region of influence not yet secure territory, yet valuable. Such a pattern sets the "expand moyo" move reason.

9.5.10 Attacking and Defending Dragons

Just as the tactical reading code tries to determine when a worm can be attacked or defended, the owl code tries to determine when a dragon can get two eyes and live. The function `owl_reasons()` generates the corresponding move reasons.

The owl attack and owl defense move reasons are self explanatory.

The owl attack threat reason is generated if owl attack on an opponent's dragon fails but the owl code determines that the dragon can be killed with two consecutive moves. The killing moves are stored in `(dragon[ai][aj].owl_attacki_i, dragon[ai][aj].owl_attacki_j)` and `(dragon[ai][aj].owl_second_attacki_i, dragon[ai][aj].owl_second_attacki_j)`.

Similarly if a friendly dragon is dead but two moves can revive it, an owl defense threat move reason is generated.

The prevent threat reasons are similar but with the colors reversed: if the opponent has an attack threat move then a move which removes the threat gets a prevent threat move reason.

The owl uncertain move reasons are generated when the owl code runs out of nodes. In order to prevent the owl code from running too long, a cap is put on the number of nodes one owl read can generate. If this is exceeded, the reading is cut short and the result is cached as usual, but marked uncertain. In this case an owl uncertain move reason may be generated. For example, if the owl code finds the dragon alive but is unsure, a move to defend may still be generated.

9.5.11 Combination Attacks

The function `atari_atari` tries to find a sequence of ataris culminating in an unexpected change of status of any opponent string, from ALIVE to CRITICAL, or from CRITICAL to DEAD. Once such a sequence of ataris is found, it tries to shorten it by rejecting irrelevant moves.

9.6 Valuation of suggested moves

Moves are valued with respect to five different criteria. These are

- territorial value
- influence value
- strategical value
- shape value,
- secondary value.

All of these are floats and should be measured in terms of actual points.

Territorial value is the amount of secure territory generated (or saved) by the move. Attack and defense moves have territorial values given by twice the number of stones in the worm plus adjacent empty space. This value is in practice approximated from the "effective size" measure.

Influence value is an estimation of the move's effect on the size of potential territory and possibly "area". This is currently implemented by using delta_moyo_simple(). This can probably be improved quite a bit. If the move captures some stones, this fact should be taken into account when computing moyo/area.

Strategical value is a measure of the effect the move has on the safety of all groups on the board. Typically cutting and connecting moves have their main value here. Also edge extensions, enclosing moves and moves towards the center have high strategical value. The strategical value should be the sum of a fraction of the territorial value of the involved dragons. The fraction is determined by the change in safety of the dragon.

Shape value is a purely local shape analysis, which primarily is intended to choose between moves having the same set of reasons. An important role of this measure is to offset mistakes made by the estimation of territorial and influence values. In open positions it's often worth sacrificing a few points of (apparent) immediate profit to make good shape. Shape value is implemented by pattern matching, the Shape patterns.

Secondary value is given for move reasons which by themselves are not sufficient to play the move. One example is to reduce the number of eyes for a dragon that has several or to attack a defenseless worm.

When all these values have been computed, they are summed, possibly weighted (secondary value should definitely have a small weight), into a final move value. This value is used to decide the move.

9.6.1 Territorial Value

The algorithm for computing territorial value is in the function `estimate_territorial_value`. As the name suggests, it seeks to estimate the amount the move adds to secure territory.

This function examines every reason for the move and takes into account the safety of different dragons. For example if the reason for the move is that it attacks and kills a worm, no value is assigned if the worm is already DEAD. If the worm is not DEAD the value of the move is twice the effective size of the worm.

In addition to such additions to territory, if the move is found to be a block or expanding move, the function `influence_delta_territory` is consulted to find areas where after the move the influence function becomes so strong that these are counted as secure territory, or where the influence function is sufficiently weakened that these are removed from the secure territory of the opponent (see Section 16.6 [Influential Functions], page 149).

9.6.2 Influence Value

The function `estimate_influence_value` attempts to assign a value to the influence a move. The functions `influence_delta_strict_moyo` `influence_delta_strict_area` are called to find areas where after the move the influence function becomes strong enough that these are counted as friendly moyo or area, or which are taken away from the opponent's moyo or area (see Section 16.6 [Influential Functions], page 149).

9.6.3 Strategical Value

Strategical defense or attack reasons are assigned to any move which matches a pattern of type '`a`' or '`d`'. These are moves which in some (often intangible) way tend to help strengthen or weaken a dragon. Of course strengthening a dragon which is already alive should not be given much value, but when the move reason is generated it is not necessary to check its status or safety. This is done later, during the valuation phase.

9.6.4 Shape Factor

In the value field of a pattern (see Section 12.3 [Pattern Values], page 95) one may specify a shape value.

This is used to compute the shape factor, which multiplies the score of a move. We take the largest positive contribution to shape and add 1 for each additional positive contribution found. Then we take the largest negative contribution to shape, and add 1 for each additional negative contribution. The resulting number is raised to the power 1.05 to obtain the shape factor.

The rationale behind this complicated scheme is that every shape point is very significant. If two shape contributions with values (say) 5 and 3 are found, the second contribution should be devalued to 1. Otherwise the engine is too difficult to tune since finding multiple contributions to shape can cause significant overvaluing of a move.

9.6.5 Minimum Value

A pattern may assign a minimum (and sometimes also a maximum) value. For example the Joseki patterns have values which are prescribed in this way, or ones with a `value` field. One prefers not to use this approach but in practice it is sometimes needed.

9.6.6 Secondary Value

Secondary move reasons are weighed very slightly. Such a move can tip the scales if all other factors are equal.

Followup value refers to value which may acrue if we get two moves in a row in a local area. It is assigned by the function `add_followup_value`, for example through the `followup_value` autohelper macro.

Attack and defense threats, including owl threats are usually given a small amount of weight, as is followup value.

If the largest move on the board is a ko which we cannot legally take, then such a move becomes attractive as a ko threat and the followup value or the value of the threat are taken in full.

9.7 Move Generation Functions

The following functions are defined in '`move_reasons.c`'.

- `void clear_move_reasons(void)`

  Initialize move reason data structures.
- `void add_lunch(int ai, int aj, int bi, int bj)`

  See if a lunch is already in the list of lunches, otherwise add a new entry. A lunch is in this context a pair of eater (a dragon) and food (a worm).
- `void remove_lunch(int ai, int aj, int bi, int bj)`

  Remove a lunch from the list of lunches.
- `void add_defense_move(int ti, int tj, int ai, int aj)`

  Add to the reasons for the move at (ti, tj) that it defends the worm at (ai, aj).
- `int defense_move_known(int ti, int tj, int ai, int aj)`

  Query whether a defense move is already known. Add to the reasons for the move at (ti, tj) that it attacks the worm at (ai, aj).
- `int attack_move_known(int ti, int tj, int ai, int aj)`

  Query whether an attack move is already known.
- `void remove_defense_move(int ti, int tj, int ai, int aj)`

  Remove from the reasons for the move at (ti, tj) that it defends the worm at (ai, aj). We do this by adding a NON_DEFEND move reason and wait until later to actually remove it. Otherwise it may be added again. We must also check that there do exist a defense move reason for this worm. Otherwise we may end up in an infinite loop when trying to actually remove it.
- `void remove_attack_move(int ti, int tj, int ai, int aj)`

  Remove from the reasons for the move at (ti, tj) that it attacks the worm at (ai, aj). We do this by adding a NON_ATTACK move reason and wait until later to actually remove it. Otherwise it may be added again.

- `void add_connection_move(int ti, int tj, int ai, int aj, int bi, int bj)`

  Add to the reasons for the move at (ti, tj) that it connects the dragons at (ai, aj) and (bi, bj). Require that the dragons are distinct.
- `void add_cut_move(int ti, int tj, int ai, int aj, int bi, int bj)`

  Add to the reasons for the move at (ti, tj) that it cuts the dragons at (ai, aj) and (bi, bj). Require that the dragons are distinct.
- `void add_antisuji_move(int ti, int tj)`

  Add to the reasons for the move at (ti, tj) that it is an anti-suji. This means that it's a locally inferior move or for some other reason must **not** be played.
- `void add_semeai_move(int ti, int tj, int ai, int aj, int bi, int bj)`

  Add to the reasons for the move at (ti, tj) that it wins a semeai between my worm at (ai, aj) and your worm at (bi, bj).
- `void add_vital_eye_move(int ti, int tj, int ai, int aj, int color)`

  Add to the reasons for the move at (ti, tj) that it's the vital point for the eye space at (ai, aj) of color color.
- `void add_attack_either_move(int ti, int tj, int ai, int aj, int bi, int bj)`

  Add to the reasons for the move at (ti, tj) that it attacks either (ai, aj) or (bi, bj) (e.g. a double atari). This move reason is only used for double attacks on opponent stones. Before accepting the move reason, check that the worms are distinct and that neither is undefendable.
- `void add_defend_both_move(int ti, int tj, int ai, int aj, int bi, int bj)`

  Add to the reasons for the move at (ti, tj) that it defends both (ai, aj) and (bi, bj) (e.g. from a double atari). This move reason is only used for defense of own stones.
- `void add_block_territory_move(int ti, int tj)`

  Add to the reasons for the move at (ti, tj) that it secures territory by blocking.
- `void add_expand_territory_move(int ti, int tj)`

  Add to the reasons for the move at (ti, tj) that it expands territory.
- `void add_expand_moyo_move(int ti, int tj)`

  Add to the reasons for the move at (ti, tj) that it expands moyo.
- `void add_shape_value(int ti, int tj, float value)`

  Increase or decrease the shape value for the move at (ti, tj).
- `void add_strategical_attack_move(int ti, int tj, int ai, int aj)`

  Add to the reasons for the move at (ti, tj) that it attacks the dragon (ai, aj) on a strategical level.
- `void add_strategical_defense_move(int ti, int tj, int ai, int aj)`

  Add to the reasons for the move at (ti, tj) that it defends the dragon (ai, aj) on a strategical level.
- `void add_followup_value(int ti, int tj, float value)`

  Add value of followup moves.

- `void set_minimum_move_value(int ti, int tj, float value)`
  Set a minimum allowed value for the move.
- `void set_maximum_move_value(int ti, int tj, float value)`
  Set a maximum allowed value for the move.
- `void set_minimum_territorial_value(int ti, int tj, float value)`
  Set a minimum allowed territorial value for the move.
- `void set_maximum_territorial_value(int ti, int tj, float value)`
  Set a maximum allowed territorial value for the move.
- `int review_move_reasons(int *i, int *j, float *val, int color)`
  Review the move reasons to find which (if any) move we want to play.

9.8 Local Move Generation Functions

The following functions in '`move_reasons.c`' are declared static. Their scope is limited to that file.

- `static int find_worm(int ai, int aj)`
  Find the index of a worm in the list of worms. If necessary, add a new entry. (ai, aj) must point to the origin of the worm.
- `static int find_dragon(int ai, int aj)`
  Find the index of a dragon in the list of dragons. If necessary, add a new entry. (ai, aj) must point to the origin of the dragon.
- `static int find_connection(int dragon1, int dragon2)`
  Find the index of a connection in the list of connections. If necessary, add a new entry.
- `static int find_semeai(int myworm, int yourworm)`
  Find the index of a semeai in the list of semeais. If necessary, add a new entry.
- `static int find_worm_pair(int worm1, int worm2)`
  Find the index of an unordered pair of worms in the list of worm pairs. If necessary, add a new entry.
- `static int find_eye(int i, int j, int color)`
  Find the index of an eye space in the list of eye spaces. If necessary, add a new entry.
- `static int find_reason(int type, int what)`
  Find a reason in the list of reasons. If necessary, add a new entry.
- `static void add_move_reason(int ti, int tj, int type, int what)`
  Add a move reason for (ti, tj) if it's not already there or the table is full.
- `static void remove_move_reason(int ti, int tj, int type, int what)`
  Remove a move reason for (ti, tj). Ignore silently if the reason wasn't there.
- `static int move_reason_known(int ti, int tj, int type, int what)`
  Check whether a move reason already is recorded for a move.

- `static void find_more_attack_and_defense_moves(int color)`

  Test all moves that defends, attacks, connects or cuts to see if they also attack or defend some other worm.
- `static void remove_opponent_attack_and_defense_moves(int color)`

  Remove attacks on own stones and defense of opponent stones, i.e. moves which are only relevant for the opponent. It might seem useful to take these into account (proverb "my enemy's vital point is my vital point") but now it seems they only lead to trouble. It's easiest just to remove them altogether.
- `static void do_remove_false_attack_and_defense_moves(void)`

  Remove attacks and defenses that have earlier been marked as NON_ATTACK or NON_DEFEND respectively, because they actually don't work.
- `static int strategically_sound_defense(int ai, int aj, int ti, int tj)`

  It's often bad to run away with a worm that is in a strategically weak position. This function gives heuristics for determining whether a move at (ti, tj) to defend the worm (ai, aj) is strategically sound. These heuristics need improvement. The biggest weakness is that they sometimes fail to detect when we're running away towards open ground. It would help much to have a reliable escape route mechanism.
- `static void induce_secondary_move_reasons(int color)`

  Any move that captures or defends a worm also connects or cuts the surrounding dragons. Find these secondary move reasons.
  - There is a certain amount of optimizations that could be done here.
  - Even when we defend a worm, it's possible that the opponent still can secure a connection, e.g. underneath a string with few liberties. Thus a defense move isn't necessarily a cut move.
  - Connections are transitive. If a move connects A with B and B with C, we should infer that it connects A with C as well.
- `static float effective_dragon_size(int ai, int aj)`

  Measure the "effective" size of a dragon. This measure is a reasonable approximation of how much area a dragon cover, including some amount of surrounding empty spaces.
- `static float dragon_safety(int ai, int aj, int ignore_dead_dragons)`

  An attempt to estimate the safety of a dragon. This should be possible to improve considerably. The resulting value is interpreted so that 1.0 means a fully safe dragon while 0.0 is an almost dead dragon.
- `static float connection_value2(int ai, int aj, int bi, int bj, int ti, int tj)`

  Strategical value of connecting (or cutting) the dragon at (ai, aj) to the dragon at (bi, bj). This function is assymetric.
- `static void estimate_territorial_value(int m, int n, int color)`

  Estimate the direct territorial value of a move at (m,n).
- `static void estimate_influence_value(int m, int n, int color)`

  Estimate the influence value of a move at (m,n).

- `static void estimate_strategical_value(int m, int n, int color)`

  Estimate the strategical value of a move at (m,n).
- `static int is_antisuji_move(int m, int n)`

  Look through the move reasons to see whether (m, n) is an antisuji move.
- `static float value_move_reasons(int m, int n, int color)`

  Combine the reasons for a move at (m, n) into an old style value. These heuristics are now somewhat less ad hoc but probably still need a lot of improvement.
- `static void value_moves(int color)`

  Loop over all possible moves and value the move reasons for each.

9.9 End Game

Endgame moves are generated just like any other move by GNU Go. In fact, the concept of endgame does not exist explicitly, but if the largest move initially found is worth 6 points or less, an extra set of patterns in '`endgame.db`' is matched and the move valuation is redone.

10 Worms and Dragons

Before considering its move, GNU Go collects some data in several arrays. Two of these arrays, called `worm` and `dragon`, are discussed in this document. Others are discussed in See Chapter 11 [Eyes], page 82.

This information is intended to help evaluate the connectedness, eye shape, escape potential and life status of each group.

Later routines called by `genmove()` will then have access to this information. This document attempts to explain the philosophy and algorithms of this preliminary analysis, which is carried out by the two routines `make_worm()` and `make_dragon()` in '`dragon.c`'.

A *worm* is a maximal set of vertices on the board which are connected along the horizontal and vertical lines, and are of the same color, which can be `BLACK`, `WHITE` or `EMPTY`. The term `EMPTY` applied to a worm means that the worm consists of empty (unoccupied) vertices. It does **not** mean that that the worm is the empty set. A *string* is a nonempty worm. An empty worm is called a *cavity*. If a subset of vertices is contained in a worm, there is a unique worm containing it; this is its *worm closure*.

A *dragon* is a union of strings of the same color which will be treated as a unit. The dragons are generated anew at each move. If two strings are in the dragon, it is the computer's working hypothesis that they will live or die together and are effectively connected.

The purpose of the dragon code is to allow the computer to formulate meaningful statements about life and death. To give one example, consider the following situation:

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

The X's here should be considered a single group with one three-space eye, but they consist of two separate strings. Thus we must amalgamate these two strings into a single dragon. Then the assertion makes sense, that playing at the center will kill or save the dragon, and is a vital point for both players. It would be difficult to formulate this statement if the X's are not perceived as a unit.

The present implementation of the dragon code involves simplifying assumptions which can be refined in later implementations.

10.1 Worms

The array `struct worm_data worm[MAX_BOARD][MAX_BOARD]` collects information about the worms. We will give definitions of the various fields. Each field has constant value at each vertex of the worm. We will define each field.

```
struct worm_data {
  int color;
  int size;
```

```
    float effective_size;
    int origini;
    int originj;
    int liberties;
    int liberties2;
    int liberties3;
    int liberties4;
    int attacki;
    int attackj;
    int attack_code;
    int defendi;
    int defendj;
    int defend_code;
    int lunchi;
    int lunchj;
    int cutstone;
    int cutstone2;
    int genus;
    int value;
    int ko;
    int inessential;
    int invincible;
    int unconditional_status;
  };
```

- `color`

  If the worm is `BLACK` or `WHITE`, that is its color. Cavities (empty worms) have an additional attribute which we call *bordercolor*. This will be one of `BLACK_BORDER`, `WHITE_BORDER` or `GRAY_BORDER`. Specifically, if all the worms adjacent to a given empty worm have the same color (black or white) then we define that to be the bordercolor. Otherwise the bordercolor is gray.

  Rather than define a new field, we keep this data in the field color. Thus for every worm, the color field will have one of the following values: `BLACK`, `WHITE`, `GRAY_BORDER`, `BLACK_BORDER` or `WHITE_BORDER`. The last three categories are empty worms classified by bordercolor.

- `size`

  This field contains the cardinality of the worm.

- `effective_size`

  This is the number of stones in a worm plus the number of empty intersections that are at least as close to this worm as to any other worm. Intersections that are shared are counted with equal fractional values for each worm. This measures the direct territorial value of capturing a worm. *effective_size* is a floating point number. Only intersections at a distance of 4 or less are counted.

- `(origini, originj)`

  Each worm has a distinguished member, called its *origin*. Its coordinates are `(origini, originj)`. The purpose of this field is to make it easy to determine when two vertices lie in the same worm: we compare their origin. Also if we wish to perform some test once for each worm, we simply perform it at the origin and ignore the other vertices. The origin is characterized by the test:

```
(worm[m][n].origini == m) && (worm[m][n].originj == n).
```

- `liberties`

  For a nonempty worm the field liberties is the number of liberties of the string. This is supplemented by `LIBERTIES2`, `LIBERTIES3` and `LIBERTIES4`, which are the number of second order, third order, and fourth order liberties, respectively. The definition of liberties of order >1 is adapted to the problem of detecting the shape of the surrounding cavity. In particular we want to be able to see if a group is loosely surrounded. *a liberty of order n* is an empty vertex which may be connected to the string by placing n stones of the same color on the board, but no fewer. The path of connection may pass through an intervening group of the same color. The stones placed at distance >1 may not touch a group of the opposite color. Connections through ko are not permitted. Thus in the following configuration:

```
.XX...    We label the     .XX.4.
XO....    liberties of     XO1234
XO....    order < 5 of     XO1234
......    the O group:     .12.4.
.X.X..                     .X.X..
```

  The convention that liberties of order >1 may not touch a group of the opposite color means that knight's moves and one space jumps are perceived as impenetrable barriers. This is useful in determining when the string is becoming surrounded.

  The path may also not pass through a liberty at distance 1 if that liberty is flanked by two stones of the opposing color. This reflects the fact that the O stone is blocked from expansion to the left by the two X stones in the following situation:

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

  We say that n is the *distance* of the liberty of order n from the dragon.

- `(attacki, attackj)`:

  If it is determined that the string may be easily captured, `(attacki, attackj)` points to an attacking move. This is only used for strings with <5 liberties. If no attacking move is found, then `attack_code == 0`.

- `attack_code`

  1 if the worm can be captured unconditionally, 2 or 3 if it can be captured with ko. If it can be captured provided the attacker is willing to ignore any ko threat, then the `attack_code == 2`. If it can be captured provided the

attacker can come up with a sufficiently large ko threat, then the `attack_code == 3`.

- `lunch`

  If `lunchi != -1` then `(lunchi, lunchj)` points to a boundary worm which can be easily captured. (It does not matter whether or not the string can be defended.)

- `defend`:

  If there is an attack on the string (stored in the `attack` field defined above), and there is a move which defends the string, this move is stored in `(defendi, defendj)`. Otherwise `defend_code == 0`.

- `defend_code`

  1 if the worm can be defended unconditionally, 2 or 3 if it can be defended with ko. If it can be defended provided the defender is willing to ignore any ko threat, then the `defend_code == 2`. If it can be captured provided the defender can come up with a sufficiently large ko threat, then the `defend_code == 3`. If there is no attack, `defend_code` is 0.

We have two distinct notions of cutting stone, which we keep track of in the separate fields `worm.cutstone` and `worm.cutstone2`. We maintain both fields because the historically older cutstone field is needed to deal with the fact that e.g. in the position

```
OXX.O
.OOXO
OXX.O
```

the X stones are amalgamated into one dragon because neither cut works as long as the two O stones are in atari. Therefore we add one to the cutstone field for each potential cutting point, indicating that these O stones are indeed worth rescuing.

For the time being we use both concepts in parallel. It's possible we also old concept for correct handling of lunches.

`cutstone`:

  This field is equal to 2 for cutting stones, 1 for potential cutting stones. Otherwise it is zero. Definitions for this field: a *cutting stone* is one adjacent to two enemy strings, which do not have a liberty in common. The most common type of cutting string is in this situation:

```
XO
OX
```

  A *potential cutting stone* is adjacent to two enemy strings which do share a liberty. For example, X in:

```
XO
O.
```

  For cutting strings we set `worm[m][n].cutstone=2`. For potential cutting strings we set `worm[m][n].cutstone=1`.

`cutstone2`:

Cutting points are identified by the patterns in the connections database. Proper cuts are handled by the fact that attacking and defending moves also count as moves cutting or connecting the surrounding dragons. The cutstone2 field is set during find_cuts(), called from make_domains().

The `cutstone2` field is needed to deal with the fact that e.g. in the position

```
OXX.O
.OOXO
OXX.O
```

the X stones are amalgamated into one dragon because neither cut works as long as the two O stones are in atari. Therefore we add one to the cutstone field for each potential cutting point, indicating that these O stones are indeed worth rescuing.

For the time being we use both concepts in parallel, with the new concept stored in `cutstone2`. It's possible that we have to keep the old concept for correct handling of lunches.

`genus`:

There are two separate notions of *genus* for worms and dragons. The dragon notion is more important, so `dragon[m][n].genus` is a far more useful field than `worm[m][n].genus`. Both fields are intended as approximations to the number of eyes. The *genus* of a string is the number of connected components of its complement, minus one. It is an approximation to the number of eyes of the string.

`ko`:

For every ko, the flag `ko` is set to 1 at the ko stone which is in atari, and also at the ko cavity adjacent to it. Thus in this situation:

```
 XO
X.XO
 XO
```

the flag `ko` is set to 1 at the rightmost X stone, and also at the cavity to its left.

`inessential`:

An *inessential* string is one which meets a criterion designed to guarantee that it has no life potential unless a particular surrounding string of the opposite color can be killed. More precisely an *inessential string* is a string S of genus zero, not adjacent to any opponent string which can be easily captured, and which has no edge liberties or second order liberties, and which satisfies the following further property: If the string is removed from the board, then the empty worm E which is the worm closure of the set of vertices which it occupied has bordercolor the opposite of the removed string. The empty worm E (empty, that is, as a worm of the board modified by removal of S) consists of the union of support of S together with certain other empty worms which we call the *boundary components* of S.

The inessential strings are used in the amalgamation of cavities in `make_dragon()`.

`invincible`:

An *invincible* worm is one which GNU Go thinks cannot be captured. Invincible worms are computed by the function `unconditional_life()` which tries to find those worms of the given color that can never be captured, even if the opponent is allowed an arbitrary number of consecutive moves.

unconditional_status

Unconditional status is also set by the function `unconditional_life`. This is set ALIVE for stones which are invincible. Stones which can not be turned invincible even if the defender is allowed an arbitrary number of consecutive moves are given an unconditional status of DEAD. Empty points where the opponent cannot form an invincible worm are called unconditional territory. The unconditional status is set to WHITE_BORDER or BLACK_BORDER depending on who owns the territory. Finally, if a stone can be captured but is adjacent to unconditional territory of its own color, it is also given the unconditional status ALIVE. In all other cases the unconditional status is UNKNOWN.

To make sense of these definitions it is important to notice that any stone which is alive in the ordinary sense (even if only in seki) can be transformed into an invincible group by some number of consecutive moves. Well, this is not entirely true because there is a rare class of seki groups not satisfying this condition. Exactly which these are is left as an exercise for the reader. Currently `unconditional_life`, which strictly follows the definitions above, calls such seki groups unconditionally dead, which of course is a misfeature. It is possible to avoid this problem by making the algorithm slightly more complex, but this is left for a later revision.

The function `makeworms()` will generate data for all worms. For empty worms, the following fields are significant: `color`, `size`, `origini` and `originj`. The `liberty`, `attack`, `defend`, `cutstone`, `genus` and `inessential` fields have significance only for nonempty worms.

10.2 Amalgamation

A dragon, we have said, is a group of stones which are treated as a unit. It is a working hypothesis that these stones will live or die together. Thus the program will not expect to disconnect an opponent's strings if they have been amalgamated into a single dragon.

The function `make_dragons()` will amalgamate worms into dragons by maintaining separate arrays `worm[]` and `dragon[]` containing similar data. Each dragon is a union of worms. Just as the data maintained in `worm[][]` is constant on each worm, the data in `dragon[][]` is constant on each dragon.

Amalgamation of two worms means means in practice replacing the origin of one worm by the origin of the other. Amalgamation takes place in two stages: first, the amalgamation of empty worms (cavities) into empty dragons (caves); then, the amalgamation of colored worms into dragons.

10.3 Amalgamation of cavities

As we have already defined it, a cavity is an empty worm. A cave is an empty dragon.

Under certain circumstances we want to amalgamate two or more cavities into a single cave. This is done before we amalgamate strings. An example where we wish to amalgamate two empty strings is the following:

```
 OOOOO
OOXXXOO
OXaObXO
OOXXXOO
 OOOOO
```

The two empty worms at a and b are to be amalgamated.

We have already defined a string to be *inessential* if it meets a criterion designed to guarantee that it has no life potential unless a particular surrounding string of the opposite color can be killed. An *inessential string* is a string S of genus zero which is not a cutting string or potential cutting string, and which has no edge liberties or second order liberties (the last condition should be relaxed), and which satisfies the following further property: If the string is removed from the board, then the empty worm E which is the worm closure of the set of vertices which it occupied has bordercolor the opposite of the removed string.

Thus in the previous example, after removing the inessential string at the center the worm closure of the center vertex consists of an empty worm of size 3 including a and b. The latter are the boundary components.

The last condition in the definition of inessential worms excludes examples such as this:

```
  OOOO
 OXXOO
OXX.XO
OX.XXO
OOXXO
 OOO
```

Neither of the two X strings should be considered inessential (together they form a live group!) and indeed after removing one of them the resulting space has gray bordercolor, so by this definition these worms are not inessential.

Some strings which should by rights be considered inessential will be missed by this criterion.

The algorithm for amalgamation of empty worms consists of amalgamating the boundary components of any inessential worm. The resulting dragon has bordercolor the opposite of the removed string.

Any dragon consisting of a single cavity has bordercolor equal to that of the cavity.

10.4 Amalgamation of strings

Amalgamation of nonempty worms in GNU Go 3.0 proceeds as follows. First we amalgamate all boundary components of an eyeshape. Thus in the following example:

```
.OOOO.       The four X strings are amalgamated into a
OOXXO.       single dragon because they are the boundary
OX..XO       components of a blackbordered cave. The
OX..XO       cave could contain an inessential string
OOXXO.       with no effect on this amalgamation.
XXX...
```

The code for this type of amalgamation is in the routine `dragon_eye()`, discussed further in EYES.

Next, we amalgamate strings which seem uncuttable. We amalgamate dragons which either share two or more common liberties, or share one liberty into the which the opponent cannot play without being captured. (ignores ko rule).

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

A database of connection patterns may be found in '`patterns/conn.db`'.

10.5 Connection

The fields `black_eye.cut` and `white_eye.cut` are set where the opponent can cut, and this is done by the B (break) class patterns in `conn.db`. There are two important uses for this field, which can be accessed by the autohelper functions `xcut()` and `ocut()`. The first use is to stop amalgamation in positions like

```
..X..
OO*OO
X.O.X
..O..
```

where X can play at * to cut off either branch. What happens here is that first connection pattern CB1 finds the double cut and marks * as a cutting point. Later the C (connection) class patterns in conn.db are searched to find secure connections over which to amalgamate dragons. Normally a diagonal connection would be deemed secure and amalgamated by connection pattern CC101, but there is a constraint requiring that neither of the empty intersections is a cutting point.

A weakness with this scheme is that X can only cut one connection, not both, so we should be allowed to amalgamate over one of the connections. This is performed by connection pattern CC401, which with the help of `amalgamate_most_valuable_helper()` decides which connection to prefer.

The other use is to simplify making alternative connection patterns to the solid connection. Positions where the diag_miai helper thinks a connection is necessary are marked as cutting points by connection pattern 12. Thus we can write a connection pattern like `CC6`:

```
?xxx?     straight extension to connect
XOO*?
O...?

:8,C,NULL

?xxx?
XOOb?
Oa..?

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

where we verify that a move at `*` would stop the enemy from safely playing at the cutting point, thus defending against the cut.

10.6 Half Eyes and False Eyes

A *half eye* is a place where, if the defender plays first, an eye will materialize, but where if the attacker plays first, no eye will materialize. A *false eye* is a vertex which is surrounded by a dragon yet is not an eye. Here is a half eye:

```
XXXXX
OO..X
O.O.X
OOXXX
```

Here is a false eye:

```
XXXXX
XOO.X
O.O.X
OOXXX
```

The "topological" algorithm for determining half and false eyes is described elsewhere (see Section 11.7 [Eye Topology], page 87).

The half eye data is collected in the dragon array. Before this is done, however, an auxiliary array called half_eye_data is filled with information. The field `type` is 0, or else `HALF_EYE` or `FALSE_EYE` depending on which type is found; and `(ki, kj)` points to a move to kill the half eye.

```
struct half_eye_data half_eye[MAX_BOARD][MAX_BOARD];

struct half_eye_data {
  int type;         /* HALF_EYE or FALSE_EYE; */
  int num_attacks;  /* number of attacking points */
  int num_defends;  /* number of defending points */
  int ai[4];        /* (ai, aj) attacks a topological halfeye */
  int aj[4];
  int di[4];        /* (di, dj) defends a topological halfeye */
  int dj[4];
};
```

The array `struct half_eye_data half_eye[MAX_BOARD][MAX_BOARD]` contains information about half and false eyes. If the type is `HALF_EYE` then up to four moves are recorded which can either attack or defend the eye. In rare cases the attack points could be different from the defense points.

10.7 Dragons

The array `struct dragon_data dragon[MAX_BOARD][MAX_BOARD]` collects information about the dragons. We will give definitions of the various fields. Each field has constant value at each vertex of the dragon.

```
struct dragon_data {
  int color;
  int id;
  int origini;
  int originj;
  int borderi;
  int borderj;
  int size;
  float effective_size;
  int heyes;
  int heyei;
  int heyej;
  int genus;
  int escape_route;
  int lunchi;
  int lunchj;
  int status;
  int owl_status;
  int owl_attacki;
  int owl_attackj;
  int owl_attack_certain;
  int owl_second_attacki;
  int owl_second_attackj;
  int owl_defendi;
```

```
    int owl_defendj;
    int owl_defend_certain;
    int owl_second_defendi;
    int owl_second_defendj;
    int old_safety;
    int matcher_status;
    int semeai;
    int semeai_margin_of_safety;
  };
```

Here are the definitions of each field.

- `color`:

  For strings, this is `BLACK` or `WHITE`. For caves, it is `BLACK_BORDER`, `WHITE_BORDER` or `GRAY_BORDER`. The meaning of these concepts is the same as for worms.

- `id`:

  This is a pointer to the dragon's field in the `dragon2` array (see Section 10.10 [Dragon2], page 80).

- `(origini, originj)`

  The origin of the dragon is a unique particular vertex of the dragon, useful for determining when two vertices belong to the same dragon. Before amalgamation the worm origins are copied to the dragon origins. Amalgamation of two dragons amounts to changing the origin of one.

- `(borderi, borderj)`

  This field is relevant for caves. If the color of the cave is `BLACK_BORDER` or `WHITE_BORDER` then the surrounding worms all have the same color `BLACK` or `WHITE` and these have been amalgamated into a dragon with origin `(borderi, borderj)`.

- `size`:

  This is the cardinality of the dragon.

- `effective_size`:

  The sum of the effective sizes of the constituent worms. Remembering that vertices equidistant between two or more worms are counted fractionally in `worm.effective_size`, this equals the cardinality of the dragon plus the number of empty vertices which are nearer this dragon than any other.

- `heyes`:

  This is the number of half eyes the dragon has. A *half eye* is a pattern where an eye may or may not materialize, depending on who moves first.

- `(heyi,heyj)`:

  If any half eyes are found, `(heyi,heyj)` points to a move which will create an eye.

- `genus`:

  The *genus* of a nonempty dragon consists of the number of distinct adjacent caves whose bordercolor is the color of the dragon, minus the number of false eyes found. The genus is a computable approximation to the number of eyes a dragon has.

- `escape_route`:

  This is a measure of the escape potential of the dragon. If `dragon.escape_route` is large, GNU Go believes that the dragon can escape, so finding two eyes locally becomes less urgent. Further documentation may be found else where (see Section 16.5 [Escape], page 148).

- `(lunchi, lunchj)`

  If `lunchi != -1`, then `(lunchi, lunchj)` points to a boundary worm which can be captured easily. In contrast with the worm version of this parameter, we exclude strings which cannot be saved.

- `status`:

  An attempt is made to classify the dragons as `ALIVE`, `DEAD`, `CRITICAL` or `UNKNOWN`. The `CRITICAL` classification means that the fate of the dragon depends on who moves first in the area. The exact definition is in the function `dragon_status()`. If the dragon is found to be surrounded, the status is `DEAD` if it has less than 1.5 eyes or if the reading code determines that it can be killed, `ALIVE` if it has 2 or more eyes, and `CRITICAL` if it has 1.5 eyes. A lunch generally counts as a half eye in these calculations. If it has less than 2 eyes but seems possibly able to escape, the status may be `UNKNOWN`.

- `owl_status`

  This is a classification similar to `dragon.status`, but based on the life and death reading in '`owl.c`'. The owl code (see Section 15.1 [The Owl Code], page 141) is only run on dragons with dragon.escape_route>5 and dragon2.moyo>10 (see Section 10.10 [Dragon2], page 80). If these conditions are not met, the owl status is `UNCHECKED`. If `owl_attack()` determines that the dragon cannot be attacked, it is classified as `ALIVE`. Otherwise, `owl_defend()` is run, and if it can be defended it is classified as `CRITICAL`, and if not, as `DEAD`.

- `(owl_attacki, owl_attackj)`

  If the owl code finds that the dragon can be attacked, this is the move. This may be tenuki (i.e. `(-1,-1)`) if the owl code thinks the group is dead as it stands.

- `owl_attack_certain`

  The function `owl_attack`, which is used to set `(owl_attacki, owl_attackj)`, is given an upper bound of `owl_node_limit` in the number of nodes it is allowed to generate. If this is exceeded the result is considered uncertain and this flag is set.

- `(owl_second_attack_i, owl_second_attack_j)`

  If the level is at least 8, and if a dragon is not owl attackable, the owl function `owl_threaten_attack` is asked if the dragon can be killed with two moves in a row. If two such killing moves are found, they are cached in `(owl_attacki, owl_attackj)` and `(owl_second_attack_i, owl_second_attack_j)`.

- `(owl_defendi, owl_defendj)`

  If the owl code finds that the dragon can be defended, this is the move.
- `owl_defend_certain`
- `(owl_second_defend_i, owl_second_defend_j)` Similar to `owl_attack_certain` and `(owl_second_attack_i, owl_second_attack_j)`
- `matcher_status`

  This is the status used by the pattern matcher. If `owl_status` is available (not `UNCHECKED`) this is used. Otherwise, we use the `status` field, except that we upgrade `DEAD` to

  `UNKNOWN`.
- `semeai`

  True if the dragon is part of a semeai.
- `semeai_margin_of_safety`

  Small if the semeai is close. Somewhat unreliable.

10.8 Colored Dragon Display

You can get a colored ASCII display of the board in which each dragon is assigned a different letter; and the different values of `dragon.status` values (`ALIVE`, `DEAD`, `UNKNOWN`, `CRITICAL`) have different colors. This is very handy for debugging. A second diagram shows the values of `owl.status`. If this is `UNCHECKED` the dragon is displayed in White.

Save a game in sgf format using CGoban, or using the '`-o`' option with GNU Go itself.

Open an `xterm` or `rxvt` window. You may also use the Linux console. Using the console, you may need to use "SHIFT-PAGE UP" to see the first diagram. Xterm will only work if it is compiled with color support—if you do not see the colors try `rxvt`. Make the background color black and the foreground color white.

Execute:

`gnugo -l [filename] -L [movenum] -T` to get the colored display.

The color scheme: Green = `ALIVE`; Yellow = `UNKNOWN`; Cyan = `DEAD` and Red = `CRITICAL`. Worms which have been amalgamated into the same dragon are labelled with the same letter.

Other useful colored displays may be obtained by using instead:

- the option -E to display eye spaces (see Chapter 11 [Eyes], page 82).
- the option -m 1 to display territory (see Chapter 17 [Moyo], page 152).

The colored displays are documented elsewhere (see Section 5.8 [Colored Display], page 33).

10.9 Worm and Dragon Functions

Here are the public functions in '`engine/worm.c`':

- `void make_worms(void)`

  Each worm is marked with an origin, having coordinates `(origini, originj)`. This is an arbitrarily chosen element of the worm, in practice the algorithm puts the origin at the first element when they are given the lexicographical order, though its location is irrelevant for applications. To see if two stones lie in the same worm, compare their origins.
- `void propagate_worm(int m, int n)`

  `propagate_worm()` takes the worm data at one stone and copies it to the remaining members of the worm.
- `int examine_cavity(int m, int n, int *edge, int *size, int *vertexi, int *vertexj)`

  If `(m, n)` is EMPTY, this function examines the cavity at `(m, n)`, determines its size and returns its bordercolor, which can be `BLACK_BORDER`, `WHITE_BORDER` or `GRAY_BORDER`. The edge parameter is set to the number of edge vertices in the cavity. `(vertexi[], vertexj[])` hold the vertices of the cavity. `vertexi[]` and `vertexj[]` should be dimensioned to be able to hold the whole board.

  If `(m, n)` is nonempty, it returns the same result, imagining that the string at `(m, n)` is removed. The edge parameter is set to the number of vertices where the cavity meets the edge in a point outside the removed string.

Here are the public functions in '`engine/dragon.c`':

- `void make_dragons()`

  This basic function finds all dragons and collects some basic information about them in the dragon array.
- `void show_dragons(void)`

  Print status info on all dragons. (Can be invoked from gdb)
- `void join_dragons(int ai, int aj, int bi, int bj)`

  Amalgamates the dragon at `(ai, aj)` to the dragon at `(bi, bj)`.
- `static int compute_dragon_status(int i, int j)`

  Tries to determine whether the dragon at `(i, j)` is `ALIVE`, `DEAD`, or `UNKNOWN`. The algorithm is not perfect and can give incorrect answers. The dragon is judged alive if its genus is >1. It is judged dead if the genus is <2, it has no escape route, and no adjoining string can be easily captured. Otherwise it is judged `UNKNOWN`.
- `void compute_escape_potential(void)`

  Compute the escape potential for the `escape2` field (see Section 10.7 [Dragons], page 76).

10.10 The Second Dragon Array.

In addition to `dragon[][]` there is a second complementary dragon data array `dragon2[]`. In contrast to `dragon[][]`, the information in this one is not duplicated to every intersection of the board. Instead the dragons are numbered, using the new field id in `dragon[][]`, and this number is used as index into the `dragon2[]` array. This number can of course not be assigned until all dragon amalgamations have been finished. Neither is the `dragon2[]` array initialized until this has been done.

The first thing this array contains is a list of neighbor dragons. The intention of this information is to be able to modify the perceived safety of a dragon with respect to the strength of its neighbors. The list of neighbors should be useful for other purposes too.

For the algorithm we refer to the source code and its comments, in the function `compute_supplementary_dragon_data()` in '`dragon.c`'.

To access the `dragon[][]` array given a dragon id number or the `dragon2[]` array given a board coordinate, there are the two handy macros `DRAGON(d)` and `DRAGON2(m, n)`. Also notice that the `dragon2[]` data and the id number only are valid for non-empty dragons, i.e. not for caves.

11 Eyes and Half Eyes

The purpose of this Chapter is to describe the algorithm used in GNU Go 3.0 to determine eyes. There are actually two alternative algorithms: the graph-based algorithm in `optics.c`, and the algorithm based on reading in `life.c`. The life code is slower than the graph based algorithm, but more accurate. You can make it the default by using the option `--life`. Otherwise, GNU Go will only call the life code if the graph based algorithm decides that it needs an expert opinion.

11.1 Local games

Each connected eyespace of a dragon affords a local game which yields a local game tree. The score of this local game is the number of eyes it yields. Usually if the players take turns and make optimal moves, the end scores will differ by 0 or 1. In this case, the local game may be represented by a single number, which is an integer or half integer. Thus if '`n(O)`' is the score if '`O`' moves first, both players alternate (no passes) and make alternate moves, and similarly '`n(X)`', the game can be represented by '`{n(O)|n(X)}`'. Thus `{1|1}` is an eye, `{2|1}` is an eye plus a half eye, etc.

The exceptional game `{2|0}` can occur, though rarely. We call an eyespace yielding this local game a CHIMERA. The dragon is alive if any of the local games ends up with a score of 2 or more, so `{2|1}` is not different from `{3|1}`. Thus `{3|1}` is NOT a chimera.

Here is an example of a chimera:

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

11.2 Eye spaces

In order that each eyespace be assignable to a dragon, it is necessary that all the dragons surrounding it be amalgamated (see Section 10.2 [Amalgamation], page 72). This is the function of `dragon_eye()`.

An EYE SPACE for a black dragon is a collection of vertices adjacent to a dragon which may not yet be completely closed off, but which can potentially become eyespace. If an open eye space is sufficiently large, it will yield two eyes. Vertices at the edge of the eye space (adjacent to empty vertices outside the eye space) are called MARGINAL.

Here is an example from a game:

```
|. 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
```

Here the '`O`' dragon which is surrounded in the center has open eye space. In the middle of this open eye space are three dead '`X`' stones. This space is large enough that O cannot be killed. We can abstract the properties of this eye shape as follows. Marking certain vertices as follows:

```
|- 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
```

the shape in question has the form:

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

The marginal vertices are marked with an exclamation point ('`!`'). The captured '`X`' stones inside the eyespace are naturally marked '`X`'.

The precise algorithm by which the eye spaces are determined is somewhat complex. Documentation of this algorithm is in the comments in the source to the function `make_domains()` in '`src/optics.c`'.

The eyespaces can be conveniently displayed using a colored ascii diagram by running `gnugo -E`.

11.3 The eyespace as local game

In the abstraction, an eyespace consists of a set of vertices labelled:

```
!  .  X
```

Tables of many eyespaces are found in the database '`patterns/eyes.db`'. Each of these may be thought of as a local game. The result of this game is listed after the eyespace in the form :max,min, where max is the number of eyes the pattern yields if '`O`' moves first, while min is the number of eyes the pattern yields if '`X`' moves first. The player who owns the eye space is denoted '`O`' throughout this discussion. Since three eyes are no better than two, there is no attempt to decide whether the space yields two eyes or three, so max never exceeds 2. Patterns with min>1 are omitted from the table.

For example, we have:

```
Pattern 1

  x
!x*x

:2,1
```

Here notation is as above, except that '`x`' means '`X`' or `EMPTY`. The result of the pattern is not different if '`X`' has stones at these vertices or not.

We may abstract the local game as follows. The two players '`O`' and '`X`' take turns moving, or either may pass.

RULE 1: '`O`' for his move may remove any vertex marked '`!`' or marked '`.`' .

RULE 2: '`X`' for his move may replace a '`.`' by an '`X`'.

RULE 3: '`X`' may remove a '`!`'. In this case, each '`.`' adjacent to the `"!"` which is removed becomes a `"!"` . If an `"`'`X`'`"` adjoins the `"!"` which is removed, then that `"`'`X`'`"` and any which are connected to it are also removed. Any '`.`' which are adjacent to the removed '`X`''s then become '`.`'

Thus if '`O`' moves first he can transform the eyeshape in the above example to:

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

However if '`X`' moves he may remove the '`!`' and the '`.`'s adjacent to the '`!`' become '`!`' themselves. Thus if '`X`' moves first he may transform the eyeshape to:

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

NOTE: A nuance which is that after the '`X:1`', '`O:2`' exchange below, '`O`' is threatening to capture three X stones, hence has a half eye to the left of 2. This is subtle, and there are other such subtleties which our abstraction will not capture. Some of these at least can be dealt with by a refinements of the scheme, but we will content ourselves for the time being with a simplified

```
|- 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
```

We will not attempt to characterize the terminal states of the local game (some of which could be seki) or the scoring.

11.4 An example

Here is a local game which yields exactly one eye, no matter who moves first:

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

Here are some variations, assuming 'O' moves first.

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


...      (after 'O''s move)
...!


...
..!


...
..


.X.       (nakade)
..
```

Here is another variation:

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

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

     !          (after 'X''s move)
     . .
     ..X!


     . .
     ..X!

     . !
     .!
```

11.5 Graphs

It is a useful observation that the local game associated with an eyespace depends only on the underlying graph, which as a set consists of the set of vertices, in which two elements are connected by an edge if and only if they are adjacent on the Go board. For example the two eye shapes:

```
     ..
      ..

     and

     ....
```

though distinct in shape have isomorphic graphs, and consequently they are isomorphic as local games. This reduces the number of eyeshapes in the database '`patterns/eyes.db`'.

A further simplification is obtained through our treatment of half eyes and false eyes. Such patterns are tabulated in the database hey.h. During make_worms, which runs before the eye space analysis, the half eye and false eye patterns are tabulated in the array `half_eye`.

A half eye is isomorphic to the pattern `(!.)` . To see this, consider the following two eye shapes:

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

     and:

     XXOOOOO
     XOa...O
     XbOOOOO
     XXXXXX
```

These are equivalent eyeshapes, with isomorphic local games `{2|1}`. The first has shape:

```
!....
```

The second eyeshape has a half eye at a which is taken when '`O`' or '`X`' plays at '`b`'. This is found by the topological criterion (see Section 11.7 [Eye Topology], page 87).

```
ooo      half eye
OhO
*OX
```

and it is recorded in the half_eye array as follows. If `(i,j)` are the coordinates of the point '`a`', `half_eye[i][j].type==HALF_EYE` and `(half_eye[i][j].ki, half_eye[i][j].kj)` are the coordinates of '`b`'.

The graph of the eye_shape, ostensibly '`....`' is modified by replacing the left '`.`' by '`!`'.

11.6 Eye shape analysis

The patterns in '`patterns/eyes.db`' are compiled into graphs represented essentially by linked lists in '`patterns/eyes.c`'.

Each actual eye space as it occurs on the board is also compiled into a graph. Half eyes are handled as follows. Referring to the example

```
XXOOOOO
XOa...O
XbOOOOO
XXXXXX
```

repeated from the preceding discussion, the vertex at '`b`' is added to the eyespace as a marginal vertex. The adjacency condition in the graph is a macro (in '`optics.c`'): two vertices are adjacent if they are physically adjacent, or if one is a half eye and the other is its key point.

In recognize_eyes, each such graph arising from an actual eyespace is matched against the graphs in '`eyes.c`'. If a match is found, the result of the local game is known. If a graph cannot be matched, its local game is assumed to be `{2|2}`.

11.7 Topology of Half Eyes and False Eyes

A HALF EYE is a pattern where an eye may or may not materialize, depending on who moves first. Here is a half eye for `O`:

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

A FALSE EYE is a cave which cannot become an eye. Here is are two examples of false eyes for `O`:

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

We describe now the topological algorithm used to find half eyes and false eyes.

False eyes and half eyes can locally be characterized by the status of the diagonal intersections from an eye space. For each diagonal intersection, which is not within the eye space, there are three distinct possibilities:

- occupied by an enemy (`X`) stone, which cannot be captured.
- either empty and `X` can safely play there, or occupied by an `X` stone that can both be attacked and defended.
- occupied by an `O` stone, an `X` stone that can be attacked but not defended, or it's empty and `X` cannot safely play there.

We give the first possibility a value of two, the second a value of one, and the last a value of zero. Summing the values for the diagonal intersections, we have the following criteria:

- sum >= 4: false eye
- sum == 3: half eye
- sum <= 2: proper eye

If the eye space is on the edge, the numbers above should be decreased by 2. An alternative approach is to award diagonal points which are outside the board a value of 1. To obtain an exact equivalence we must however give value 0 to the points diagonally off the corners, i.e. the points with both coordinates out of bounds.

The algorithm to find all topologically false eyes and half eyes is:

For all eye space points with at most one neighbor in the eye space, evaluate the status of the diagonal intersections according to the criteria above and classify the point from the sum of the values.

11.8 False Margins

The following situation is rare but special enough to warrant separate attention:

```
OOOOXX
OXaX..
------
```

Here '`a`' may be characterized by the fact that it is adjacent to O's eyespace, and it is also adjacent to an X group which cannot be attacked, but that an X move at 'a' results in a string with only one liberty. We call this a *false margin*.

For the purpose of the eye code, O's eyespace should be parsed as `(X)`, not `(X!)`.

11.9 Functions in 'optics.c'

Here are the public functions in 'optics.c'. The statically declared functions are documented in the source code.

- `void make_domains(struct eye_data b_eye[MAX_BOARD][MAX_BOARD], struct eye_data w_eye[MAX_BOARD][MAX_BOARD])`

  This function is called from make_dragons(). It marks the black and white domains (eyeshape regions) and collects some statistics about each one.

- `void originate_eye(int i, int j, int m, int n, int *esize, int *msize, struct eye_data eye[MAX_BOARD][MAX_BOARD])`

  originate_eye(i, j, i, j, *size) creates an eyeshape with origin (i, j). the last variable returns the size. The repeated variables (i, j) are due to the recursive definition of the function.

- `static void print_eye(struct eye_data eye[MAX_BOARD][MAX_BOARD], int i, int j)`

  Print debugging data for the eyeshape at (i,j). Useful with GDB.

- `void compute_eyes(int i, int j, int *max, int *min, int *attacki, int *attackj, struct eye_data eye[MAX_BOARD][MAX_BOARD], int add_moves, int color)`

  Given an eyespace with origin (i,j), this function computes the minimum and maximum numbers of eyes the space can yield. If `add_moves==1`, this function may add a move_reason for `color` at a vital point which is found by the function. If `add_moves==0`, set `color==EMPTY`.

- `void compute_eyes_pessimistic(int i, int j, int *max, int *min, int *pessimistic_min, int *attacki, int *attackj, int *defendi, int *defendj, struct eye_data eye[MAX_BOARD][MAX_BOARD], struct half_eye_data heye[MAX_BOARD][MAX_BOARD])`

  This function works like compute_eyes(), except that it also gives a pessimistic view of the chances to make eyes. Since it is intended to be used from the owl code, the option to add move reasons has been removed.

- `void propagate_eye (int i, int j, struct eye_data eye[MAX_BOARD][MAX_BOARD])`

  Copies the data at the origin (i, j) to the rest of the eye (certain fields only).

- `static int recognize_eye(int i, int j, int *ai, int *aj, int *di, int *dj, int *max, int *min, struct eye_data eye[MAX_BOARD][MAX_BOARD], struct half_eye_data heye[MAX_BOARD][MAX_BOARD], int add_moves, int color)`

  Declared static but documented here because of its importance. The life code supplies an alternative version of this function called `recognize_eye2()`. Here `(i,j)` is the origin of an eyespace. Returns 1 if there is a pattern in 'eyes.c' matching the eyespace, or 0 if no match is found. If there is a key point for attack, `(*ai, *aj)` are set to its location, or `(-1, -1)` if there is none. Similarly `(*di, *dj)` is the location of a vital defense point. `*min` and `*max` are the minimum and maximum number of eyes that can be made in this eyespace respectively. Vital attack/defense points exist if and only if `*min != *max`. If `add_moves==1`, this function may add a move_reason for (color) at a vital point which is found by the function. If `add_moves==0`, set `color==EMPTY`.

- `void add_half_eye(int m, int n, struct eye_data eye[MAX_BOARD][MAX_BOARD], struct half_eye_data hey[MAX_BOARD][MAX_BOARD])`

  This function adds a half eye or false eye to an eye shape.

- `int eye_space(int i, int j)`

  Used from constraints to identify eye spaces, primarily for late endgame moves. This returns true if the location is an eye space of either color.

- `int proper_eye_space(int i, int j)`

  Used from constraints to identify proper eye spaces, primarily for late endgame moves. Returns true if the location is an eye space of either color and is not marginal.

- `int marginal_eye_space(int i, int j)`

  Used from constraints to identify marginal eye spaces, primarily for late endgame moves. Returns true if the location is a marginal eye space of either color.

- `void make_proper_eye_space(int i, int j, int color)`

  Turn a marginal eye space into a proper eye space.

- `void remove_half_eye(int m, int n, int color)`

  Remove a halfeye from an eye shape.

- `void remove_eyepoint(int m, int n, int color)`

  Remove an eye point. This function can only be used before the segmentation into eyespaces.

- `int topological_eye(int m, int n, int color, int *ai, int *aj, int *di, int *dj, struct eye_data b_eye[MAX_BOARD][MAX_BOARD], struct eye_data w_eye[MAX_BOARD][MAX_BOARD], struct half_eye_data heye[MAX_BOARD][MAX_BOARD])`

  See See Section 11.7 [Eye Topology], page 87. Evaluate the eye space at `(m, n)` topologically (see Section 11.7 [Eye Topology], page 87). Returns 2 or less if `(m, n)` is a proper eye for (color); 3 if `(m, n)` is a half eye; 4 if `(m, n)` is a false eye. `(*ai, *aj)` and `(*di, *dj)` return the coordinates of an unsettled diagonal intersection, or an attack or defense point of defense of an opponent stone occupying a diagonal intersection.

- `int evaluate_diagonal_intersection(int m, int n, int color, int *vitali, int *vitalj)`

  Evaluate an intersection which is diagonal to an eye space (see Section 11.7 [Eye Topology], page 87). Returns 0 if the opponent cannot safely play at the vertex; Returns 1 if empty and the opponent can safely play on it, or if the vertex is occupied by an opponent stone which can be either attacked or defended. Returns 2 if safely occupied by the opponent. Exception: if one coordinate is off the board, returns 1; if both are off the board, returns 0. This guarantees correct behavior for diagonal intersections of points on the edge or in the corner. If the return value is 1, `(*vitali, *vitalj)` returns `(m, n)` if the vertex is empty, or the vital point of defense if it is occupied by an opponent stone.

12 The Pattern Code

12.1 Overview

Several pattern databases are in the patterns directory. This chapter primarily discusses the patterns in '`patterns.db`', '`patterns2.db`', and the pattern files '`hoshi.db`' etc. which are compiled from the SGF files '`hoshi.sgf`' (see Section 12.16 [Joseki Compiler], page 112). There is no essential difference between these files, except that the ones in '`patterns.db`' and '`patterns2.db`' are hand written. They are concatenated before being compiled by `mkpat` into `patterns.c`. The purpose of the separate file '`patterns2.db`' is that it is handy to move patterns into a new directory in the course of organizing them. The patterns in '`patterns.db`' are more disorganized, and are slowly being moved to '`patterns2.db`'.

During the execution of `genmove()`, the patterns are matched in '`shapes.c`' in order to find move reasons.

The same basic pattern format is used by '`attack.db`', '`defense.db`', '`conn.db`', '`apats.db`' and '`dpats.db`'. However these patterns are used for different purposes. These databases are discussed in other parts of this documentation. The patterns in '`eyes.db`' are entirely different and are documented elsewhere (see Chapter 11 [Eyes], page 82).

The patterns described in the databases are ascii representations, of the form:

Pattern EB112

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

:8,ed,NULL
```

Here 'O' marks a friendly stone, 'X' marks an enemy stone, '.' marks an empty vertex, '*' marks O's next move, 'o' marks a square either containing 'O' or empty but not X. (The symbol 'x', which does not appear in this pattern, means 'X' or '.'.) Finally '?' Indicates a location where we don't care what is there, except that it cannot be off the edge of the board.

The line of -'s along the bottom in this example is the edge of the board itself—this is an edge pattern. Corners can also be indicated. Elements are not generated for '?' markers, but they are not completely ignored - see below.

The line beginning '`:`' describes various attributes of the pattern, such as its symmetry and its class. Optionally, a function called a "helper" can be provided to assist the matcher in deciding whether to accept move. Most patterns do not require a helper, and this field is filled with NULL.

The matcher in '`matchpat.c`' searches the board for places where this layout appears on the board, and the callback function `shapes_callback()` in '`shapes.c`' registers the appropriate move reasons.

After the pattern, there is some supplementary information in the format:

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

Here trfno represents the number of transformations of the pattern to consider, usually 8 (no symmetry, for historical reasons), or one of `|` `\` `/` `-` `+` `X`, where the line represents the axis of symmetry. (E.g. `|` means symmetrical about a vertical axis.)

The above pattern could equally well be written on the left edge:

```
|?X?.?
|O.*oo
|O....
|o....

:8,ed,NULL
```

The program `mkpat` is capable of parsing patterns written this way, or for that matter, on the top or right edges, or in any of the four corners. As a matter of convention all the edge patterns in '`patterns.db`' are written on the bottom edge or in the lower left corners. In the '`patterns/`' directory there is a program called `transpat` which can rotate or otherwise transpose patterns. This program is not built by default—if you think you need it, `make transpat` in the '`patterns/`' directory and consult the usage remarks at the beginning of '`patterns/transpat.c`'.

12.2 Pattern Attributes

The attribute field in the '`:`' line of a pattern consists of a sequence of zero or more of the following characters, each with a different meaning. The attributes may be roughly classified as *constraints*, which determine whether or not the pattern is matched, and *actions*, which describe what is to be done when the pattern is matched, typically to add a move reason.

12.2.1 Constraint Pattern Attributes

'`s`'

Safety of the move is not checked. This is appropriate for sacrifice patterns. If this classification is omitted, the matcher requires that the stone played cannot be trivially captured. Even with s classification, a check for legality is made, though.

'`n`'

In addition to usual check that the stone played cannot be trivially captured, it is also confirmed that an opponent move here could not be captured.

'`O`'

It is checked that every friendly ('`O`') stone of the pattern belongs to a dragon which has matcher_status (see Section 10.7 [Dragons], page 76) ALIVE or UNKNOWN. The CRITICAL matcher status is excluded. It is possible for a string to have ALIVE matcher_status and still be tactically critical, since it might be amalgamated into an ALIVE dragon, and the matcher status is constant on the dragon. Therefore, an additional test is performed: if the pattern contains a string which is tactically critical, and if '`*`' does not rescue it, the pattern is rejected.

'`o`'

It is checked that every friendly ('`O`') stone of the pattern belongs to a dragon which is classified as DEAD or UNKNOWN.

'`X`'

It is checked that every opponent ('`X`') stone of the pattern belongs to a dragon with matcher_status ALIVE, UNKNOWN or CRITICAL. Note that there is an asymmetry with '`O`' patterns, where CRITICAL dragons are rejected.

'`x`'

It is checked that every opponent ('`X`') stone of the pattern belongs to a dragon which is classified as DEAD or UNKNOWN

12.2.2 Action Attributes

'`C`'

If two or more distinct O dragons occur in the pattern, the move is given the move reasons that it connects each pair of dragons. An exception is made for dragons where the underlying worm can be tactically captured and is not defended by the considered move.

'`c`'

Add strategical defense move reason for all our dragons and a small shape bonus. This classification is appropriate for weak connection patterns.

'`B`'

If two or more distinct X dragons occur in the pattern, the move is given the move reasons that it cuts each pair of dragons.

'`b`'

The move secures territory by blocking it from intrusion.

'`e`'

The move makes territory by expanding, e.g. along the edge.

'`E`'

The move attempts increase influence and create/expand a moyo.

'`d`'

The move strategically defends all O dragons in the pattern, except those that can be tactically captured and are not tactically defended by this move. If any O dragon should happen to be perfectly safe already, this only reflects in the move reason being valued to zero.

'`a`'

The move strategically attacks all X dragons in the pattern.

'`J`'

Standard joseki move. Unless there is an urgent move on the board these moves are made as soon as they can be. This is equivalent to adding the 'd' and 'a' classifications together with a shape bonus of 5 and a minimum accepted value of 25.

'j'
: Slightly less urgent joseki move. These moves will be made after those with the 'J' classification. This is equivalent to adding the 'e' and 'E' classifications together with a minimum accepted value of 22.

't'
: Minor joseki move (tenuki OK). This is equivalent to adding the 'e' and 'E' classifications together with a minimum accepted value of 18.

'U'
: Urgent joseki move (never tenuki). This is equivalent to the 'd' and 'a' classifications together with a shape bonus of 20 and a minimum accepted value of 50.

A commonly used class is OX (which rejects pattern if either side has dead stones). The string '-' may be used as a placeholder. (In fact any characters other than the above and ',' are ignored.)

The types o and O could conceivably appear in a class, meaning it applies only to UNKNOWN. X and x could similarly be used together. All classes can be combined arbitrarily.

12.3 Pattern Attributes

The second third field in the ':' line of a pattern is optional and of the form `value1(x),value2(y),...`. The available set of values are as follows.

- `terri(x)`

  Forces the territorial value of the move to be at most x
- `minterri(x)` :

  Forces the territorial value of the move to be at least x
- `maxterri(x)` :

  Forces the territorial value of the move to be at most x.
- `value(x)` :

  Forces the final value of the move to be at least x.
- `minvalue(x)` `maxvalue(x)` :

  Forces the final value of the move to be at last/most x.
- `shape(x)` :

  Adds x to the move's shape value.
- `followup(x)` :

  Adds x to the move's followup value.

The meaning of these values is documented in See Chapter 9 [Move Generation], page 55.

12.4 Helper Functions

Helper functions can be provided to assist the matcher in deciding whether to accept a pattern, register move reasons, and setting various move values. The helper is supplied with the compiled pattern entry in the table, and the (absolute) position on the board of the '*' point.

One difficulty is that the helper must be able to cope with all the possible transformations of the pattern. To help with this, the OFFSET macro is used to transform relative pattern coordinates to absolute board locations.

The actual helper functions are in 'helpers.c'. They are declared in 'patterns.h'.

As an example to show how to write a helper function, we consider `wedge_helper`. (This helper does not exist anymore but has been replaced by a constraint, discussed in the following section. Due to its simplicity it's still a good example.) The helper begins with a comment:

```
/*

?O.          ?Ob
.X*          aXt
?O.          ?Oc

:8,C,wedge_helper
*/
```

The image on the left is the actual pattern. On the right we've taken this image and added letters to label (ti, tj), (ai, aj) and (bi, bj). Of course t is always at *, the point where GNU Go will move if the pattern is adopted.

```
int
wedge_helper (ARGS)
{
  int ai, aj, bi, bj, ci, cj;
  int other = OTHER_COLOR(color);
  int success = 0;

  OFFSET(0, -2, ai, aj);
  OFFSET(-1, 0, bi, bj);
  OFFSET(1, 0, ci, cj);

  if (TRYMOVE(ti, tj, color)) {
    if (TRYMOVE(ai, aj, other)) {
      if (!p[ai][aj] || attack(ai, aj, NULL, NULL))
success = 1;
      else if (TRYMOVE(bi, bj, color)) {
if (!safe_move(ci, cj, other))
  success = 1;
popgo();
      }
      popgo();
    }
```

```
      popgo();
    }

    return success;
  }
```

The OFFSET lines tell GNU Go the positions of the three stones at a=(ai,aj), b=(bi,bj), and c=(ci,cj). To decide whether the pattern guarantees a connection, we do some reading. First we use the TRYMOVE macro to place an O at t and let X draw back to a. Then we try whether O can capture these stones by calling attack(). The test if there is a stone at a before calling attack() is in this position not really necessary but it's good practice to do so, because if the attacked stone should happen to already have been captured while placing stones, GNU Go would crash with an assertion failure.

If this attack fails we let O connect at b and use the safe_move() function to examine whether a cut by X at c could be immediately captured. Before we return the result we need to remove the stones we placed from the reading stack. This is done with the function popgo().

12.5 Autohelpers and Constraints

In addition to the hand-written helper functions in 'helpers.c', GNU Go can automatically generate helper functions from a diagram with labels and an expression describing a constraint. The constraint diagram, specifying the labels, is placed below the ":" line and the constraint expression is placed below the diagram on line starting with a ";". Constraints can only be used to accept or reject a pattern. If the constraint evaluates to zero (false) the pattern is rejected, otherwise it's accepted (still conditioned on passing all other tests of course). To give a simple example we consider a connection pattern.

Pattern Conn311

```
O*.
?XO

:8,C,NULL

O*a
?BO

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

Here we have given the label 'a' to the empty spot to the right of the considered move and the label 'B' to the 'X' stone in the pattern. In addition to these, '*' can also be used as a label. A label may be any lowercase or uppercase ascii letter except OoXxt. By convention we use uppercase letters for X stones and lowercase for O stones and empty intersections. When labeling a stone that's part of a larger string in the pattern, all stones of the string should be marked with the label. (These conventions are not enforced by the pattern compiler, but to make the database consistent and easy to read they should be followed.)

The labels can now be used in the constraint expression. In this example we have a reading constraint which should be interpreted as "Play an O stone at * followed by an X stone at a. Accept the pattern if O now can capture either at a or at B (or both strings)."

The functions that are available for use in the constraints are listed in the section 'Autohelpers Functions' below. Technically the constraint expression is transformed by mkpat into an automatically generated helper function in '`patterns.c`'. The functions in the constraint are replaced by C expressions, often functions calls. In principle any valid C code can be used in the constraints, but there is in practice no reason to use anything more than boolean and arithmetic operators in addition to the autohelper functions. Constraints can span multiple lines, which are then concatenated.

12.6 Autohelper Actions

As a complement to the constraints, which only can accept or reject a pattern, one can also specify an action to perform when the pattern has passed all tests and finally has been accepted.

Example:

```
Pattern EJ4

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

:8,Ed,NULL

...*.     never play a here
.OOX.
.aXOX
.....
-----

>antisuji(a)
```

The line starting with '`>`' is the action line. In this case it tells the move generation that the move at a should not be considered, whatever move reasons are found by other patterns. The action line uses the labels from the constraint diagram. Both constraint and action can be used in the same pattern. If the action only needs to refer to '`*`', no constraint diagram is required. Like constraints, actions can span multiple lines.

12.7 Autohelper Functions

The autohelper functions are translated into C code by the program in '`mkpat.c`'. To see exactly how the functions are implemented, consult the autohelper function definitions in that file. Autohelper functions can be used in both constraint and action lines.

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

Number of first, second, third, and fourth order liberties of a worm respectively. See Chapter 10 [Worms and Dragons], page 67, the documentation on worms for definitions.

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

The number of liberties that an enemy or own stone, respectively, would obtain if played at the empty intersection x.

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

Calls `cut_possible` (see Section 18.1 [General Utilities], page 156) to determine whether 'X' or 'O' can cut at the empty intersection x.

```
ko(x)
```

True if x is either a stone or an empty point involved in a ko position.

```
status(x)
```

The matcher status of a dragon. status(x) returns an integer that can have the values `ALIVE`, `UNKNOWN`, `CRITICAL`, or `DEAD` (see Chapter 10 [Worms and Dragons], page 67).

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

Each function true if the dragon has the corresponding matcher status and false otherwise (see Chapter 10 [Worms and Dragons], page 67).

```
status(x)
```

Returns the status of the dragon at 'x' (see Chapter 10 [Worms and Dragons], page 67).

```
genus(x)
```

The number of eyes of a dragon. It is only meaningful to compare this value against 0, 1, or 2.

```
xarea(x)
oarea(x)
xmoyo(x)
omoyo(x)
xterri(x)
oterri(x)
```

Functions related to various kinds of influence and territory estimations, as described in See Chapter 17 [Moyo], page 152. xarea(x) evaluates to true if x is either a living enemy stone or an empty point within his "area". `oarea(x)` is analogous but with respect to our stones and area. The main difference between area, moyo, and terri is that area is a very far reaching kind of influence, moyo gives a more realistic estimate of what may turn in to territory, and terri gives the points that already are believed to be secure territory.

```
weak(x)
```

True for a dragon that is perceived as weak. The definition of weak is given in See Chapter 17 [Moyo], page 152.

```
attack(x)
defend(x)
```

Results of tactical reading. attack(x) is true if the worm can be captured, defend(x) is true if there also is a defending move. Please notice that defend(x) will return false if there is no attack on the worm.

```
safe_xmove(x)
safe_omove(x)
```

True if an enemy or friendly stone, respectively, can safely be played at x. By safe it is understood that the move is legal and that it cannot be captured right away.

```
legal_xmove(x)
legal_omove(x)
```

True if an enemy or friendly stone, respectively, can legally be played at x.

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

True if O (respectively X) has a stone at one of the labelled vertices. In the diagram, these vertices should be marked with a '?'.

```
odefend_against(x,y)
xdefend_against(x,y)
```

True if an own stone at x would stop the enemy from safely playing at y, and conversely for the second function.

```
does_defend(x,y)
does_attack(x,y)
```

True if a move at x defends/attacks the worm at y. For defense a move of the same color as y is tried and for attack a move of the opposite color.

```
xplay_defend(a,b,c,...,z)
```

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

These functions make it possible to do more complex reading experiments in the constraints. All of them work so that first the sequence of moves a,b,c,... is played through with alternating colors, starting with X or O as indicated by the name. Then it is tested whether the worm at z can be attacked or defended, respectively. It doesn't matter who would be in turn to move, a worm of either color may be attacked or defended. For attacks the opposite color of the string being attacked starts moving and for defense the same color starts. The defend functions return true if the worm cannot be attacked in the position or if it can be attacked but also defended. The attack functions return true if there is a way to capture the worm, whether or not it can also be defended. If there is no stone present at z after the moves have been played, it is assumed that an attack has already been successful or a defense has already failed. If some of the moves should happen to be illegal, typically because it would have been suicide, the following moves are played as if nothing has happened and the attack or defense is tested as usual. It is assumed that this convention will give the relevant result without requiring a lot of special cases.

The special label '`?`' can be used to represent a tenuki. Thus `oplay_defend(a,?,b,c)` tries moves by '`O`' at '`a`' and '`b`', as if '`X`' plays the second move in another part of the board, then asks if '`c`' can be defended. The tenuki cannot be the first move of the sequence, nor does it need to be: instead of `oplay_defend(?,a,b,c)` you can use `xplay_defend(a,b,c)`.

```
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)
```

These functions are similar to the previous ones. The difference is that the last *two* arguments denote worms to be attacked or defended simultaneously. Obviously y and z must have the same color. If either location is empty, it is assumed that an attack has been successful or a defense has failed. The typical use for these functions is in cutting patterns, where it usually suffices to capture either cutstone.

The function `xplay_defend_both` plays alternate moves beginning with an X at '`a`'. Then it passes the last two arguments to `defend_both` in '`engine/utils.c`'. This function checks to determine whether the two strings can be simultaneously defended.

The function `xplay_attack_either` plays alternate moves beginning with an X move at '`a`'. Then it passes the last two arguments to `attack_either` in '`engine/utils.c`'. This function looks for a move which captures at least one of the two strings. In its current implementation `attack_either` only looks for uncoordinated attacks and would thus miss a double atari.

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

These functions are used to set up a position like

```
.O.    .y.
OXO    xXz
```

and X aims at capturing at least one of x, y, and z. If this succeeds 1 is returned. If it doesn't, X tries instead to cut through on either side and if this succeeds, 2 is returned. Of course the same shape with opposite colors can also be used.

Important notice: x, y, and z must be given in the order they have in the diagram above, or any reflection and/or rotation of it.

```
seki_helper(x)
```

Checks whether the string at 'x' can attack any surrounding string. If so, return false as the move to create a seki (probably) wouldn't work.

```
threaten_to_save(x)
```

Calls `add_followup_value` to add as a move reason a conservative estimate of the value of saving the string 'x' by capturing one opponent stone.

```
area_stone(x)
```

Returns the number of stones in the area around 'x'.

```
area_space(x)
```

Returns the amount of space in the area around 'x'.

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

True if x is an eye space for either color, a non-marginal eye space for either color, or a marginal eye space for either color, respectively.

```
antisuji(x)
```

Tell the move generation that x is a substandard move that never should be played.

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

Return true if x and y are the same dragon or worm respectively.

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

Number of stones in the indicated dragon or worm.

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

Explicitly notify the move generation about move reasons for the move in the pattern.

```
halfeye(x)
```

Returns true if the empty intersection at 'x' is a half eye.

```
remove_attack(x)
```

Inform the tactical reading that a supposed attack does in fact not work.

```
potential_cutstone(x)
```

True if cutstone2 field from worm data is larger than one. This indicates that saving the worm would introduce at least two new cutting points.

```
not_lunch(x,y)
```

Prevents the misreporting of 'x' as lunch for 'y'. For example, the following pattern tells GNU Go that even though the stone at 'a' can be captured, it should not be considered "lunch" for the dragon at 'b', because capturing it does not produce an eye:

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

> not_lunch(a,b)
```

```
vital_chain(x)
```

Calls `vital_chain` to determine whether capturing the stone at 'x' will result in one eye for an adjacent dragon. The current implementation just checks that the stone is not a singleton on the first line.

```
amalgamate(x,y)
```

Amalgamate (join) the dragons at 'x' and 'y' (see Chapter 10 [Worms and Dragons], page 67).

```
amalgamate_most_valuable(x,y,z)
```

Called when (x,y,z) point to three (preferably distinct) dragons, in situations such as this:

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

In this situation, the opponent can play at *, preventing the three dragons from becoming connected. However 'O' can decide which cut to allow. The helper amalgamates the dragon at 'y' with either 'x' or 'z', whichever is largest.

```
make_proper_eye(x)
```

This autohelper should be called when 'x' is an eyespace which is misidentified as marginal. It is reclassified as a proper eyespace (see Section 11.2 [Eye Space], page 82).

```
remove_halfeye(x)
```

Remove a half eye from the eyespace. This helper should not be run after `make_dragons` is finished, since by that time the eyespaces have already been analyzed.

```
remove_eyepoint(x)
```

Remove an eye point. This function can only be used before the segmentation into eyespaces.

```
owl_topological_eye(x,y)
```

Here 'x' is an empty intersection which may be an eye or half eye for some dragon, and 'y' is a stone of the dragon, used only to determine the color of the eyespace in question.

Returns the sum of the values of the diagonal intersections, relative to 'x', as explained in See Section 11.7 [Eye Topology], page 87, equal to 4 or more if the eye at 'x' is false, 3 if it is a half eye, and 2 if it is a true eye.

```
owl_escape_value(x)
```

Returns the escape value at 'x'. This is only useful in owl attack and defense patterns and only if the '--alternative_escape' option is turned on. Otherwise 0 is returned.

12.8 Attack and Defense Database

The patterns in 'attack.db' and 'defense.db' are used to assist the tactical reading in finding moves that attacks or defends worms. The matching is performed during make_worms(), at the time when the tactical status of all worms is decided. None of the classes described above are useful in these databases, instead we have two other classes.

D : For each O worm in the pattern that can be tactically captured (worm[m][n].attack_code != 0), the move at '*' is tried. If it is found to defend the stone, this is registered as a reason for the move * and the defense point of the worm is set to '*'.

A : For each X worm in the pattern, it's tested whether the move at * captures the worm. If that is the case, this is registered as a reason for the move at '*'. The attack point of the worm is set to * and if it wasn't attacked before, a defense is searched for.

Furthermore, A patterns can only be used in 'attack.db' and D patterns only in 'defense.db'. Unclassified patterns may appear in these databases, but then they must work through actions to be effective.

12.9 The Connections Database

The patterns in 'conn.db' are used for helping make_dragons() amalgamate worms into dragons and to some extent for modifying eye spaces. The patterns in this database use the classifications 'B', 'C', and 'e'. 'B' patterns are used for finding cutting points, where amalgamation should not be performed, 'C' patterns are used for finding existing connections, over which amalgamation is to be done, and 'e' patterns are used for modifying eye spaces and reevaluating lunches. There are also some patterns without classification, which use action lines to have an impact. These are matched together with the 'C' patterns. Further details and examples can be found in See Chapter 10 [Worms and Dragons], page 67.

We will illustrate these databases by example. In this situation:

```
XOO
O.O
...
```

'X' cannot play safely at the cutting point, so the 'O' dragons are to be amalgamated. Two patterns are matched here:

```
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))
```

The constraints are mostly clear. For example the second pattern should not be matched if the ‘`X`’ stone cannot be attacked and ‘`X`’ can play safely at ‘`B`’, or if ‘`B`’ is a ko. The constraint `!xcut(B)` means that connection has not previously been inhibited by `find_cuts`. For example consider this situation:

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

The previous pattern is matched here twice, yet X can push in and break one of the connections. To fix this, we include a pattern:

```
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,*)
```

After this pattern is found, the `xcut` autohelper macro will return true at any of the points '`*`', '`a`' and '`b`'. Thus the patterns CB204 and CB205 will not be matched, and the dragons will not be amalgamated.

12.10 Connections Functions

Here are the public functions in '`connections.c`'.

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

  Try to match all (permutations of) connection patterns at (m,n). For each match, if it is a B pattern, set cutting point in worm data structure and make eye space marginal for the connection inhibiting entries of the pattern. If it is a C pattern, amalgamate the dragons in the pattern.
- `void find_cuts(void)`

  Find cutting points which should inhibit amalgamations and sever the adjacent eye space. This goes through the connection database consulting only patterns of type B. When such a function is found, the function `cut_connect_callback` is invoked.
- `void find_connections(void)`

  Find explicit connection patterns and amalgamate the involved dragons. This goes through the connection database consulting patterns except those of type B, E or e. When such a function is found, the function `cut_connect_callback` is invoked.
- void modify_eye_spaces1(void)

  Find explicit connection patterns and amalgamate the involved dragons. This goes through the connection database consulting only patterns of type E (see Section 12.9 [Connections Database], page 104). When such a function is found, the function `cut_connect_callback` is invoked.
- void modify_eye_spaces1(void)

  Find explicit connection patterns and amalgamate the involved dragons. This goes through the connection database consulting only patterns of type e (see Section 12.9 [Connections Database], page 104). When such a function is found, the function `cut_connect_callback` is invoked.

12.11 Tuning the Pattern databases

Since the pattern databases, together with the valuation of move reasons, decide GNU Go's personality, much time can be devoted to "tuning" them. Here are some suggestions.

If you want to experiment with modifying the pattern database, invoke with the -a option. This will cause every pattern to be evaluated, even when some of them may be skipped due to various optimizations.

You can obtain a Smart Go Format (SGF) record of your game in at least two different ways. One is to use CGoban to record the game. You can also have GNU Go record the game in Smart Go Format, using the -o option. It is best to combine this with -a. Do not try to read the SGF file until the game is finished and you have closed the game window. This does not mean that you have to play the game out to its conclusion. You may close the CGoban window on the game and GNU Go will close the SGF file so that you can read it.

If you record a game in SGF form using the -o option, GNU Go will add labels to the board to show all the moves it considered, with their values. This is an extremely useful feature, since one can see at a glance whether the right moves with appropriate weights are being proposed by the move generation.

First, due to a bug of unknown nature, it occasionally happens that GNU Go will not receive the `SIGTERM` signal from CGoban that it needs to know that the game is over. When this happens, the SGF file ends without a closing parenthesis, and CGoban will not open the file. You can fix the file by typing:

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

at the command line to add this closing parenthesis. Or you could add the) using an editor.

Move values exceeding 99 (these should be rare) can be displayed by CGoban but you may have to resize the window in order to see all three digits. Grab the lower right margin of the CGoban window and pull it until the window is large. All three digits should be visible.

If you are playing a game without the -o option and you wish to analyze a move, you may still use CGoban's "Save Game" button to get an SGF file. It will not have the values of the moves labelled, of course.

Once you have a game saved in SGF format, you can analyze any particular move by running:

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

to see why GNU Go made that move, and if you make changes to the pattern database and recompile the program, you may ask GNU Go to repeat the move to see how the behavior changes. If you're using emacs, it's a good idea to run GNU Go in a shell in a buffer (M-x shell) since this gives good navigation and search facilities.

Instead of a move number, you can also give a board coordinate to -L in order to stop at the first move played at this location. If you omit the -L option, the move after those in the file will be considered.

If a bad move is proposed, this can have several reasons. To begin with, each move should be valued in terms of actual points on the board, as accurately as can be expected by the program. If it's not, something is wrong. This may have two reasons. One possibility is that there are reasons missing for the move or that bogus reasons have been found. The other possibility is that the move reasons have been misevaluated by the move valuation functions. Tuning of patterns is with a few exceptions a question of fixing the first kind of problems.

If there are bogus move reasons found, search through the trace output for the pattern that is responsible. (Some move reasons, e.g. most tactical attack and defense, do not originate from patterns. If no pattern produced the bogus move reason, it is not a tuning problem.) Probably this pattern was too general or had a faulty constraint. Try to make it more specific or correct bugs if there were any. If the pattern and the constraint looks right, verify that the tactical reading evaluates the constraint correctly. If not, this is either a reading bug or a case where the reading is too complicated for GNU Go.

If a connecting move reason is found, but the strings are already effectively connected, there may be missing patterns in '`conn.db`'. Similarly, worms may be incorrectly amalgamated due to some too general or faulty pattern in '`conn.db`'. To get trace output from the matching of patterns in '`conn.db`' you need to add a second -t option.

If a move reason is missing, there may be a hole in the database. It could also be caused by some existing pattern being needlessly specific, having a faulty constraint, or being rejected due to a reading mistake. Unless you are familiar with the pattern databases, it may be hard to verify that there really is a pattern missing. Look around the databases to try to get a feeling for how they are organized. (This is admittedly a weak point of the pattern databases, but the goal is to make them more organized with time.) If you decide that a new pattern is needed, try to make it as general as possible, without allowing incorrect matches, by using proper classification from among snOoXx and constraints. The reading functions can be put to good use. The reason for making the patterns as general as they can be is that we need a smaller number of them then, which makes the database much easier to maintain. Of course, if you need too complicated constraints, it's usually better to split the pattern.

If a move has the correct set of reasons but still is misevaluated, this is usually not a tuning problem. There are, however, some possibilities to work around these mistakes with the use of patterns. In particular, if the territorial value is off because `delta_terri()` give strange results, the (min)terri and maxterri values can be set by patterns as a workaround. This is typically done by the endgame patterns, where we can know the (minimum) value fairly well from the pattern. If it should be needed, (min)value and maxvalue can be used similarly. These possibilities should be used conservatively though, since such patterns are likely to become obsolete when better (or at least different) functions for e.g. territory estimation are being developed.

In order to choose between moves with the same move reasons, e.g. moves that connect two dragons in different ways, patterns with a nonzero shape value should be used. These should give positive shape values for moves that give good shape or good aji and negative values for bad shape and bad aji. Notice that these values are additive, so it's important that the matches are unique.

Sente moves are indicated by the use of the pattern followup value. This can usually not be estimated very accurately, but a good rule is to be rather conservative. As usual it should be measured in terms of actual points on the board. These values are also additive so the same care must be taken to avoid unintended multiple matches.

You can also get a visual display of the dragons using the -T option. The default GNU Go configuration tries to build a version with color support using either curses or the ansi escape sequences. You are more likely to find color support in rxvt than xterm, at least on many systems, so we recommend running:

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

in an rxvt window. If you do not see a color display, and if your host is a GNU/Linux machine, try this again in the Linux console.

Worms belonging to the same dragon are labelled with the same letters. The colors indicate the value of the field `dragon.safety`, which is set in '`moyo.c`'.

Green: GNU Go thinks the dragon is alive
Yellow: Status unknown
Blue: GNU Go thinks the dragon is dead
Red: Status critical (1.5 eyes) or weak by the algorithm
in '`moyo.c`'

If you want to get the same game over and over again, you can eliminate the randomness in GNU Go's play by providing a fixed random seed with the -r option.

12.12 Implementation

The pattern code in GNU Go is fairly straightforward conceptually, but because the matcher consumes a significant part of the time in choosing a move, the code is optimized for speed. Because of this there are implementation details which obscure things slightly.

In GNU Go, the ascii '`.db`' files are precompiled into tables (see '`patterns.h`') by a standalone program '`mkpat.c`', and the resulting '`.c`' files are compiled and linked into the main gnugo executable.

Each pattern is compiled to a header, and a sequence of elements, which are (notionally) checked sequentially at every position and orientation of the board. These elements are relative to the pattern 'anchor' (or origin). One X or O stone is (arbitrarily) chosen to represent the origin of the pattern. (We cannot dictate one or the other since some patterns contain only one colour or the other.) All the elements are in co-ordinates relative to this position. So a pattern matches "at" board position (m,n,o) if the the pattern anchor stone is on (m,n), and the other elements match the board when the pattern is transformed by transformation number '`o`'. (See below for the details of the transformations, though these should not be necessary)

12.13 Symmetry and transformations

In general, each pattern must be tried in each of 8 different permutations, to reflect the symmetry of the board. But some patterns have symmetries which mean that it is unnecessary (and therefore inefficient) to try all eight. The first character after the '`:`' can be one of '`8`','`|`','`\`','`/`', '`X`', '`-`', '`+`', representing the axes of symmetry. It can also be '`O`', representing symmetry under 180 degrees rotation.

```
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
```

Then if the pattern has the following symmetries, the following are true:

```
|  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
```

We can choose to use transformations a,d,f,g as the unique transformations for patterns with either ‘|’, ‘-’, ‘\’, or ‘/’ symmetry.

Thus we choose to order the transformations a,g,d,f,h,b,e,c and choose first 2 for ‘X’ and ‘+’, the first 4 for ‘|’, ‘-’, ‘/’, and ‘\’, the middle 4 for ‘O’, and all 8 for non-symmetrical patterns.

Each of the reflection operations (e-h) is equivalent to reflection about one arbitrary axis followed by one of the rotations (a-d). We can choose to reflect about the axis of symmetry (which causes no net change) and can therefore conclude that each of e-h is equivalent to the reflection (no-op) followed by a-d. This argument therefore extends to include - and / as well as | and \.

12.14 Implementation Details

1. An entry in the pattern header states whether the anchor is an X or an O. This helps performance, since all transformations can be rejected at once if the anchor stone does not match. (Ideally, we could just define that the anchor is always O or always X, but some patterns contain no O’s and some contain no X’s.)
2. The pattern header contains the size of the pattern (ie the co-ordinates of the top left and bottom right elements) relative to the anchor. This allows the pattern can be rejected quickly if there is not room for the pattern to fit around the anchor stone in a given orientation (ie it is too near the edge of the board). The bounding box information must first be transformed like the elements before it can be tested, and after transforming, we need to work out where the top-left and bottom-right corners are.
3. The edge constraints are implemented by notionally padding the pattern with rows or columns of ‘?’ until it is exactly 19 (or whatever the current board size is) elements wide or high. Then the pattern is quickly rejected by (ii) above if it is not at the edge. So the example pattern above is compiled as if it was written

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

4. The elements in a pattern are sorted so that non-space elements are checked before space elements. It is hoped that, for most of the game, more squares are empty, and so the pattern can be more quickly rejected doing it this way.

5. The actual tests are performed using an 'and-compare' sequence. Each board position is a 2-bit quantity. %00 for empty, %01 for O, %10 for X. We can test for an exact match by and-ing with %11 (no-op), then comparing with 0, 1 or 2. The test for '`o`' is the same as a test for 'not-X', ie not %10. So and with %01 should give 0 if it matches. Similarly '`x`' is a test that bit 0 is not set.

12.15 The "Grid" Optimization

The comparisons between pattern and board are performed as 2-bit bitwise operations. Therefore they can be performed in parallel, 16-at-a-time on a 32-bit machine.

Suppose the board is layed out as follows :

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

which is internally stored internally in a 2d array (binary)

```
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
```

we can compile this to a composite array in which each element stores the state of a 4x4 grid of squares :

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

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

...

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

Where '??' is off the board.

We can store these 32-bit composites in a 2d merged-board array, substituting the illegal value %11 for '??'.

Similarly, for each pattern, mkpat produces appropriate 32-bit and-value masks for the pattern elements near the anchor. It is a simple matter to test the pattern with a similar test to (5) above, but for 32-bits at a time.

12.16 The Joseki Compiler

GNU Go includes a joseki compiler in '`patterns/joseki.c`'. This processes an SGF file (with variations) and produces a sequence of patterns which can then be fed back into mkpat. The joseki database is currently in files in '`patterns/`' called '`hoshi.sgf`', '`komoku.sgf`', '`sansan.sgf`', '`mokuhazushi.sgf`' and '`takamoku.sgf`'. This division can be revised whenever need arises.

The SGF files are transformed into the pattern database '`.db`' format by the program in '`joseki.c`'. These files are in turn transformed into C code by the program in '`mkpat.c`' and the C files are compiled and linked into the GNU Go binary.

Not every node in the SGF file contributes a pattern. The nodes which contribute patterns have the joseki in the upper right corner, with the boundary marked with a square mark and other information to determine the resulting pattern marked in the comments.

The intention is that the move valuation should be able to choose between the available variations by normal valuation. When this fails the primary workaround is to use shape values to increase or decrease the value. It is also possible to add antisuji variations to forbid popular suboptimal moves. As usual constraints can be used, e.g. to condition a variation on a working ladder.

The joseki format has the following components for each SGF node:

- A square mark (`SQ` or `MA` property) to decide how large part of the board should be included in the pattern.
- A move ('`W`' or '`B`' property) with the natural interpretation. If the square mark is missing or the move is a pass, no pattern is produced for the node.
- Optional labels (`LB` property), which must be a single letter each. If there is at least one label, a constraint diagram will be produced with these labels.
- A comment ('`C`' property). As the first character it should have one of the following characters to decide its classification:
  - '`U`' - urgent move
  - '`S`' or '`J`' - standard move
  - '`s`' or '`j`' - lesser joseki
  - '`T`' - trick move
  - '`t`' - minor joseki move (tenuki OK)
  - '`0`' - antisuji ('`A`' can also be used)

  The rest of the line is ignored, as is the case of the letter. If neither of these is found, it's assumed to be a standard joseki move.

  In addition to this, rows starting with the following characters are recognized:
  - '`#`' - Comments. These are copied into the patterns file, above the diagram.

- '`;`' - Constraints. These are copied into the patterns file, below the constraint diagram.
- '`>`' - Actions. These are copied into the patterns file, below the constraint diagram.
- '`:`' - Colon line. This is a little more complicated, but the colon line of the produced patterns always start out with ":8,s" for transformation number and sacrifice pattern class (it usually isn't a sacrifice, but it's pointless spending time checking for tactical safety). Then a joseki pattern class character is appended and finally what is included on the colon line in the comment for the SGF node.

Example: If the comment in the SGF file looks like

```
F
:C,shape(3)
;xplay_attack(A,B,C,D,*)
```

the generated pattern will have a colon line

```
:8,sjC,shape(3)
```

and a constraint

```
;xplay_attack(A,B,C,D,*)
```

12.17 Ladders in Joseki

As an example of how to use autohelpers with the Joseki compiler, we consider an example where a Joseki is bad if a ladder fails. Assume we have the taisha and are considering connecting on the outside with the pattern

```
--------+
........|
........|
...XX...|
...OXO..|
...*O...|
....X...|
........|
........|
```

But this is bad unless we have a ladder in our favor. To check this we add a constraint which may look like

```
--------+
........|
........|
...XX...|
...OXO..|
...*OAC.|
....DB..|
........|
........|

;oplay_attack(*,A,B,C,D)
```

In order to accept the pattern we require that the constraint on the semicolon line evaluates to true. This particular constraint has the interpretation "Play with alternating colors, starting with '`O`', on the intersections '`*`', '`A`', '`B`', and '`C`'. Then check whether the stone at '`D`' can be captured." I.e. play to this position

```
--------+
........|
........|
...XX...|
...OXO..|
...OOXX.|
....XO..|
........|
........|
```

and call `attack()` to see whether the lower '`X`' stone can be captured. This is not limited to ladders, but in this particular case the reading will of course involve a ladder.

The constraint diagram above with letters is how it looks in the '`.db`' file. The joseki compiler knows how to create these from labels in the SGF node. '`Cgoban`' has an option to create one letter labels, but this ought to be a common feature for SGF editors.

Thus in order to implement this example in SGF, one would add labels to the four intersections and a comment:

```
;oplay_attack(*,A,B,C,D)
```

The appropriate constraint (autohelper macro) will then be added to the Joseki '`.db`' file.

13 The DFA pattern matcher

In this chapter, we describe the principles of the gnugo DFA pattern matcher. The aim of this system is to permit a fast pattern matching when it becomes time critical like in owl module (Section 15.1 [The Owl Code], page 141). The actual version is still experimental but is expected to be fully integrated in later versions of gnugo. If you want to test it with version 3.0 you must run `configure --enable-dfa` then recompile GNU Go (Chapter 13 [Using DFA], page 115). The basic principle is to generate off line a finite state machine called a Deterministic Finite State Automaton (Chapter 13 [What is a DFA], page 116) from the pattern database and then use it at runtime to speedup pattern matching (Chapter 13 [Pattern matching with DFA], page 118 and Chapter 13 [Incremental Algorithm], page 120).

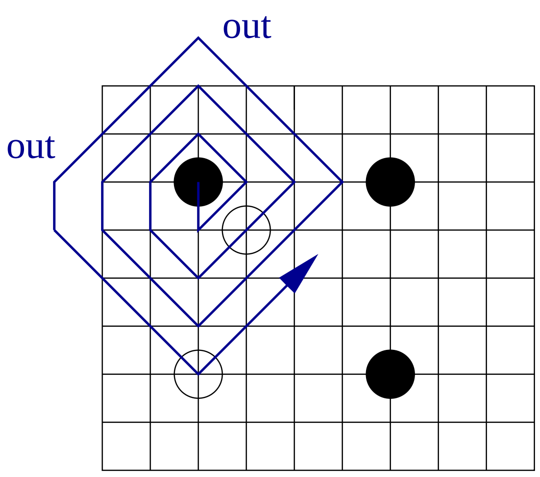
13.0.1 Using DFA

First build the program with 'configure –enable-dfa', then type 'make' as usual.

Some .db files will be compiled into DFA's by the program mkpat. DFA are stored into C files and compiled with the engine. When a DFA is found, gnugo write "<pattern database name> –> using dfa" at startup and use the dfa to "filter" patterns. When no DFA is found, the standard pattern matcher is used.

13.0.2 Scan Path

The board is scanned following a predefined path. The default path is a spiral starting from the center of the pattern. This path is used both to build the DFA and to scan the board.



This path is encoded by two arrays of integers order_i[k] and order_j[k] giving the offset where to read the values on the board.

Reading the board following a predefined path reduces the two dimentional pattern matching to a linear text searching problem. This pattern for example:

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

```
?OO
```

scanned following the path

```
149
238
567
```

```
(i,j)->(i+1,j)->(i+1,j+1)->(i,j+1)->(i+2,j+0)->(i+2,j+1)->(i+2,j+2)...
```

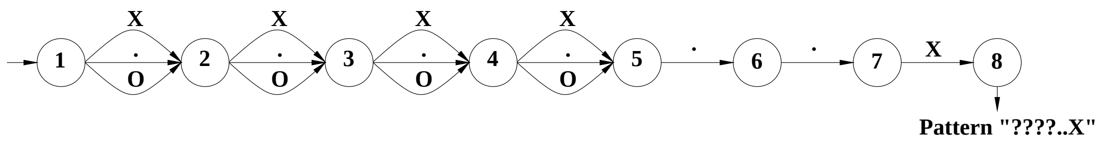
gives the string "**?.OX?OO??**" where "**?**" means **'don't care'**. We can forget the two dimensional patterns for a time to focus on linear patterns.

13.0.3 What is a DFA

The acronym DFA means Deterministic Finite state Automaton (See `http://www.eti.pg.gda.pl/~jandac/thesis/node12.html` or *Hopcroft & Ullman* "*Introduction to Language Theory*" for more details). DFA are common tools in compilers design (Read *Aho, Ravi Sethi, Ullman* "*COMPILERS: Principles, Techniques and Tools*" for a complete introduction), a lot of powerfull text searching algorithm like *Knuth-Morris-Pratt* or *Boyer-Moore* algorithms are based on DFA's (See `http://www-igm.univ-mlv.fr/~lecroq/string/` for a bibliography of pattern matching algorithms).

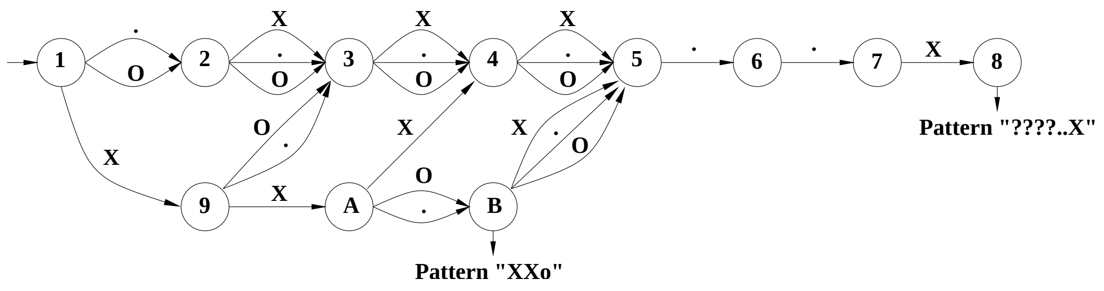
Basically, a DFA is a set of *states* connected by labeled *transitions*. The labels are the values read on the board, in gnugo these values are EMPTY, WHITE, BLACK or OUT_BOARD, denoted respectively by '.','O','X' and '#'.

The best way to represent a dfa is to draw its transition graph: the pattern "**????..X**" is recognized by the following DFA:

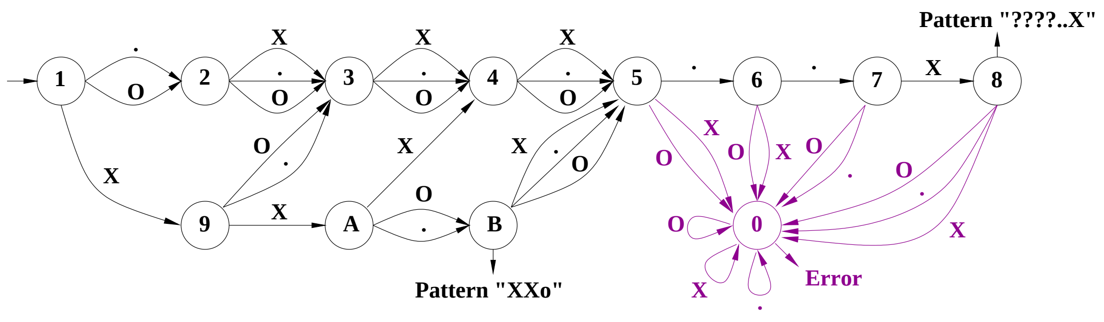


This means that starting from state [1], if you read '.','X' or 'O' on the board, go to state [2] and so on until you reach state [5]. From state [5], if you read '.', go to state [6] otherwise go to error state [0]. And so on until you reach state [8]. As soon as you reach state [8], you recognize Pattern "**????..X**"

Adding a pattern like "**XXo**" ('o' is a wildcard for not 'X') will transform directly the automaton by synchronization product (Chapter 13 [Building the DFA], page 119). Consider the following DFA:



By adding a special *error* state and completing each state by a transition to error state when there is none, we transform easily a DFA in a *Complete Deterministic Finite state Automaton* (CDFA). The synchronization product (Chapter 13 [Building the DFA], page 119) is only possible on CDFA's.



The graph of a CDFA is coded by an array of states: The 0 state is the "error" state and the start state is 1.

state	.	O	X	#	att
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	Found pattern "????..X"
9	3	3	A	0	
A	B	B	4	0	
B	5	5	5	0	Found pattern "XXo"

To each state we associate an often empty list of attributes which is the list of pattern indexes recognized when this state is reached. In '`dfa.h`' this is basically represented by two stuctures:

```
/* 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;
```

13.0.4 Pattern matching with DFA

Recognizing with a DFA is very simple and thus very fast (See '`scan_for_pattern()`' in the '`engine/matchpat.c`' file).

Starting from the start state, we only need to read the board following the spiral path, jump from states to states following the transitions labelled by the values read on the board and collect the patterns indexes on the way. If we reach the error state (zero), it means that no more patterns will be matched. The worst case complexity of this algorithm is o(m) where m is the size of the biggest pattern.

Here is an example of scan:

First we build a minimal dfa recognizing these patterns: "**X..X**", "**X???**", "**X.OX**" and "**X?oX**". Note that wildcards like '?','o', or 'x' give multiple out-transitions.

```
----------------------------------------------------
 state  |   .    |   O    |   X    |   #    |  att
----------------------------------------------------
      1 |      0 |      0 |      2 |      0 |
      2 |      3 |     10 |     10 |      0 |
      3 |      4 |      7 |      9 |      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
----------------------------------------------------
```

We perform the scan of the string "**X..XXO....**" starting from state 1:

Current state: 1, substring to scan : **X..XXO....**

We read an 'X' value, so from state 1 we must go to state 2.

Current state: 2, substring to scan : **..XXO....**

We read a '.' value, so from state 2 we must go to state 3 and so on ...

```
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
```

After reaching state 6 where we match patterns 1,2 and 4, there is no out-transitions so we stop the matching. To keep the same match order as in the standard algorithm, the patterns indexes are collected in an array and sorted by indexes.

13.0.5 Building the DFA

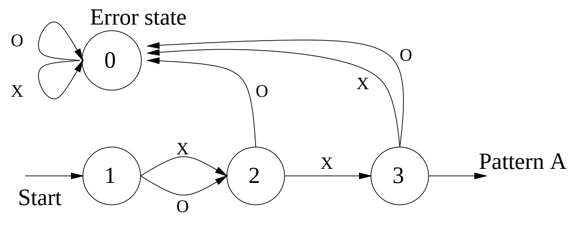
The most flavouring point is the building of the minimal DFA recognizing a given set of patterns. To perform the insertion of a new pattern into an already existing DFA one must completly rebuild the DFA: the principle is to build the minimal CDFA recognizing the new pattern to replace the original CDFA with its *synchronised product* by the new one.

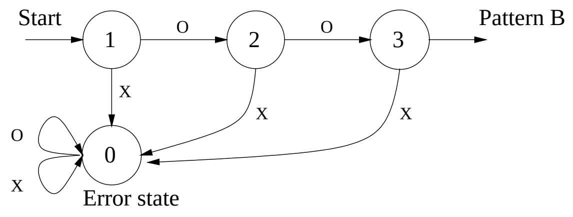
We first give a formal definition: Let **L** be the left CDFA and **R** be the right one. Let **B** be the *synchronised product* of **L** by **R**. Its states are the couples **(l,r)** where **l** is a state of **L** and **r** is a state of **R**. The state **(0,0)** is the error state of **B** and the state **(1,1)** is its initial state. To each couple **(l,r)** we associate the union of patterns recognized in both **l** and **r**. The transitions set of **B** is the set of transitions **(l1,r1)—a—>(l2,r2)** for each symbol **'a'** such that both **l1—a—>l2** in **L** and **r1—a—>r2** in **R**.

The maximal number of states of **B** is the product of the number of states of **L** and **R** but almost all this states are non reachable from the initial state **(1,1)**.

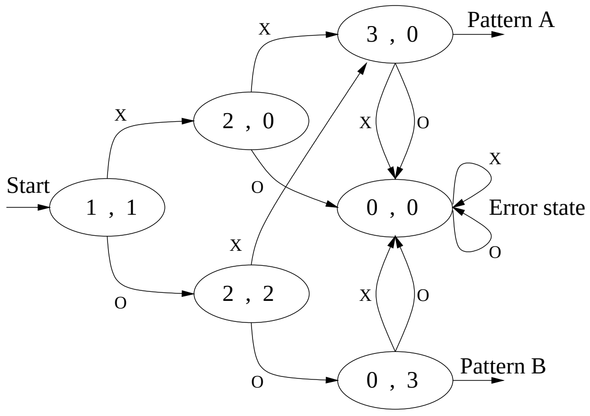
The algorithm used in function '`sync_product()`' builds the minimal product DFA only by keeping the reachable states. It recursively scans the product CDFA by following simultaneously the transitions of **L** and **R**. A hast table (`gtest`) is used to check if a state **(l,r)** has already been reached, the reachable states are remapped on a new DFA. The CDFA thus obtained is minimal and recognizes the union of the two patterns sets.

For example these two CDFA's:





Give by synchronization product the following one:



It is possible to construct a special pattern database that generates an "explosive" automaton: the size of the DFA is in the worst case exponential in the number of patterns it recognizes. But it doesn't occur in pratical situations: the dfa size tends to be *stable*. By *stable* we mean that if we add a pattern which greatly increases the size of the dfa it also increases the chance that the next added pattern does not increase its size at all. Nevertheless there are many ways to reduce the size of the DFA. Good compression methods are explained in *Aho, Ravi Sethi, Ullman "COMPILERS: Principles, Techniques and Tools" chapter Optimization of DFA-based pattern matchers*.

13.0.6 Incremental Algorithm

The incremental version of the DFA pattern matcher is not yet implemented in gnugo but we explain here how it will work. By definition of a deterministic automaton, scanning the same string will reach the same states every time.

Each reached state during pattern matching is stored in a stack `top_stack[i][j]` and `state_stack[i][j][stack_idx]` We use one stack by intersection `(i,j)`. A precomputed reverse path list allows to know for each couple of board intersections `(x,y)` its position `reverse(x,y)` in the spiral scan path starting from `(0,0)`.

When a new stone is put on the board at `(lx,ly)`, the only work of the pattern matcher is:

```
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)];
     }
```

In most situations reverse(lx-i,ly-j) will be inferior to top_stack[i][j]. This should speedup a lot pattern matching.

13.0.7 Some DFA Optimizations

The dfa is constructed to minimize jumps in memory making some assumptions about the frequencies of the values: the EMPTY value is supposed to appear often on the board, so the the '.' transition are almost always successors in memory. The OUT_BOARD are supposed to be rare, so '#' transitions will almost always imply a big jump.

14 Tactical reading

The process of visualizing potential moves done by you and your opponent to learn the result of different moves is called "reading". GNU Go does three distinct types of reading: *tactical reading* which typically is concerned with the life and death of individual strings, *Owl reading* which is concerned with the life and death of dragons, and *life reading* which attempts evaluate eye spaces. In this Chapter, we document the tactical reading code, which is in '`engine/reading.c`'. For a summary of the reading functions see See Section 14.6 [Reading Functions], page 136.

14.1 Reading Basics

In GNU Go, tactical reading is done by the functions in '`engine/reading.c`'. Each of these functions has a separate goal to fill, and they call each other recursively to carry out the reading process.

The reading code makes use of a stack onto which board positions can be pushed. The parameter `stackp` is zero if GNU Go is examining the true board position; if it is higher than zero, then GNU Go is examining a hypothetical position obtained by playing several moves.

The most important public reading functions are `attack` and `find_defense`. These are wrappers for functions `do_attack` and `do_find_defense` which are declared statically in '`reading.c`'. The functions `do_attack` and `do_find_defense` call each other recursively.

The return codes of the reading (and owl) functions and owl can be 0, 1, 2 or 3. Each reading function determines whether a particular player (assumed to have the move) can solve a specific problem, typically attacking or defending a string.

The nonzero return codes are called these names in the source:

```
#define WIN  3
#define KO_A 2
#define KO_B 1
```

A return code of WIN means success, 0 failure, while KO_A and KO_B are success conditioned on ko. A function returns KO_A if the position results in ko and that the player to move will get the first ko capture (so the opponent has to make the first ko threat). A return code of KO_B means that the player to move will have to make the first ko threat.

Many of the reading functions make use of *null pointers*. For example, a call to `attack(i, j, &ai, &aj)` will return WIN if the string at `(i, j)` can be captured. The point of attack (in case it is vulnerable) is returned in `(ai, aj)`. However many times we do not care about the point of attack. In this case, we can substitute a null pointer: `attack(i, j, NULL, NULL)`.

Depth of reading is controlled by the parameters `depth` and `branch_depth`. The `depth` has a default value `DEPTH` (in '`liberty.h`'), which is set to 14 in the distribution, but it may also be set at the command line using the '`-D`' or '`--depth`' option. If `depth` is increased, GNU Go will be stronger and slower. GNU Go will read moves past depth, but in doing so it makes simplifying assumptions that can cause it to miss moves.

Specifically, when `stackp > depth`, GNU Go assumes that as soon as the string can get 3 liberties it is alive. This assumption is sufficient for reading ladders.

The `branch_depth` is typically set a little below `depth`. Between `branch_depth` and `depth`, attacks on strings with 3 liberties are considered, but branching is inhibited, so fewer variations are considered.

Currently the reading code does not try to defend a string by attacking a boundary string with more than two liberties. Because of this restriction, it can make oversights. A symptom of this is two adjacent strings, each having three or four liberties, each classified as `DEAD`. To resolve such situations, a function `small_semeai()` (in '`engine/semeai.c`') looks for such pairs of strings and corrects their classification.

The backfill_depth is a similar variable with a default 10. Below this depth, GNU Go will try "backfilling" to capture stones. For example in this situation:

```
.OOOOOO.    on the edge of the board, O can capture X but
OOXXXXXO    in order to do so he has to first play at a in
.aObX.XO    preparation for making the atari at b. This is
--------    called backfilling.
```

Backfilling is only tried with `stackp <= backfill_depth`. The parameter `backfill_depth` may be set using the '`-B`' option.

The `fourlib_depth` is a parameter with a default of only 5. Below this depth, GNU Go will try to attack strings with four liberties. The `fourlib_depth` may be set using the '`-F`' option.

The parameter `ko_depth` is a similar cutoff. If `stackp<ko_depth`, the reading code will make experiments involving taking a ko even if it is not legal to do so (i.e., it is hypothesized that a remote ko threat is made and answered before continuation). This parameter may be set using the '`-K`' option.

A partial list of the functions in '`reading.c`':

- `int attack(int m, int n, int *i, int *j)`:

  The basic function `attack(m, n, *i, *j)` determines if the string at `(m, n)` can be attacked, and if so, `(*i, *j)` returns the attacking move, unless `*i` and `*j` are null pointers. (Use null pointers if you are interested in the result of the attack but not the attacking move itself.) Returns 1 if the attack succeeds, otherwise 0. Returns KO_A or KO_B if the result depends on ko: returns KO_A if the attack succeeds provided attacker is willing to ignore any ko threat. Returns KO_B if attack succeeds provided attacker has a ko threat which must be answered.

- `find_defense(int m, int n, int *i, int *j)`:

  The function `find_defense(m, n, *i, *j)` attempts to find a move that will save the string at `(m,n)`. It returns true if such a move is found, with `(*i, *j)` the location of the saving move (unless `(*i, *j)` are null pointers). It is not checked that tenuki defends, so this may give an erroneous answer if `!attack(m,n)`. Returns KO_A or KO_B if the result depends on ko. Returns KO_A if the string can be defended provided (color) is willing to ignore any ko threat. Returns KO_B if (color) has a ko threat which must be answered.

- `safe_move(int i, int j, int color)` :

  The function `safe_move(i, j, color)` checks whether a move at `(i, j)` is illegal or can immediately be captured. If `stackp==0` the result is cached. If the move only can be captured by a ko, it's considered safe. This may or may not be a good convention.

The next few functions are essentially special cases of `attack` and `find_defense`. They are coded individually, and are static in '`engine/reading.c`'.

- `attack2(int m, int n, int *i, int *j)` :

  Determine whether a string with 2 liberties can be captured. Usage is similar to `attack`.

- `attack3(int m, int n, int *i, int *j)` :

  Determine whether a string with 3 liberties can be captured. Usage is similar to `attack`.

- `attack4(int m, int n, int *i, int *j)` :

  Determine whether a string with 4 liberties can be captured. Usage is similar to `attack`.

- `defend1(int m, int n, int *i, int *j)` :

  Determine whether a string with 1 liberty can be rescued. Usage is similar to `find_defense`.

- `defend2()` :

  Determine whether a string with 2 liberties can be rescued. Usage is similar to `find_defense`.

- `defend3()` :

  Determine whether a string with 3 liberties can be rescued. Usage is similar to `find_defense`.

- `find_cap2()` :

  If `(m,n)` points to a string with 2 liberties, `find_cap2(m,n,&i,&j)` looks for a configuration:

  ```
  O.
  .*
  ```

  where '`O`' is an element of the string in question. It tries the move at '`*`' and returns true this move captures the string, leaving `(i,j)` pointing to *.

- `break_chain(int si, int sj, int *i, int *j, int *k, int *l)`:

  The function `break_chain(si, sj, *i, *j, *k, *l)` returns 1 if part of some surrounding string is in atari, and if capturing this string results in a live string at `(si, sj)`. Returns 2 if the capturing string can be taken (as in a snapback), or the the saving move depends on ignoring a ko threat; Returns 3 if the saving move requires making a ko threat and winning the ko. The pointers (i,j), if not NULL, are left pointing to the appropriate defensive move. The pointers (k,l), if not NULL, are left pointing to the boundary string which is in atari.

- `break_chain2(int si, int sj, int *i, int *j)`:

  The function `break_chain2(si, sj, *i, *j)` returns 1 if there is a string in the surrounding chain having exactly two liberties whose attack leads to the rescue of `(si, sj)`. Then `*i, *j` points to the location of the attacking move. Returns 2 if the attacking stone can be captured, 1 if it cannot.

- `snapback(snapback(int si, int sj, int i, int j, int color)`:

  The function `snapback(si, sj, i, j, color)` considers a move by color at `(i, j)` and returns true if the move is a snapback. Algorithm: It removes dead pieces of the other color, then returns 1 if the stone at `(si, sj)` has <2 liberties. The purpose of this test is to avoid snapbacks. The locations `(i, j)` and `(si,sj)` may be either same or different. Also returns 1 if the move at `(i, j)` is illegal, with the trace message "ko violation" which is the only way I think this could happen. It is not a snapback if the capturing stone can be recaptured on its own, e.g.

  ```
  XXOOOOO
  X*XXXXO
  -------
  ```

  Here 'O' capturing at '*' is in atari, but this is not a snapback. Use with caution: you may want to condition the test on the string being captured not being a singleton. For example

  ```
  XXXOOOOOOOO
  XO*XXXXXXXO
  -----------
  ```

  is rejected as a snapback, yet 'O' captures more than it gives up.

14.2 Hashing of Positions

To speed up the reading process, we note that a position can be reached in several different ways. In fact, it is a very common occurrence that a previously checked position is rechecked, often within the same search but from a different branch in the recursion tree.

This wastes a lot of computing resources, so in a number of places, we store away the current position, the function we are in, and which worm is under attack or to be defended. When the search for this position is finished, we also store away the result of the search and which move made the attack or defense succeed.

All this data is stored in a hash table, sometimes also called a transposition table, where Go positions are the key and results of the reading for certain functions and groups are the data. You can increase the size of the Hash table using the '-M' or '--memory' option see Section 3.9 [Invoking GNU Go], page 12.

The hash table is created once and for all at the beginning of the game by the function `hashtable_new()`. Although hash memory is thus allocated only once in the game, the table is reinitialized at the beginning of each move by a call to `hashtable_clear()` from `genmove()`.

14.2.1 Calculation of the hash value

The hash algorithm is called Zobrist hashing, and is a standard technique for go and chess programming. The algorithm as used by us works as follows:

1. First we define a *go position*. This positions consists of
   - the actual board, i.e. the locations and colors of the stones
   - A *ko point*, if a ko is going on. The ko point is defined as the empty point where the last single stone was situated before it was captured.

   It is not necessary to specify the color to move (white or black) as part of the position. The reason for this is that read results are stored separately for the various reading functions such as `attack3`, and it is implicit in the calling function which player is to move.
2. For each location on the board we generate random numbers:
   - A number which is used if there is a white stone on this location
   - A number which is used if there is a black stone on this location
   - A number which is used if there is a ko on this location

   These random numbers are generated once at initialization time and then used throughout the life time of the hash table.
3. The hash key for a position is the XOR of all the random numbers which are applicable for the position (white stones, black stones, and ko position).

14.2.2 Organization of the hash table

The hash table consists of 3 parts:

- An area which contains so called *Hash Nodes*. Each hash node contains:
  - A go position as defined above.
  - A computed hash value for the position
  - A pointer to Read Results (see below)
  - A pointer to another hash node.
- An area with so called Read Results. These are used to store which function was called in the go position, which string was under attack or to be defended, and the result of the reading.

  Each Read Result contains:
  - the function ID (an int between 0 and 255), the position of the string under attack and a depth value, which is used to determine how deep the search was when it was made, packed into one 32 bit integer.
  - The result of the search (a numeric value) and a position to play to get the result packed into one 32 bit integer.

  - A pointer to another Read Result.
- An array of pointers to hash nodes. This is the hash table proper.

When the hash table is created, these 3 areas are allocated using `malloc()`. When the hash table is populated, all contents are taken from the Hash nodes and the Read results. No further allocation is done and when all nodes or results are used, the hash table is full. Nothing is deleted from the hash table except when it is totally emptied, at which point it can be used again as if newly initialized.

When a function wants to use the hash table, it looks up the current position using `hashtable_search()`. If the position doesn't already exist there, it can be entered using

`hashtable_enter_position()`.

Once the function has a pointer to the hash node containing a function, it can search for a result of a previous search using `hashnode_search()`. If a result is found, it can be used, and if not, a new result can be entered after a search using `hashnode_new_result()`.

Hash nodes which hash to the same position in the hash table (collisions) form a simple linked list. Read results for the same position, created by different functions and different attacked or defended strings also form a linked list.

This is deemed sufficiently efficient for now, but the representation of collisions could be changed in the future. It is also not determined what the optimum sizes for the hash table, the number of positions and the number of results are.

14.2.3 Hash Structures

The basic hash structures are declared in '`hash.h`'.

```
typedef struct hashposition_t {
  Compacttype  board[COMPACT_BOARD_SIZE];
  int          ko_i;
  int          ko_j;
} Hashposition;
```

Represents the board and optionally the location of a ko, which is an illegal move. The player whose move is next is not recorded.

```
typedef struct {
  Hashvalue     hashval;
  Hashposition  hashpos;
} Hash_data;
```

Represents the return value of a function (`hashval`) and the board state (`hashpos`).

```
typedef struct read_result_t {
  unsigned int compressed_data;

  int result_ri_rj;
  struct read_result_t *next;
} Read_result;
```

Here the `compressed_data` field packs into 32 bits the following fields:

```
komaster: 2 bits (EMPTY, BLACK, WHITE, or GRAY)
kom_i   : 5 bits
kom_j   : 5 bits
```

```
routine : 4 bits (currently 10 different choices)
i       : 5 bits
j       : 5 bits
stackp  : 5 bits
```

The `komaster` and `(kom_i,kom_j)` field are documented in See Section 14.3 [Ko], page 131. The integer `result_ri_rj` encodes:

```
unsigned char  status;
unsigned char  result;
unsigned char  ri;
unsigned char  rj;
```

When a new result node is created, 'status' is set to 1 'open'. This is then set to 2 'closed' when the result is entered. The main use for this is to identify open result nodes when the hashtable is partially cleared. Another potential use for this field is to identify repeated positions in the reading, in particular local double or triple kos.

```
typedef struct hashnode_t {
  Hash_data            key;
  Read_result        * results;
  struct hashnode_t  * next;
} Hashnode;
```

The hash table consists of hash nodes. Each hash node consists of The hash value for the position it holds, the position itself and the actual information which is purpose of the table from the start.

There is also a pointer to another hash node which is used when the nodes are sorted into hash buckets (see below).

```
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;
```

The hash table consists of three parts:

- The hash table proper: a number of hash buckets with collisions being handled by a linked list.
- The hash nodes. These are allocated at creation time and are never removed or reallocated in the current implementation.
- The results of the searches. Since many different searches can be done in the same position, there should be more of these than hash nodes.

14.2.4 Hash Functions

The following functions are defined in '`hash.c`':

- `void hash_init()`

  Initialize the entire hash system.

- `int hashdata_compare(Hash_data *key1, Hash_data *key2)`

  Returns 0 if `*key1 == *key2`, 2 if the hashvalues differ, or 1 if only the hashpositions differ. This adheres (almost) to the standard compare function semantics which are used e.g. by the comparison functions used in qsort().

- `void hashposition_dump(Hashposition *pos, FILE *outfile)`

  Dump an ASCII representation of the contents of a Hashposition onto the FILE outfile.

- `int hashdata_diff_dump(Hash_data *key1,Hash_data *key2 )`

  Compare two Hashdata structs. If equal: return zero. If not: dump a human readable summary of any differences to stderr. The return value is the same as for hashdata_compare. This function is primarily intended to be used in assert statements.

- `void hashdata_recalc(Hash_data *target, Intersection board[MAX_BOARD][MAX_BOARD], int koi, int koj)`

  Calculate the compactboard and the hashvalue in one function. They are always used together and it saves us a loop and a function call.

- `void hashdata_set_ko(Hash_data *hd, int i, int j)`

  Set or remove a ko at (i, j).

- `void hashdata_remove_ko(Hash_data *hd)`

  Remove any ko from the hash value and hash position.

- `void hashdata_invert_stone(Hash_data *hd, int i, int j, int color)`

  Set or remove a stone of COLOR at (I, J) in a Hash_data.

- `void read_result_dump(Read_result *result, FILE *outfile)`

  Dump an ASCII representation of the contents of a Read_result onto the FILE outfile.

- `void hashnode_dump(Hashnode *node, FILE *outfile)`

  Dump an ASCII representation of the contents of a Hashnode onto the FILE outfile.

- `int hashtable_init(Hashtable *table, int tablesize, int num_nodes, int num_results)`

  Initialize a hash table for a given total size and size of the hash table. Returns 0 if something went wrong. Just now this means that there wasn't enough memory available.

- `Hashtable * hashtable_new(int tablesize, int num_nodes, int num_results)`

  Allocate a new hash table and return a pointer to it. Return NULL if there is insufficient memory.

- `void hashtable_clear(Hashtable *table)`

  Clear an existing hash table.
- `void hashtable_clear_if_full(Hashtable *table)`

  Clear an existing hash table only if it happens to be full. By full we mean that we are either out of positions or read results.
- `Hashnode * hashtable_enter_position(Hashtable *table, Hash_data *hd)`

  Enter a position with a given hash value into the table. Return a pointer to the hash node where it was stored. If it is already there, don't enter it again, but return a pointer to the old one.
- `Hashnode * hashtable_search(Hashtable *table, Hash_data *hd)`

  Given a Hashposition and a Hash value, find the hashnode which contains this position with the given hash value.
- `void hashtable_dump(Hashtable *table, FILE *outfile)`

  Dump an ASCII representation of the contents of a Hashtable onto the FILE outfile.
- `Read_result * hashnode_search(Hashnode *node, int routine, int i, int j)`

  Search the result list in a hash node for a particular result. This result is from `routine` (the calling function) at `(i, j)` and reading depth stackp. All these numbers must be unsigned, and 0<= x <= 255).
- `Read_result * hashnode_new_result(Hashtable *table, Hashnode *node, int routine, int i, int j)`

  Enter a new Read_result into a Hashnode. We already have the node, now we just want to enter the result itself. We will fill in the result itself later, so we only need the routine number for now.

The following macros are defined in '`hash.h`'

- `rr_get_routine(Read_result rr)`
- `rr_get_pos_i(Read_result rr)`
- `rr_get_pos_j(Read_result rr)`
- `rr_get_stackp(Read_result rr)`

  Get the constituent parts of a `Read_result`.

The following macros and functions are defined in '`engine/reading.c`':

- `static int get_read_result(int routine, int *si, int *sj, Read_result **read_result)`

  Return a Read_result for the current position, routine and location. For performance, the location is changed to the origin of the string.
- `READ_RETURN0(Read_result *read_result)`

  Cache a negative read result.
- `READ_RETURN(Read_result *read_result, int *pointi, int *pointj, int resulti, int resultj, int value)`

  If `pointi` and `pointj` are not null pointers, then give `(*pointi, *pointj)` the values `(resulti, resultj)`. Then cache the `read_result`. Clear the hashtable if full and return `value`.

14.2.5 Persistent Reading Cache

Some reading calculations can be safely saved from move to move.

The function `store_persistent_cache()` is called only by `attack` and `find_defense`, never from their static recursive counterparts `do_attack` and `do_defend`. The function `store_persistent_reading_cache()` attempts to cache the most expensive reading results. The function `search_persistent_reading_cache` attempts to retrieve a result from the cache.

If all cache entries are occupied, we try to replace the least useful one. This is indicated by the score field, which is initially the number of nodes expended by this particular reading, and later multiplied by the number of times it has been retrieved from the cache.

Once a (permanent) move is made, a number of cache entries immediately become invalid. These are cleaned away by the function `purge_persistent_reading_cache().` To have a criterion for when a result may be purged, the function `store_persistent_cache()` computes the *reading shadow* and *active area*. If a permanent move is subsequently played in the active area, the cached result is invalidated. We now explain this algorithm in detail.

The *reading shadow* is the concatenation of all moves in all variations, as well as locations where an illegal move has been tried.

Once the read is finished, the reading shadow is expanded to the *active area* which may be cached. The intention is that as long as no stones are played in the active area, the cached value may safely be used.

Here is the algorithm used to compute the active area. This algorithm is in the function `store_persistent_reading_cache()`. The most expensive readings so far are stored in the persistent cache.

- The reading shadow and the string under attack are marked with the character '`1`'. We also include the successful move, which is most often a part of the reading shadow, but sometimes not, for example with the function `attack1()`.
- Next the reading shadow is expanded by marking strings and empty vertices adjacent to the area marked '`1`' with the character '`2`'.
- Next vertices adjacent to empty vertices marked '`2`' are labelled with the character '`3`'.
- Next all vertices adjacent to previously marked vertices. These are marked '`-1`' instead of the more logical '`4`' because it is slightly faster to code this way.
- If the stack pointer is `>`0 we add the moves already played from the moves stack with mark 4.

14.3 Ko Handling

The principles of ko handling are the same for tactical reading and owl reading.

We have already mentioned (see Section 14.1 [Reading Basics], page 122) that GNU Go uses a return code of KO_A or KO_B if the result depends on ko. The return code of KO_B means that the position can be won provided the player whose move calls the function can come up with a sufficiently large ko threat. In order to verify this, the function must simulate making a ko threat and having it answered by taking the ko even if it is illegal. We call such an experimental taking of the ko a *conditional* ko capture.

Conditional ko captures are accomplished by the function `tryko()`. This function is like `trymove()` except that it does not require legality of the move in question.

The static reading functions, and the global functions `do_attack` and `do_find_defense` have arguments `komaster`, `kom_i` and `kom_j`. These mediate ko captures to prevent the occurrence of infinite loops.

Normally `komaster` is EMPTY but it can also be BLACK, WHITE or GRAY. The komaster is set to COLOR when COLOR makes a conditional ko capture. In this case `kom_i, kom_j` is set to the location of the captured ko stone.

If the opponent is komaster, the reading functions will not try to take the ko at `kom_i, kom_j`. Also, the komaster is normally not allowed to take another ko. The exception is a nested ko, characterized by the condition that the captured ko stone is at distance 1 both vertically and horizontally from `(kom_i, kom_j)`, which is the location of the last stone taken by the komaster. Thus in this situation:

```
             .OX
             OX*X
            OmOX
             OO
```

Here if '`m`' is the location of `(kom_i,kom_j)`, then the move at '`*`' is allowed.

The rationale behind this rule is that in the case where there are two kos on the board, the komaster cannot win both, and by becoming komaster he has already chosen which ko he wants to win. But in the case of a nested ko, taking one ko is a precondition to taking the other one, so we allow this.

If the komaster's opponent takes a ko, then both players have taken one ko. In this case 'komaster' is set to GRAY and after this further ko captures are not allowed.

If the ko at `(kom_i, kom_j)` is filled, then the komaster reverts to EMPTY.

The komaster scheme may be summarized as follows. It is assumed that '`O`' is about to move.

- 1. Komaster is EMPTY.
  - Unconditional ko capture is allowed. Komaster remains EMPTY.
  - Conditional ko capture is allowed. Komaster is set to '`O`' and `(kom_i, kom_j)` to the location of the ko, where a stone was just removed.
- Komaster is O:
  - Only nested ko captures are allowed.
  - If komaster fill the ko at `(kom_i,kom_j)` then komaster reverts to EMPTY.
- Komaster is X:
  - Play at `(kom_i,kom_j)` is not allowed. Any other ko capture is allowed. If '`O`' takes another ko, komaster becomes GRAY.
- Komaster is GRAY:
  - Ko captures are not allowed. If the ko at `(kom_i,kom_j)` is filled then the komaster reverts to EMPTY.

14.4 A Ko Example

To see the komaster scheme in action, consider this position from the file ‘`regressions/games/life_and_death/tripod9.sgf`’. We recommend studying this example by examining the variation file produced by the command:

```
gnugo -l tripod9.sgf --decidedragon C3 -o vars.sgf
```

In the lower left hand corner, there are kos at A2 and B4. Black is unconditionally dead because if W wins either ko there is nothing B can do.

```
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
```

This is how the komaster scheme sees this. B (i.e. X) starts by taking the ko at B4. W replies by taking the ko at A1. The board looks like this:

```
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
```

Now any move except the ko recapture (currently illegal) at A1 loses for B, so B retakes the ko and becomes komaster. The board looks like this:

```
8 . . . . . . . .          komaster: BLACK
7 . . O . . . . .          (kom_i, kom_j): 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
```

W takes the ko at B3 after which the komaster is GRAY and ko recaptures are not allowed.

```
8 . . . . . . . .          komaster: GRAY
7 . . O . . . . .          (kom_i, kom_j): 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
```

Since X is not allowed any ko recaptures, there is nothing he can do and he is found dead. Thus the komaster scheme produces the correct result.

We now consider an example to show why the komaster is reset to EMPTY if the ko is resolved in the komaster's favor. This means that the ko is filled, or else that is becomes no longer a ko and it is illegal for the komaster's opponent to play there.

The position resulting under consideration is in the file '`regressions/games/ko5.sgf`'. This is the position:

```
. . . . . . 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
```

We recommend studying this example by examining the variation file produced by the command:

```
gnugo -l ko5.sgf --quiet --decidestring L1 -o vars.sgf
```

The correct resolution is that H1 attacks L1 while K2 defends it with ko (code KO_A).

After Black (X) takes the ko at K3, white can do nothing but retake the ko conditionally, becoming komaster. B cannot do much, but in one variation he plays at K4 and W takes at H1. The following position results:

```
. . . . . . 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
```

Now it is important the '`O`' is no longer komaster. Were '`O`' still komaster, he could capture the ko at N3 and there would be no way to finish off B.

The following alternate schemes have been proposed. It is assumed that '`O`' is the player about to move.

14.4.1 Essentially the 2.7.232 scheme.

- Komaster is EMPTY.
  - Unconditional ko capture is allowed. Komaster remains EMPTY.
  - Conditional ko capture is allowed. Komaster is set to O and `(kom_i, kom_j)` to the location of the ko, where a stone was just removed.
- Komaster is O:
  - Conditional ko capture is not allowed.
  - Unconditional ko capture is allowed. Komaster parameters unchanged.
- Komaster is X:
  - Conditional ko capture is not allowed.
  - Unconditional ko capture is allowed except for a move at `(kom_i, kom_j)`. Komaster parameters unchanged.

14.4.2 Revised 2.7.232 version

- Komaster is EMPTY.
  - Unconditional ko capture is allowed. Komaster remains EMPTY.
  - Conditional ko capture is allowed. Komaster is set to '`O`' and `(kom_i, kom_j)` to the location of the ko, where a stone was just removed.
- Komaster is '`O`':
  - Ko capture (both kinds) is allowed only if after playing the move, `is_ko(kom_i, kom_j, X)` returns false. In that case, `(kom_i, kom_j)` is updated to the new ko position, i.e. the stone captured by this move.
- Komaster is X:
  - Conditional ko capture is not allowed.
  - Unconditional ko capture is allowed except for a move at `(kom_i, kom_j)`. Komaster parameters unchanged.

14.5 Superstrings

A *superstring* is an extended string, where the extensions are through the following kinds of connections:

1. Solid connections (just like ordinary string).
   ```
   OO
   ```
2. Diagonal connection or one space jump through an intersection where an opponent move would be suicide or self-atari.
   ```
   ...
   O.O
   XOX
   X.X
   ```
3. Bamboo joint.

```
OO
..
OO
```

4. Diagonal connection where both adjacent intersections are empty.

```
.O
O.
```

5. Connection through adjacent or diagonal tactically captured stones. Connections of this type are omitted when the superstring code is called from '`reading.c`', but included when the superstring code is called from '`owl.c`'.

Like a dragon, a superstring is an amalgamation of strings, but it is a much tighter organization of stones than a dragon, and its purpose is different. Superstrings are encountered already in the tactical reading because sometimes attacking or defending an element of the superstring is the best way to attack or defend a string. This is in contrast with dragons, which are ignored during tactical reading.

14.6 Reading Functions

Here we list the publically callable functions in '`reading.c`'. The return codes of these functions are explained elsewhere (see Section 14.1 [Reading Basics], page 122).

- `attack(si, sj, *i, *j)`

  Determines if the string at (m, n) can be captured, and if so, (*i, *j) returns the attacking move, unless (*i, *j) are null pointers. Use null pointers if you are interested in the result of the attack but not the attacking move itself. The string is assumed to be alive if it can get five liberties—fewer if `stackp` is large.

  - Returns 1 if the attack succeeds unconditionally
  - Returns 0 if the attack fails unconditionally
  - Returns 2 if the attack succeeds provided attacker is willing to ignore any ko threat (the attacker makes the first ko capture).
  - Returns 3 if attack succeeds provided attacker has a ko threat which must be answered (the defender makes the first ko capture).

- `find_defense(m, n, *i, *j)`

  Attempts to find a move that will save the string at `(m, n)`. It returns 1 if such a move is found, with `(*i, *j)` the location of the saving move, unless `(*i, *j)` are null pointers. It is not checked that tenuki defends, so this may give an erroneous answer if `!attack(m,n)`. Returns 2 or 3 if the result depends on ko. Returns 2 if the string can be defended provided the defender is willing to ignore any ko threat. Returns 3 if the defender wins by having a ko threat which must be answered.

- `int attack_and_defend(int si, int sj, int *attack_code, int *attacki, int *attackj,int *defend_code, int *defendi, int *defendj)`

  This is a frontend to the `attack()` and `find_defense()` which guarantees a consistent result. If a string cannot be attacked, 0 is returned and acode is 0. If a string can be attacked and defended, WIN is returned, acode and dcode are both non-zero, and `(ai, aj)`, `(di, dj)` both point to vertices on the board. If a string can be attacked but not defended, 0 is again returned, acode is non-zero, dcode is 0, and (ai, aj) point to a vertex on the board. This function in particular guarantees that if there is an attack, it will never return `(di, dj) = (-1, -1)`, which means the string is safe without defense. Separate calls to `attack()` and `find_defense()` may occasionally give this result, due to irregularities introduced by the persistent reading cache.

- `attack_either(ai, aj, bi, bj)`

  returns true if there is a move which guarantees that at least one of the strings `(ai, aj)` and `(bi, bj)` can be captured. A typical application for this is in connection patterns, where after a cut it suffices to capture one of the cutting stones. The current implementation looks only for uncoordinated attacks and is not even sufficient to find a double atari.

- `defend_both(ai, aj, bi, bj)`

  Returns true if both the strings `(ai, aj)` and `(bi, bj)` can be defended simultaneously or if there is no attack. A typical application for this is in connection patterns, where after a cut it's necessary to defend both cutting stones.

- `int break_through(int ai, int aj, int bi, int bj, int ci, int cj)`

  Returns 1 if a position can succesfully be broken through and 2 if it can be cut. The position is assumed to have the shape (the colors may be reversed)

  ```
  .O.      dbe
  OXO      aFc
  ```

  It is '`X`' to move and try to capture at least one of '`a`', '`b`', and '`c`'. If this succeeds, '`X`' is said to have broken through the position. Otherwise '`X`' may try to cut through the position, which means keeping '`F`' safe and getting a tactically safe string at either '`d`' or '`e`'. **Important**: '`a`', '`b`', and '`c`' must be given in the correct order.

- `int atari_atari(int color, int *i, int *j, int save_verbose)`

  Looks for a series of ataris on strings of the other color culminating in the capture of a string which is thought to be invulnerable by the reading code. Such a move can be missed since it may be that each string involved individually can be rescued, but nevertheless one of them can be caught. The simplest example is a double atari. The return value is the size of the smallest opponent worm. A danger with this scheme is that the first atari tried might be irrelevant to the actual combination. To avoid this, once we've found a combination, we mark the first move as forbidden, then try again. If no combination of the same size or larger turns up, then the first move was indeed essential. Returns the size of the smallest of the worms under attack.

- `int atari_atari_confirm_safety(int color, int ti, int tj, int *i, int *j, int minsize)`

  Uses the `atari_atari` code to detect blunders. Ask whether there appears any combination attack which would capture at least minsize stones after playing at `(ti, tj)`. If this happens, `(*i, *j)` points to a defensive move which prevents this blunder.

- `int atari_atari_try_combination(int color, int ai, int aj, int bi, int bj)`

  Ask the atari_atari code if after color plays at (ai,aj) and other plays at (bi,bj) there appears any combination attack. Returns the size of the combination.

14.7 Debugging the reading code

The reading code searches for a path through the move tree to determine whether a string can be captured. We have a tool for investigating this with the '`--decidestring`' option. This may be run with or without an output file.

Simply running

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

will run `attack()` to determine whether the string can be captured. If it can, it will also run `find_defense()` to determine whether or not it can be defended. It will give a count of the number of variations read. The '`-t`' is necessary, or else GNU Go will not report its findings.

If we add '`-o` *output file*' GNU Go will produce an output file with all variations considered. The variations are numbered in comments.

This file of variations is not very useful without a way of navigating the source code. This is provided with the GDB source file, listed at the end. You can source this from GDB, or just make it your GDB init file.

If you are using GDB to debug GNU Go you may find it less confusing to compile without optimization. The optimization sometimes changes the order in which program steps are executed. For example, to compile '`reading.c`' without optimization, edit '`engine/Makefile`' to remove the string `-O2` from the file, touch '`engine/reading.c`' and make. Note that the Makefile is automatically generated and may get overwritten later.

If in the course of reading you need to analyze a result where a function gets its value by returning a cached position from the hashing code, rerun the example with the hashing turned off by the command line option '`--hash 0`'. You should get the same result. (If you do not, please send us a bug report.) Don't run '`--hash 0`' unless you have a good reason to, since it increases the number of variations.

With the source file given at the end of this document loaded, we can now navigate the variations. It is a good idea to use cgoban with a small '`-fontHeight`', so that the variation window takes in a big picture. (You can resize the board.)

Suppose after perusing this file, we find that variation 17 is interesting and we would like to find out exactly what is going on here.

The macro 'jt n' will jump to the n-th variation.

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

```
(gdb) run
(gdb) jt 17
```

will then jump to the location in question.

Actually the attack variations and defense variations are numbered separately. (But `find_defense()` is only run if `attack()` succeeds, so the defense variations may or may not exist.) It is redundant to have to tbreak main each time. So there are two macros avar and dvar.

```
(gdb) avar 17
```

restarts the program, and jumps to the 17-th attack variation.

```
(gdb) dvar 17
```

jumps to the 17-th defense variation. Both variation sets are found in the same sgf file, though they are numbered separately.

Other commands defined in this file:

```

dump will print the move stack.
nv moves to the next variation
ascii i j converts (i,j) to 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
```

15 Life and Death Reading

GNU Go does two very different types of life and death reading. First, there is the OWL code (Optics with Limit Negotiation) which attempts to read out to a point where the code in ‘engine/optics.c’ (see Chapter 11 [Eyes], page 82) can be used to evaluate it.

Secondly, there is the code in ‘engine/life.c’ which is a potential replacement for the code in ‘optics.c’. It attempts to evaluate eyespaces more accurately than the code in ‘optics.c’, but since it is fairly slow, it is partially disabled unless you run GNU Go with the option ‘--life’. The default use of the life code is that it can be called from ‘optics.c’ when the graph based life and death code concludes that it needs an expert opinion.

Like the tactical reading code, a persistent cache is employed to maintain some of the owl data from move to move. This is an essential speedup without which GNU Go would play too slowly.

15.1 The Owl Code

The life and death code in ‘optics.c’, described elsewhere (see Chapter 11 [Eyes], page 82), works reasonably well as long as the position is in a *terminal position*, which we define to be one where there are no moves left which can expand the eye space, or limit it. In situations where the dragon is surrounded, yet has room to thrash around a bit making eyes, a simple application of the graph-based analysis will not work. Instead, a bit of reading is needed to reach a terminal position.

The defender tries to expand his eyespace, the attacker to limit it, and when neither finds an effective move, the position is evaluated. We call this type of life and death reading *Optics With Limit-negotiation* (OWL). The module which implements it is in ‘engine/owl.c’.

There are two reasonably small databases ‘patterns/owl_defendpats.db’ and ‘patterns/owl_attackpats.db’ of expanding and limiting moves. The code in ‘owl.c’ generates a small move tree, allowing the attacker only moves from ‘owl_attackpats.db’, and the defender only moves from ‘owl_defendpats.db’. In addition to the moves suggested by patterns, vital moves from the eye space analysis are also tested.

A third database, ‘owl_vital_apats.db’ includes patterns which override the eyespace analysis done by the optics code. Since the eyeshape graphs ignore the complications of shortage of liberties and cutting points in the surrounding chains, the static analysis of eyespace is sometimes wrong. The problem is when the optics code says that a dragon definitely has 2 eyes, but it isn’t true due to shortage of liberties, so the ordinary owl patterns never get into play. In such situations owl_vital_apats.db is the only available measure to correct mistakes by the optics. Currently the patterns in ‘owl_vital_apats.db’ are only matched when the level is 9 or greater.

The owl code is tuned by editing these three pattern databases, principally the first two.

A node of the move tree is considered terminal if no further moves are found from ‘apats.db’ or ‘dpats.db’, or if the function compute_eyes_pessimistic() reports that the group is definitely alive or dead. At this point, the status of the group is evaluated. The functions owl_attack() and owl_defend(), with usage similar to attack() and find_defense(), make use of the owl pattern databases to generate the move tree and decide the status of the group.

The function `compute_eyes_pessimistic()` used by the owl code is very conservative and only feels certain about eyes if the eyespace is completely closed (i.e. no marginal vertices).

The maximum number of moves tried at each node is limited by the parameter `MAX_MOVES` defined at the beginning of '`engine/owl.c`'. The most most valuable moves are tried first, with the following restrictions:

- If `stackp > owl_branch_depth` then only one move is tried per variation.
- If `stackp > owl_reading_depth` then the reading terminates, and the situation is declared a win for the defender (since deep reading may be a sign of escape).
- If the node count exceeds `owl_node_limit`, the reading also terminates with a win for the defender.
- Any pattern with value 99 is considered a forced move: no other move is tried, and if two such moves are found, the function returns false. This is only relevant for the attacker.
- Any pattern in '`patterns/owl_attackpats.db`' and '`patterns/owl_defendpats.db`' with value 100 is considered a win: if such a pattern is found by `owl_attack` or `owl_defend`, the function returns true. This feature must be used most carefully.

The functions `owl_attack()` and `owl_defend()` may, like `attack()` and `find_defense()`, return an attacking or defending move through their pointer arguments. If the position is already won, `owl_attack()` may or may not return an attacking move. If it finds no move of interest, it will return `PASS`, that is, `(-1,-1)`. The same goes for `owl_defend()`.

When `owl_attack()` or `owl_defend()` is called, the dragon under attack is marked in the array `goal`. The stones of the dragon originally on the board are marked with goal=1; those added by `owl_defend()` are marked with goal=2. If all the original strings of the original dragon are captured, `owl_attack()` considers the dragon to be defeated, even if some stones added later can make a live group.

Only dragons with small escape route are studied when the functions are called from `make_dragons()`.

The owl code can be conveniently tested using the '`--decidedragon` *location*' This should be used with '`-t`' to produce a useful trace, '`-o`' to produce an SGF file of variations produced when the life and death of the dragon at *location* is checked, or both. '`--decideposition`' performs the same analysis for all dragons with small escape route.

15.2 Functions in '`owl.c`'

In this section we list the non-static functions in '`owl.c`'. Note that calls to `owl_attack` and `owl_defend` should be made only when `stackp==0`. If you want to set up a position, then use the owl code to analyze it, you may call `do_owl_attack` and `do_owl_defend` with `stackp>0` but first you must set up the goal and boundary arrays. See `owl_does_defend` and `owl_substantial` for examples.

The reason that we do not try to write a general `owl_attack` which works when `stackp>0` is that we make use of cached information in the calls to `same_dragon` from the (static) function `owl_mark_dragon`. This requires the dragon data to be current, which it is not when `stackp>0`.

- `int owl_attack(int m, int n, int *ui, int *uj)`

  Returns 1 if a move can be found to attack the dragon at `(m, n)`, in which case `(*ui, *uj)` is the recommended move. `(*ui, *uj)` can be null pointers if the result is not needed.

  - Returns 2 if the attack prevails provided attacker is willing to ignore any ko threat (the attacker makes the first ko capture).
  - Returns 3 if attack succeeds provided attacker has a ko threat which must be answered (the defender makes the first ko capture).

- `int owl_threaten_attack(int m, int n, int *ui, int *uj, int *vi, int *vj)`

  Returns 1 if the dragon at `(m, n)` can be captured given two moves in a row. The first two moves to capture the dragon are given as `(*ui, *uj)` and `(*vi, *vj)`.

- `int owl_defend(int m, int n, int *ui, int *uj)`

  Returns 1 if a move can be found to defend the dragon at `(m, n)`, in which case `(*ui, *uj)` is the recommended move. `(*ui, *uj)` can be null pointers if the result is not needed.

  - Returns 2 if the defense prevails provided defender is willing to ignore any ko threat (the defender makes the first ko capture).
  - Returns 3 if defense succeeds provided defender has a ko threat which must be answered (the attacker makes the first ko capture).

- `int owl_threaten_defense(int m, int n, int *ui, int *uj, int *vi, int *vj)`

  Returns true if the dragon at `(m, n)` can be defended given two moves in a row. The first two moves to defend the dragon are given as `(*ui, *uj)` and `(*vi, *vj)`.

- `void goaldump(char goal[MAX_BOARD][MAX_BOARD])` quotation Lists the goal array. For use in GDB:

  ```
  (gdb) set goaldump(goal)
  ```

- `void owl_reasons(int color)`

  Add owl reasons. This function should be called once during genmove.

- `owl_does_defend(int ti, int tj, int m, int n)`

  Use the owl code to determine whether the move at `(ti, tj)` makes the dragon at `(m, n)` owl safe. This is used to test whether tactical defenses are strategically viable, whether a strategical defense move is effective, and whether a vital eye point does save an owl critical dragon.

- `owl_does_attack(int ti, int tj, int m, int n)`

  Use the owl code to determine whether the move at `(ti, tj)` owl kills the dragon at `(m, n)`. This is used to test whether strategical attack moves are dangerous enough to kill and whether a vital eye point does kill an owl critical dragon.

- `int owl_connection_defends(int ti, int tj, int ai, int aj, int bi, int bj)`

  Use the owl code to determine whether connecting the two dragons `(ai, aj)` and `(bi, bj)` by playing at `(ti, tj)` results in a living dragon. Should be called only when `stackp==0`.

- `int owl_lively(int i, int j)`

  True unless `(i, j)` is `EMPTY` or occupied by a lunch for the goal dragon. Used during `make_domains()` (see `optics.c`: lively macro).
- `int owl_substantial(int i, int j)`

  This function, called when `stackp==0`, returns true if capturing the string at `(i,j)` results in a live group.
- `int vital_chain(int m, int n)`

  This function returns true if it is judged that the capture of the string at `(m,n)` is sufficient to create one eye or to escape.

16 Influence Function

16.1 Conceptual Outline of Influence

We define *lively* stones to be all stones that can't be tactically attacked or have a tactical defense. Stones that have been found to be strategically dead are called dead while all other stones are called *alive*. If we want to use the influence function before deciding the strategical status, all lively stones count as alive.

Every alive stone on the board works as an influence source, with influence of its color radiating outwards in all directions. The strength of the influence declines exponentially with the distance from the source.

Influence can only flow unhindered if the board is empty, however. All lively stones (regardless of color) act as influence barriers, as do connections between enemy stones that can't be broken through. For example the one space jump counts as a barrier unless either of the stones can be captured. Notice that it doesn't matter much if the connection between the two stones can be broken, since in that case there would come influence from both directions anyway.

We define *territory* to be the intersections where one color has no influence at all and the other player does have. We can introduce moyo and area concepts similar to those provided by the Bouzy algorithms in terms of the influence values for the two colors. "Territory" refers to certain or probable territory while "Moyo" refers to an area of dominant influence which is not necessarily guaranteed territory. "Area" refers to the breathing space around a group in which it can manoeever if it is attacked.

In order to avoid finding bogus territory, we add extra influence sources at places where an invasion can be launched, e.g. at 3-3 under a handicap stone, in the middle of wide edge extensions and in the center of large open spaces anywhere. Similarly we add extra influence sources where intrusions can be made into what otherwise looks as solid territory, e.g. monkey jumps.

Walls typically radiate an influence that is stronger than the sum of the influence from the stones building the wall. To accommodate for this phenomenon, we also add extra influence sources in empty space at certain distances away from walls.

16.2 The Core of the Influence Function

The basic influence radiation process can efficiently be implemented as a breadth first search for adjacent and more distant points, using a queue structure.

Influence barriers can be found by pattern matching, assisted by reading through constraints and/or helpers. Wall structures, invasion points and intrusion points can be found by pattern matching as well.

When influence is computed, the basic idea is that there are a number of influence sources on the board, whose contributions are summed to produce the influence values. For the time being we can assume that the living stones on the board are the influence sources, although this is not the whole story.

The function `compute_influence()` contains a loop over the board, and for each influence source on the board, the function `accumulate_influence()` is called. This is the core of the influence function. Before we get into the details, this is how the influence field from a single isolated influence source of strength 100 turns out:

```
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
```

These values are in reality floating point numbers but have been rounded down to the nearest integer for presentation. This means that the influence field does not stop when the numbers become zeroes.

Internally `accumulate_influence()` starts at the influence source and spreads influence outwards by means of a breadth first propagation, implemented in the form of a queue. The order of propagation and the condition that influence only is spread outwards guarantee that no intersection is visited more than once and that the process terminates. In the example above, the intersections are visited in the following order:

```
+  +  +  +  +  +  +  +  +  +  +
+ 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  +
+  +  +  +  +  +  +  +  +  +  +
```

The visitation of intersections continues in the same way on the intersections marked ''+' and further outwards. In a real position there will be stones and tight connections stopping the influence from spreading to certain intersections. This will disrupt the diagram above, but the main property of the propagation still remains, i.e. no intersection is visited more than once and after being visited no more influence will be propagated to the intersection.

16.3 The Core of the Influence Function

Let `(m, n)` be the coordinates of the influence source and `(i, j)` the coordinates of a an intersection being visited during propagation, using the same notation as in the `accumulate_influence()` function. Influence is now propagated to its eight closest neighbors, including the diagonal ones, according to the follow scheme:

For each of the eight directions `(di, dj)`, do:

1. Compute the scalar product `di*(i-m) + dj*(j-n)` between the vectors `(di,dj)` and `(i,j) - (m,n)`
2. If this is negative or zero, the direction is not outwards and we continue with the next direction. The exception is when we are visiting the influence source, i.e. the first intersection, when we spread influence in all directions anyway.
3. If `(i+di, j+dj)` is outside the board or occupied we also continue with the next direction.
4. Let S be the strength of the influence at `(i, j)`. The influence propagated to `(i+di, j+dj)` from this intersection is given by `P*(1/A)*D*S`, where the three different kinds of damping are:
   - The permeability '`P`', which is a property of the board intersections. Normally this is one, i.e. unrestricted propagation, but to stop propagation through e.g. one step jumps, the permeability is set to zero at such intersections through pattern matching. This is further discussed below.
   - The attenuation '`A`', which is a property of the influence source and different in different directions. By default this has the value 3 except diagonally where the number is twice as much. By modifying the attenuation value it is possible to obtain influence sources with a larger or a smaller effective range.
   - The directional damping '`D`', which is the squared cosine of the angle between `(di,dj)` and `(i,j) - (m,n)`. The idea is to stop influence from "bending" around an interfering stone and get a continuous behavior at the right angle cutoff. The choice of the squared cosine for this purpose is rather arbitrary, but has the advantage that it can be expressed as a rational function of '`m`', '`n`', '`i`', '`j`', '`di`', and '`dj`', without involving any trigonometric or square root computations. When we are visiting the influence source we let by convention this factor be one.

Influence is typically contributed from up to three neighbors "between" this intersection and the influence source. These values are simply added together. As pointed out before, all contributions will automatically have been made before the intersection itself is visited.

When the total influence for the whole board is computed by `compute_influence()`, `accumulate_influence()` is called once for each influence source. These invocations are totally independent and the influence contributions from the different sources are added together.

16.4 Permeability

The permeability at the different points is initially one at all empty intersections and zero at occupied intersections. To get a useful influence function we need to modify this, however. Consider the following position:

```
|......
|OOOO..
|...O..
|...a.X   ('a' empty intersection)
|...O..
|...OOO
|.....O
```

```
  +------
```

The corner is of course secure territory for ‘O’ and clearly the ‘X’ stone has negligible effect inside this position. To stop ‘X’ influence from leaking into the corner we use pattern matching (pattern Barrier1/Barrier2 in ‘barriers.db’) to modify the permeability for ‘X’ at this intersection to zero. ‘O’ can still spread influence through this connection.

Another case that needs to be mentioned is how the permeability damping is computed for diagonal influence radiation. For horizontal and vertical radiation we just use the permeability (for the relevant color) at the intersection we are radiating from. In the diagonal case we additionally multiply with the maximum permeability at the two intersections we are trying to squeeze between. The reason for this can be found in the diagram below:

```
  |...X     |...X
  |OO..     |Oda.
  |..O.     |.bc.
  |..O.     |..O.
  +----     +----
```

We don’t want ‘X’ influence to be spread from ‘a’ to ‘b’, and since the permeability at both c and d is zero, the rule above stops this.

16.5 Escape

One application of the influence code is in computing the dragon.escape_route field. This is computed by the function compute_escape() as follows. First, every intersection is assigned an escape value, ranging between 0 and 4, depending on the influence value of the opposite color.

In addition to assiging an escape value to empty vertices, we also assign an escape value to friendly dragons. This value can range from 0 to 6 depending on the status of the dragon, with live dragons having value 6.

Then we sum the values of the resulting influence escape values over the intersections (including friendly dragons) at distance 4, that is, over those intersections which can be joined to the dragon by a path of length 4 (and no shorter path) not passing adjacent to any unfriendly dragon. In the following example, we sum the influence escape value over the four vertices labelled ’4’.

```
      . . . . . . . . .    . . . . . . . . .
      . . . . . X . . O    . . . . . X . . O
      . . X . . . . . O    . . X . 2 . 4 . O
      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 . . .
      . . . . . . . . .    . . . . . . . . .
```

Since the dragon is trying to reach safety, the reader might wonder why compute_influence() is called with the opposite color of the dragon contemplating escape. To explain this point, we first remind the reader why there is a color parameter to `compute_influence()`. Consider the following example position:

```
...XX...
OOO..OOO
O......O
O......O
--------
```

Whether the bottom will become O territory depends on who is in turn to play. This is implemented with the help of patterns in barriers.db, so that X influence is allowed to leak into the bottom if X is in turn to move but not if O is. There are also “invade” patterns which add influence sources in sufficiently open parts of the board which are handled differently depending on who is in turn to move.

In order to decide the territorial value of an O move in the third line gap above, influence is first computed in the original position with the opponent (i.e. X) in turn to move. Then the O stone is played to give:

```
...XX...
OOO.OOOO
O......O
O......O
--------
```

Now influence is computed once more, also this time with X in turn to move. The difference in territory (as computed from the influence values) gives the territorial value of the move.

Exactly how influence is computed for use in the escape route estimation is all ad hoc. But it makes sense to assume the opponent color in turn to move so that the escape possibilities aren’t overestimated. After we have made a move in the escape direction it is after all the opponent’s turn.

The current escape route mechanism seems good enough to be useful but is not completely reliable. Mostly it seems to err on the side of being too optimistic.

16.6 Influential Functions

- `static void accumulate_influence(struct influence_data *q, int m, int n, int color)`

  Limited in scope to ‘`influence.c`’, this is the core of the influence function. Given the coordinates and color of an influence source, it radiates the influence outwards until it hits a barrier or the strength of the influence falls under a certain threshold. The radiation is performed by a breadth first propagation, implemented by means of an internal queue.

- `void compute_initial_influence(int color, int dragons_known)`

  Compute the influence before a move has been made, which can later be compared to the influence after a move. Assume that the other color is in turn to move.
- `static void compute_move_influence(int m, int n, int color)`

  Let color play at (m, n) and compute the influence after this move, assuming that the other color is in turn to move next.
- `int influence_territory_color(int m, int n)`

  Return the color who has territory at (m, n), or EMPTY.
- `int influence_moyo_color(int m, int n)`

  Return the color who has moyo at (m, n), or EMPTY.
- `int influence_area_color(int m, int n)`

  Return the color who has area at (m, n), or EMPTY.
- `int influence_delta_territory(int m, int n, int color)`

  Compute the difference in territory made by a move by color at (m, n).
- `int influence_delta_moyo(int m, int n, int color)`

  Compute the difference in moyo made by a move by color at (m, n).
- `int influence_delta_strict_moyo(int m, int n, int color)`

  Compute the difference in strict moyo made by a move by color at (m, n).
- `int influence_delta_area(int m, int n, int color)`

  Compute the difference in area made by a move by color at (m, n).
- `int influence_delta_strict_area(int m, int n, int color)`

  Compute the difference in strict area made by a move by color at (m, n).
- `void debug_influence_move(int i, int j)`

  Print the influence map when we have computed influence for the move at (i, j).

16.7 Colored display and debugging of influence

It is possible to obtain colored diagrams showing influence from a colored xterm or rxvt window.

- '`-m 0x08`' or '`-m 8`'

  Show diagrams for the initial influence computation. This is done twice, the first time before `make_dragons()` is run and the second time after. The difference is that dead dragons are taken into account the second time. Tactically captured worms are taken into account both times.
- '`-m 0x010`' or '`-m 16`'.

  Show colored display of territory/moyo/area regions.
  - territory: cyan
  - moyo: yellow
  - area: red

  Use either with '`-m 0x8`' (i.e. use '`-m 0x18`') or with '`--debuginfluence`'.

- '`-m 0x20`' or '`-m 32`'.

  Show numerical influence values for white and black. These come in two separate diagrams, the first one for white, the second one for black. Notice that the influence values are represented by floats and thus have been rounded in these diagrams. Use either with '`-m 0x8`' (i.e. use '`-m 0x28`') or with '`--debuginfluence`'.
- '`--debuginfluence` *location*'

  Show influence diagrams after the move at the given location. An important limitation of this option is that it's only effective for moves that the move generation is considering.
- '`-d 0x20`'

  Turn on `DEBUG_INFLUENCE`. This gives tons of messages from the pattern matching performed by the influence code. Too many to be really useful, unfortunately.

Notice that you need to activate at least one of '`-m 0x8`' or '`--debuginfluence`', and at least one of '`-m 0x10`' and '`-m 0x20`', to get any diagrams at all. The first two determine when to print diagrams while the last two determine what diagrams to print.

17 Moyo

The file '`score.c`' contains algorithms for the computation of a number of useful things. Most can be displayed visually using the '`-m`' option (see Section 5.8 [Colored Display], page 33).

In GNU Go 2.6 extensive use was made of an algorithm from Bruno Bouzy's dissertation, which is available at: `ftp://www.joy.ne.jp/welcome/igs/Go/computer/bbthese.ps.Z` This algorithm starts with the characteristic function of the live groups on the board and performs '`n`' operations called dilations, then '`m`' operations called erosions. If n=5 and m=21 this is called the 5/21 algorithm.

The Bouzy 5/21 algorithm is interesting in that it corresponds reasonably well to the human concept of territory. This algorithm is still used in GNU Go 3.0 in the function `estimate_score`. Thus we associate the 5/21 algorithm with the word *territory*. Similarly we use words *moyo* and *area* in reference to the 5/10 and 4/0 algorithms, respectively.

The principle defect of the algorithm is that it is not tunable. The current method of estimating moyos and territory is in '`influence.c`' (see Chapter 16 [Influence], page 145). The territory, moyo and area concepts have been reimplemented using the influence code.

The Bouzy algorithm is briefly reimplemented in the file '`scoring.c`' and is used by GNU Go 3.0 in estimating the score.

Not all features of the old '`moyo.c`' from GNU Go 2.6 were reimplemented—particularly the deltas were not—but the reimplementation may be more readable.

17.1 Bouzy's 5/21 algorithm

Bouzy's algorithm was inspired by prior work of Zobrist and ideas from computer vision for determining territory. This algorithm is based on two simple operations, DILATION and EROSION. Applying dilation 5 times and erosion 21 times determines the territory.

To get a feeling for the algorithm, take a position in the early middle game and try the colored display using the '`-m 1`' option in an RXVT window. The regions considered territory by this algorithm tend to coincide with the judgement of a strong human player.

Before running the algorithm, dead stones (`dragon.status==0`) must be "removed."

Referring to page 86 of Bouzy's thesis, we start with a function taking a high value (ex : +128 for black, -128 for white) on stones on the goban, 0 to empty intersections. We may iterate the following operations:

dilation: for each intersection of the goban, if the intersection is >= 0, and not adjacent to a <0 one, then add to the intersection the number of adjacent >0 intersections. The same for other color : if the intersection is <=0, and not adjacent to a >0 one, then sub to it the number of <0 intersections.

erosion: for each intersection >0 (or <0), subtract (or add) the number of adjacent <=0 (or >=0) intersection. Stop at zero. The algorithm is just : 5 dilations, then 21 erosions. The number of erosions should be 1+n(n-1) where n=number of dilation, since this permit to have an isolated stone to give no territory. Thus the couple 4/13 also works, but it is often not good, for example when there is territory on the 6th line.

For example, let us start with a tobi.

```
                128    0    128
```

1 dilation :

```
                 1          1

             1  128    2   128   1

                 1          1
```

2 dilations :

```
                 1          1

             2   2     3    2    2

         1   2  132    4   132   2   1

             2   2     3    2    2

                 1          1
```

3 dilations :

```
                 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
```

and so on...

Next, with the same example

3 dilations and 1 erosion :

```
              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 dilations and 2 erosions :

```
                1

      2    6    6     6    2

      6   136   8    136   6

      2    6    6     6    2

                1
```

3 dil. / 3 erosions :

```
           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
```

We interpret this as a 1 point territory.

18 Utility Functions

In this Chapter, we document some of the utilities which may be called from the GNU Go engine. If there are differences between this documentation and the source files, the source files are the ultimate reference. You may find it convenient to use Emacs' built in facility for navigating the source to find functions and their in-source documentation (see Section 4.7 [Navigating the Source], page 29).

18.1 General Utilities

Utility functions from '`engine/utils.c`'. Many of these functions underlie autohelper functions (see Section 12.7 [Autohelper Functions], page 98).

- `void change_dragon_status(int x, int y, int status)`

  Change the status of the dragon at `(x,y)`.
- `void count_territory( int *white, int *black)`

  Measure territory.
- `void evaluate_territory( int *white, int *black)`

  Evaluate territory for both sides. Removes dead dragons before counting. The position cannot be reused after this operation.
- `void change_defense(int ai, int aj, int ti, int tj, int dcode)`

  Moves the point of defense of `(ai, aj)` to `(ti, tj)`, and sets `worm[a].defend_code` to `dcode`.
- `void change_attack(int ai, int aj, int ti, int tj, int acode)`

  Moves the point of attack of the worm at `(ai, aj)` to `(ti, tj)`, and sets `worm[a].attack_code` to `acode`.
- `int defend_against(int ti, int tj, int color, int ai, int aj)`

  Returns true if a move at `(ti,tj)` prevents the enemy from playing at `(ai,aj)`. It is checked whether after the moves '`t`', '`a`', the string at '`a`' can be captured.
- `int cut_possible(int i, int j, int color)`

  Returns true if `color` can cut at `(i,j)`. This information is collected by `find_cuts()`, using the `B` patterns in the connections database.
- `int does_attack(int ti, int tj, int ai, int aj)`

  Returns true if the move at `(ti, tj)` attacks `(ai, aj)`. This means that it captures the string, and that `(ai, aj)` is not already dead.
- `int does_defend(int ti, int tj, int ai, int aj)`

  Returns true if the move at `(ti, tj)` defends `(ai, aj)`. This means that it defends the string, and that `(ai, aj)` can be captured if no defense is made.
- `int somewhere(int color, int last_move, ...)`

  Example:

  ```
  somehere(WHITE, 2, ai, aj, bi, bj, ci, cj).
  ```

  returns true if one of the vertices listed satisfies `p[i][j]==color`. Here last_move is the number of moves minus one.

- `int play_break_through_n(int color, int num_moves, ...)`

  This function plays a sequence of moves, alternating between the players and starting with color. After having played through the sequence, the three last coordinate pairs gives a position to be analyzed by `break_through()`, to see whether either color has managed to enclose some stones and/or connected his own stones. If any of the three last positions is empty, it's assumed that the enclosure has failed, as well as the attempt to connect. If one or more of the moves to play turns out to be illegal for some reason, the rest of the sequence is played anyway, and `break_through()` is called as if nothing special happened. Like `break_through()`, this function returns 1 if the attempt to break through was succesful and 2 if it only managed to cut through. The function `break_through` is documented elsewhere (see Section 14.6 [Reading Functions], page 136).

- `int play_break_through_n(int color, int num_moves, ...)`

  plays a sequence of moves, alternating between the players and starting with color. After having played through the sequence, the three last coordinate pairs gives a position to be analyzed by break_through(), to see whether either color has managed to enclose some stones and/or connected his own stones. If any of the three last positions is empty, it's assumed that the enclosure has failed, as well as the attempt to connect. If one or more of the moves to play turns out to be illegal for some reason, the rest of the sequence is played anyway, and break_through() is called as if nothing special happened. Like break_through(), this function returns 1 if the attempt to break through was succesful and 2 if it only managed to cut through.

- `int play_attack_defend_n(int color, int do_attack, int num_moves, ...)`

  Plays a sequence of moves, alternating between the players and starting with color. After having played through the sequence, the last coordinate pair gives a target to attack or defend, depending on the value of do_attack. If there is no stone present to attack or defend, it is assumed that it has already been captured. If one or more of the moves to play turns out to be illegal for some reason, the rest of the sequence is played anyway, and attack/defense is tested as if nothing special happened. A typical use for these functions is to set up a ladder in an autohelper and see whether it works or not.

- `int play_attack_defend2_n(int color, int do_attack, int num_moves, ...)`

  The function play_attack_defend2_n() plays a sequence of moves, alternating between the players and starting with color. After having played through the sequence, the two last coordinate pairs give two targets to simultaneously attack or defend, depending on the value of do_attack. If there is no stone present to attack or defend, it is assumed that it has already been captured. If one or more of the moves to play turns out to be illegal for some reason, the rest of the sequence is played anyway, and attack/defense is tested as if nothing special happened. A typical use for these functions is to set up a crosscut in an autohelper and see whether at least one cutting stone can be captured.

- `int find_lunch(int m, int n, int *wi, int *wj, int *ai, int *aj)`

Looks for a worm adjoining the string at `(m,n)` which can be easily captured. Whether or not it can be defended doesn't matter (see Chapter 10 [Worms and Dragons], page 67). Returns the location of the string in `(*wi, *wj)`, and the location of the attacking move in `(*ai, *aj)`.

- `void modify_depth_values(int n)`

  The parameters `depth`, `backfill_depth`, `fourlib_depth` and `ko_depth` are incremented by '`n`'. This is typically used to avoid horizon effects. By temporarily increasing the depth values when trying some move, one can avoid that an irrelevant move seems effective just because the reading hits a depth limit earlier than it did when reading only on relevant moves.
- `void increase_depth_values(void)`

  Same as `modify_depth_values(1)`.
- `void decrease_depth_values(void)`

  Same as `modify_depth_values(-1)`.
- `void set_temporary_depth_values(int d, int b, int f, int k)`
- `void restore_depth_values()`

  These functions allow more drastic temporary modifications of the depth values. Typical use is to turn certain depth values way down for reading where speed is more important than accuracy, e.g. for the influence function. Temporarily set or restore the values of `depth`, `backfill_depth`, `fourlib_depth`

  `ko_depth`.
- `int same_dragon(int ai, int aj, int bi, int bj)`

  Test whether two dragons are the same. Used by autohelpers.
- `int same_worm(int ai, int aj, int bi, int bj)`

  Test whether two worms are the same. Used by autohelpers.
- `int is_worm_origin(int wi, int wj, int i, int j)`

  Determine whether two worms have the same origin.
- `int accurate_approxlib(int m, int n, int color, int maxlib, int *libi, int *libj)`

  Play a stone at `(m, n)` and count the number of liberties for the resulting string. This requires `(m, n)` to be empty. This function differs from `approxlib()` by the fact that it removes captured stones before counting the liberties. If `libi != NULL` the found liberties are written into the `libi[], libj[]` arrays, but no more than `maxlib` of them. Liberties exceeding `maxlib` may or may not be reported in the return value. If you want to know the exact number of liberties, regardless how large, you should set `maxlib` to `MAXLIBS`.
- `int confirm_safety(int i, int j, int color, int value, int *di, int *dj)`

  This function will detect some blunders. Returns 1 if a move by `color` at `(i,j)` does not diminish the safety of any worm, nor tend to rescue inadvertantly an opponent stone.
- `int double_atari(int m, int n, int color)`

  Returns true if a move by (color) fits the following shape:

```
 .
X*.       (O=color)
OX
```

capturing one of the two X strings. The name is a slight misnomer since this includes attacks which are not necessarily double ataris, though the common double atari is the most important special case.

- `int unconditional_life(int wormi[MAX_STRINGS], int wormj[MAX_STRINGS], int color)`

  Find those worms of the given color that can never be captured, even if the opponent is allowed an arbitrary number of consecutive moves. The coordinates of the origins of these worms are written to the wormi, wormj arrays and the number of non-capturable worms is returned. The algorithm is to cycle through the worms until none remains or no more can be captured. A worm is removed when it is found to be capturable, by letting the opponent try to play on all its liberties. If the attack fails, the moves are undone.

- `int vital_chain(int m, int n)`

  This function returns true if it is judged that the capture of the string at (m,n) is sufficient to create one eye. The current just checks that (m,n) is not a singleton on the first line. For use when called from fill_liberty, this function may optionally return a point `(*di, *dj)` of defense, which, if taken, will presumably make the move at `(i, j)` safe on a subsequent turn.

- `double gg_gettimeofday(void)`

  Get the time of day, calling `gettimeofday` from '`sys/time.h`' if available, otherwise substituting a workaround for portability.

- `void sniff_lunch(int i, int j, int *max, int *min)`

  Computes the number of eyes yielded by capturing a lunch. The surrounding liberties are filled and the stones are removed from the board. Then compute_eyes is called to evaluate the resulting eyespace. The maximum and minimum number of resulting eyes is returned in the variables *max and *min.

- `int unconditional_life(int wormi[MAX_STRINGS], int wormj[MAX_STRINGS], int color)`

  Find those worms of the given color that can never be captured, even if the opponent is allowed an arbitrary number of consecutive moves. The coordinates of the origins of these worms are written to the `wormi`, `wormj` arrays and the number of non-capturable worms is returned.

18.2 Print utilities

Utility functions from '`engine/printutils.c`'.

- `static void vgprintf(FILE* outputfile, const char *fmt, va_list ap)`

  This function underpins all the `TRACE` and `DEBUG` stuff. It is static to '`printutils.c`' but documented here for completeness. Accepts `%c`, `%d`,

`%f`, `%s`, and `%x` as usual. But it also accepts `%m`, which takes TWO integers and writes a move. Other nonstandard format strings are `%H` for writing a hash value and `%C` to convert a color value into a string. `%o` at start means outdent (ie cancel indent). The scope of this function is limited to '`engine/utils.c`' but the format codes `%m` and `%c`$ work for all its relatives such as `TRACE`.

- `void gprintf(const char *fmt, ...)`

  Required wrapper to `vgprintf`. Writes to `stderr`.

- `void mprintf(const char *fmt, ...)`

  Identical to `gprintf` except that it prints to `stdout`. Useful when `%m` is needed for non-error messages, e.g. in the ascii interface.

- `void TRACE(const char *fmt, ...)`

  Basic tracing function. Like `gprintf` but prints only if `verbose>0`. Set the `verbose` level with the '`-t`' option (see Section 3.9 [Invoking GNU Go], page 12). Variants `RTRACE`, etc. are documented in the source.

- `void DEBUG(int flag, const char *fmt, ...)`

  Like `TRACE` but conditioned on a debug flag being set, usually at the command line with '`-d`' option (see Section 3.9 [Invoking GNU Go], page 12).

- `void abortgo(const char *file, int line, const char *msg, int x, int y)`

  A wrapper around `abort()` which shows the state variables at the time of the problem. `(i, j)` are typically a related move, or `-1, -1`.

- `ASSERT`

  This is the usual way of calling `abortgo`. This macro (defined in '`liberty.h`') terminates the program if `condition` fails.

- `const char *color_to_string(int color)`

  Convert a color value to a string.

- `const char * location_to_string(int i, int j)`

  Converts a board location to a string

- `const char * status_to_string(int i, int j)`

  Converts a status to a string.

19 Incremental Algorithms in Reading

The algorithms in `board.c` implement a method of incremental board updates that keeps track of the following information for each string:

- The color of the string.
- Number of stones in the string.
- Origin of the string, i.e. a canonical reference point, defined to be the stone with smallest '`i`' coordinate and if there is a tie with smallest '`j`' coordinate.
- A list of the stones in the string.
- Number of liberties.
- A list of the liberties. If there are too many liberties the list is truncated.
- The number of neighbor strings.
- A list of the neighbor strings.

The basic data structure is

```
struct string_data {
  int color;                  /* Color of string, BLACK or WHITE */
  int size;                   /* Number of stones in string. */
  int origini;                /* Coordinates of "origin", i.e. */
  int originj;                /* "upper left" stone. */
  int liberties;              /* Number of liberties. */
  int libi[MAX_LIBERTIES];    /* Coordinates of liberties. */
  int libj[MAX_LIBERTIES];
  int neighbors;              /* Number of neighbor strings */
  int neighborlist[MAXCHAIN]; /* List of neighbor string numbers. */
  int mark;                   /* General purpose mark. */
};

struct string_data string[MAX_STRINGS];
```

It should be clear that almost all information is stored in the `string` array. To get a mapping from the board coordinates to the `string` array we have

```
int string_number[MAX_BOARD][MAX_BOARD];
```

which contains indices into the `string` array. This information is only valid at nonempty vertices, however, so it is necessary to first verify that `p[i][j] != EMPTY`.

The `string_data` structure does not include an array of the stone coordinates. This information is stored in a separate array (or rather two):

```
int next_stonei[MAX_BOARD][MAX_BOARD];
int next_stonej[MAX_BOARD][MAX_BOARD];
```

These arrays implement cyclic linked lists of stones. Each vertex contains a pointer to another (possibly the same) vertex. Starting at an arbitrary stone on the board, following these pointers should traverse the entire string in an arbitrary order before coming back to the starting point. As for the 'string_number' array, this information is invalid at empty points on the board. This data structure has the good properties of requiring fixed space (regardless of the number of strings) and making it easy to add a new stone or join two strings.

Additionally the code makes use of some work variables:

```
static int ml[MAX_BOARD][MAX_BOARD];
static int liberty_mark;
static int string_mark;
static int next_string;
static int strings_initialized = 0;
```

The `ml` array and `liberty_mark` are used to "mark" liberties on the board, e.g. to avoid counting the same liberty twice. The convention is that if `ml[i][j]` has the same value as `liberty_mark`, then `(i, j)` is marked. To clear all marks it suffices to increase the value of `liberty_mark`, since it is never allowed to decrease.

The same relation holds between the `mark` field of the `string_data` structure and `string_mark`. Of course these are used for marking individual strings.

`next_string` gives the number of the next available entry in the `string` array. Then `strings_initialized` is set to one when all data structures are known to be up to date. Given an arbitrary board position in the '`p`' array, this is done by calling `incremental_board_init()`. It is not necessary to call this function explicitly since any other function that needs the information does this if it has not been done.

The interesting part of the code is the incremental update of the data structures when a stone is played and subsequently removed. To understand the strategies involved in adding a stone it is necessary to first know how undoing a move works. The idea is that as soon as some piece of information is about to be changed, the old value is pushed onto a stack which stores the value and its address. The stack is built from the following structures:

```
struct change_stack_entry {
  int *address;
  int value;
};

struct change_stack_entry change_stack[STACK_SIZE];
int change_stack_index;
```

and manipulated with the macros

```
BEGIN_CHANGE_RECORD()
PUSH_VALUE(v)
POP_MOVE()
```

Calling `BEGIN_CHANGE_RECORD()` stores a null pointer in the address field to indicate the start of changes for a new move. As mentioned earlier `PUSH_VALUE()` stores a value and its corresponding address. Assuming that all changed information has been duly pushed onto the stack, undoing the move is only a matter of calling `POP_MOVE()`, which simply assigns the values to the addresses in the reverse order until the null pointer is reached. This description is slightly simplified because this stack can only store 'int' values and we need to also store changes to the board. Thus we have two parallel stacks where one stores `int` values and the other one stores `Intersection` values.

When a new stone is played on the board, first captured opponent strings, if any, are removed. In this step we have to push the board values and the `next_stone` pointers for the removed stones, and update the liberties and neighbor lists for the neighbors of the removed strings. We do not have to push all information in the 'string' entries of the removed strings however. As we do not reuse the entries they will remain intact until the move is pushed and they are back in use.

After this we put down the new stone and get three distinct cases:

1. The new stone is isolated, i.e. it has no friendly neighbor.
2. The new stone has exactly one friendly neighbor.
3. The new stone has at least two friendly neighbors.

The first case is easiest. Then we create a new string by using the number given by `next_string` and increasing this variable. The string will have size one, `next_stone` points directly back on itself, the liberties can be found by looking for empty points in the four directions, possible neighbor strings are found in the same way, and those need also to remove one liberty and add one neighbor.

In the second case we do not create a new string but extend the neighbor with the new stone. This involves linking the new stone into the cyclic chain, if needed moving the origin, and updating liberties and neighbors. Liberty and neighbor information also needs updating for the neighbors of the new stone.

In the third case finally, we need to join already existing strings. In order not to have to store excessive amounts of information, we create a new string for the new stone and let it assimilate the neighbor strings. Thus all information about those can simply be left around in the 'string' array, exactly as for removed strings. Here it becomes a little more complex to keep track of liberties and neighbors since those may have been shared by more than one of the joined strings. Making good use of marks it all becomes rather straightforward anyway.

The often used construction

```
FIRST_STONE(s, i, j);
do {
    ...
    NEXT_STONE(i, j);
} while (!BACK_TO_FIRST_STONE(s, i, j));
```

traverses the stones of the string with number '`s`' exactly once, with `(i, j)` holding the coordinates. In general `(i, j)` are used as board coordinates and '`s`' as an index into the `string` array or sometimes a pointer to an entry in the `string` array.

20 The Go Text Protocol

20.1 The GNU Go Text Protocol

GNU Go 3.0 introduces a new interface, the Go Text Protocol (GTP). The intention is to make an interface that is better suited for machine-machine communication than the ascii interface and simpler, more powerful, and more flexible than the Go Modem Protocol.

The GTP has two principal current applications: it is used in the test suite (see Chapter 21 [Regression], page 170) and it is used to communicate with `gnugoclient`, which is not part of the GNU Go distribution, but has been used to run GNU Go on NNGS. Other potential uses might be any of the current uses of the GMP, for which the GTP might serve as a replacement. This would likely entail extension and standardization of the protocol.

A sample GTP session may look as follows:

```
hannah 2289% ./interface/gnugo --quiet --mode gtp
1 loadsgf regression/games/incident156.sgf 249
=1

2 countlib C3
=2 4

3 findlib C3
=3 C4 B3 D3 B2

5 attack C3
=5 0

owl_attack C3
= 1 B4

3 owl_defend C3
=3 1 B5

owl_attack A2
? vertex must not be empty

quit
=
```

By specifying '`--mode gtp`' GNU Go starts in the GTP interface. No prompt is used, just start giving commands. The commands have the common syntax

```
[id] command_name [arguments]
```

The command is exactly one line long, i.e. it ends as soon as a newline appears. It's not possible to give multiple commands on the same line. Before the command name an optional identity number can be specified. If present it must be an integer between 0 and 2^31-1. The id numbers may come in any order or be reused. The rest of the line after the command name is assumed to be arguments for the command. Empty lines are ignored, as is everything following a hash sign up to the end of the line.

If the command is successful, the response is of the form

```
=[id] result
```

Here '=' indicates success and `id` is the identity number given in the command or the empty string if the id was omitted. This is followed by the result, which is a text string ending with two consecutive newlines.

If the command fails for some reason, the response takes the form

```
?[id] error_message
```

Here '?' indicates failure, `id` is as before, and `error_message` gives an explanation for the failure. This string also ends with two consecutive newlines.

The available commands may always be listed using the single command `help`. Currently this gives the list below.

```
attack
black
boardsize
color
combination_attack
countlib
debug_influence
debug_move_influence
decrease_depths
defend
dragon_data
dragon_status
dump_stack
echo
eval_eye
final_score
findlib
fixed_handicap
genmove_black
genmove_white
get_life_node_counter
get_owl_node_counter
get_reading_node_counter
get_trymove_counter
gg_genmove
help
increase_depths
influence
is_legal
komi
level
loadsgf
move_influence
name
new_score
owl_attack
```

```
owl_defend
popgo
prisoners
quit
report_uncertainty
reset_life_node_counter
reset_owl_node_counter
reset_reading_node_counter
reset_trymove_counter
same_dragon
showboard
trymove
tune_move_ordering
version
white
worm_cutstone
worm_data
```

For exact specification of their arguments and results we refer to the comments of the functions in '`interface/play_gtp.c`'.

The protocol is asymmetric and involves two parties, which we may call '`A`' and '`B`'. '`A`' is typically some kind of arbiter or relay and '`B`' is a go engine. All communication is initiated by '`A`' in form of commands, to which '`B`' responds.

Potential setups include:

1. Regression testing.

    A (regression script) – B (engine).

    A sets up a board position and asks B to e.g. generate a move or find an attack on a specific string.

2. Human vs program.

    A (GUI) – B (engine)

    The GUI relays moves between the human and the engine and asks the engine to generate moves. Optionally the GUI may also use GTP to ask the engine which moves are legal or give a score when the game is finished.

3. Program vs program with arbiter.

    B1 (engine 1) – A (arbiter) – B2 (engine 2)

    A relays moves between the two engines and alternately asks the engines to generate moves. This involves two different GTP channels, the first between A and B1, and the second between A and B2. There is no direct communication between B1 and B2. The arbiter dictates board size, komi, rules, etc.

4. Program vs program without arbiter.

    The same as above except that B1 includes the arbiter functionality and the first GTP link is shortcut.

5. Connection between go server and program.

    Go server – A (relay) – B (engine)

> A talks with a go server using whatever protocol is needed and listens for match requests. When one arrives it accepts it, starts the go engine and issues GTP commands to set up board size, komi, etc. and if a game is restarted it also sets up the position. Then it relays moves between the server and the engine and asks the engine to generate new moves when it is in turn.

Setups 1 and 5 are in active and regular use with GNU Go. Programs implementing setup 3 are also distributed with GNU Go (the files '`interface/gtp_examples/twogtp`' and '`interface/gtp_examples/2ptkgo.pl`').

The GTP is currently unfinished and unstandardized. It is hoped that it will grow to fill the needs currently served by the GMP and perhaps other functions. As it is yet unstandardized, this section gives some general remarks which we hope will constrain its development. We also discuss how the GTP is implemented in GNU Go, for the benefit of anyone wishing to add new commands. Notice that the current set of GTP commands is a mix of generally useful ones and highly GNU Go specific ones. Only the former should be part of a standardized protocol while the latter should be private extensions.

The purpose of the protocol is machine-machine communication. It may be tempting to modify the protocol so that it becomes more comfortable for the human user, for example with an automatic showboard after every move. **This is absolutely not the purpose of the protocol!** Use the ascii interface instead if you're inclined to make such a change.

Newlines are implemented differently on different operating systems. On Unix, a newline is just a line feed LF (ascii \012). On DOS/Windows it is CRLF (\013\012). Thus whether GNU Go sends a carriage return with the line feed depends on which platform it is compiled for. The arbiter should silently discard carriage returns.

Applications using GTP should never have to guess about the basic structure of the responses, defined above. The basic construction for coding a GTP command can be found in `gtp_countlib()`:

```
static int
gtp_countlib(char *s, int id)
{
  int i, j;
  if (!gtp_decode_coord(s, &i, &j))
    return gtp_failure(id, "invalid coordinate");

  if (p[i][j] == EMPTY)
    return gtp_failure(id, "vertex must not be empty");

  return gtp_success(id, "%d", countlib(i, j));
}
```

The functions `gtp_failure()` and `gtp_success()` automatically ensures the specified response format, assuming the strings they are printing do not end with a newline.

Sometimes the output is too complex for use with gtp_success, e.g. if we want to print vertices, which gtp_success() doesn't support. Then we have to fall back to the construction in e.g. `gtp_genmove_white()`:

```
static int
gtp_genmove_white(char *s, int id)
```

```
{
  int i, j;
  UNUSED(s);
  if (genmove(&i, &j, WHITE) >= 0)
    play_move(i, j, WHITE);

  gtp_printid(id, GTP_SUCCESS);
  gtp_print_vertex(i, j);
  return gtp_finish_response();
}
```

Here `gtp_printid()` writes the equal sign and the request id while `gtp_finish_response()` adds the final two newlines. The next example is from `gtp_influence()`:

```
  gtp_printid(id, GTP_SUCCESS);
  get_initial_influence(color, 1, white_influence,
black_influence, influence_regions);
  print_influence(white_influence, black_influence, influence_regions);
  /* We already have one newline, thus can't use gtp_finish_response(). */
  gtp_printf("\n");
  return GTP_OK;
```

As we have said, the response should be finished with two newlines. Here we have to finish up the response ourselves since we already have one newline in place.

One problem that can be expected to be common is that an engine happens to finish its response with three (or more) rather than two consecutive newlines. This is an error by the engine that the controller can easily detect and ignore. Thus a well behaved engine should not send stray newlines, but should they appear the controller should ignore them. The opposite problem of an engine failing to properly finish its response with two newlines will result in deadlock. Don't do this mistake!

Although it doesn't suffice in more complex cases, gtp_success() is by far the most convenient construction when it does. For example, the function `gtp_report_uncertainty` takes a single argument which is expected to be "on" or "off", after which it sets the value of `report_uncertainty`, a variable which affects the form of future GTP responses, reports success, and exits. The function is coded thus:

```
static int
gtp_report_uncertainty(char *s, int id)
{
  if (!strncmp(s, "on", 2)) {
    report_uncertainty = 1;
    return gtp_success(id, "");
  }
  if (!strncmp(s, "off", 3)) {
    report_uncertainty = 0;
    return gtp_success(id, "");
  }
  return gtp_failure(id, "invalid argument");
}
```

20.2 Regression testing with GTP

GNU Go uses GTP for regression testing. These tests are implemented as files with GTP commands, which are fed to GNU Go simply by redirecting stdin to read from a file. The output is filtered so that equal signs and responses from commands without id numbers are removed. These results are then compared with expected results encoded in GTP comments in the file, using matching with regular expressions. More information can be found in the regression chapter (see Chapter 21 [Regression], page 170).

21 Regression testing

The standard purpose of regression testing is to avoid getting the same bug twice. When a bug is found, the programmer fixes the bug and adds a test to the test suite. The test should fail before the fix and pass after the fix. When a new version is about to be released, all the tests in the regression test suite are run and if an old bug reappears, this will be seen quickly since the appropriate test will fail.

The regression testing in GNU Go is slightly different. A typical test case involves specifying a position and asking the engine what move it would make. This is compared to one or more correct moves to decide whether the test case passes or fails. It is also stored whether a test case is expected to pass or fail, and deviations in this status signify whether a change has solved some problem and/or broken something else. Thus the regression tests both include positions highlighting some mistake being done by the engine, which are waiting to be fixed, and positions where the engine does the right thing, where we want to detect if a change breaks something.

21.1 Regression testing in GNU Go

Regression testing is performed by the files in the '`regression/`' directory. The tests are specified as GTP commands in files with the suffix '`.tst`', with corresponding correct results and expected pass/fail status encoded in GTP comments following the test. To run a test suite the shell scripts '`test.sh`', '`eval.sh`', and `regress.sh` can be used. There are also Makefile targets to do this. If you `make all_batches` most of the tests are run.

Game records used by the regression tests are stored in the directory '`regression/games/`' and its subdirectories.

21.2 Test suites

The regression tests are grouped into suites and stored in files as GTP commands. A part of a test suite can look as follows:

```
# 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]*
```

Lines starting with a hash sign, or in general anything following a hash sign, are interpreted as comments by the GTP mode and thus ignored by the engine. GTP commands are executed in the order they appear, but only those on numbered lines are used for testing. The comment lines starting with `#?` are magical to the regression testing scripts and indicate correct results and expected pass/fail status. The string within brackets is matched as a regular expression against the response from the previous numbered GTP command. A particular useful feature of regular expressions is that by using '`|`' it is possible to specify alternatives. Thus `B14|D17` above means that if either `B14` or `D17` is the move generated in test case 90, it passes. There is one important special case to be aware of. If the correct result string starts with an exclamation mark, this is excluded from the regular expression but afterwards the result of the matching is negated. Thus `!P13` in test case 95 means that any move except `P13` is accepted as a correct result.

In test case 100, the brackets on the `#?` line is followed by an asterisk. This means that the test is expected to fail. If there is no asterisk, the test is expected to pass. The brackets may also be followed by a '`&`', meaning that the result is ignored. This is primarily used to report statistics, e.g. how many tactical reading nodes were spent while running the test suite.

21.3 Performing tests

`./test.sh blunder.tst` runs the tests in '`blunder.tst`' and prints the results of the commands on numbered lines, which may look like:

```
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
```

This is usually not very informative, however. More interesting is `./eval.sh blunder.tst` which also compares the results above against the correct ones in the test file and prints a report for each test on the form:

```
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'
```

The result of a test can be one of four different cases:

- `passed`: An expected pass

  This is the ideal result.
- `PASSED`: An unexpected pass

  This is a result that we are hoping for when we fix a bug. An old test case that used to fail is now passing.
- `failed`: An expected failure

  The test failed but this was also what we expected, unless we were trying to fix the particular mistake highlighted by the test case. These tests show weaknesses of the GNU Go engine and are good places to search if you want to detect an area which needs improvement.
- `FAILED`: An unexpected failure

  This should nominally only happen if something is broken by a change. However, sometimes GNU Go passes a test, but for the wrong reason or for a combination of wrong reasons. When one of these reasons is fixed, the other one may shine through so that the test suddenly fails. When a test case unexpectedly fails, it is necessary to make a closer examination in order to determine whether a change has broken something.

If you want a less verbose report, `./regress.sh . blunder.tst` does the same thing as the previous command, but only reports unexpected results. The example above is compressed to

```
3 unexpected PASS!
5 unexpected PASS!
7 unexpected PASS!
```

For convenience the tests are also available as makefile targets. For example, `make blunder` runs the tests in the blunder test suite by executing `eval.sh blunder.tst`. `make test` runs all test suites in a sequence using the `regress.sh` script.

Appendix A Copying

The program GNU Go is distributed under the terms of the GNU General Public License (GPL). Its documentation is distributed under the terms of the GNU Free Documentation License (GFDL).

A.1 GNU GENERAL PUBLIC LICENSE

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in

independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘`show w`’ and ‘`show c`’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

ADDENDUM: How to use this License for your documents

A.3 The Go Text Protocol License

Concept Index

Functions Index

Table of Contents