1. Introduction

This is GNU Go 3.8, 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 exists that plays at the same level as the strongest human players.

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. This is especially true in view of the successes of Monte Carlo methods, and a general recent improvement of computer Go.

Before GNU Go, 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, 2002, 2003, 2004, 2005, 2006, 2007 and 2008 by the Free Software Foundation except as noted below.

All source files are distributed under the GNU General Public License (see section GNU GENERAL PUBLIC LICENSE, version 3 or any later version), except

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-3.8. Now do:

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

Several configure options will be explained in the next section. You do not need to set these unless you are dissatisfied with GNU Go's performance or wish to vary the experimental options.

As an example,

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

will make a binary in which the default level is 9, and the experimental “cosmic” ' option is enabled. A list of all configure options can be obtained by running `./configure --help`. Further information about the experimental options can be found in the next section (see section Configure Options).

After running configure and make, 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 Window System), Quarry, Jago (on any platform with a Java Runtime Environment) or other client programs offering a GUI.

You can get the most recent version of CGoban 1 from http://sourceforge.net/projects/cgoban1/. The earlier version 1.12 is available from 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 Running GNU Go via CGoban, for instructions on how to run GNU Go from Cgoban, or See section Other Clients, for Jago or other clients.

Quarry is available at http://home.gna.org/quarry/.

2.2 Configure Options

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

2.2.1 Ram Cache

By default, GNU Go makes a cache of about 8 Megabytes in RAM for its internal use. The cache is used to store intermediate results during its analysis of the position. More precisely the default cache size is 350000 entries, which translates to 8.01 MB on typical 32 bit platforms and 10.68 MB on typical 64 bit platforms.

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=32
```

creates a cache of size 32 megabytes. If you omit this, your default cache size will be 8-11 MB as discussed above. Setting cache size negative also gives the default size. 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.

2.2.2 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.

2.2.3 Other Options

Anything new in the engine is generally tested as an experimental option which can be turned on or off at compile time or run time. Some “experimental” options such as the break-in code are no longer experimental but are enabled by default.

This section can be skipped unless you are interested in the experimental options.

Moreover, some configure options were removed from the stable release. For example it is known that the owl extension code can cause crashes, so the configure option –enable-experimental-owl-ext was disabled for 3.8.

The term “default” must be clarified, since there are really two sets of defaults at hand, runtime defaults specified in ‘`config.h`’ and compile time default values for the runtime defaults, contained in ‘`configure`’ (which is created by editing ‘`configure.in`’ then running `autoconf`. For example we find in ‘`config.h`’

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

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

This means that the experimental cosmic option, which causes GNU Go to play a center-oriented game (and makes the engine weaker) is disabled by default, but that the break-in module is used. These are defaults which are used when GNU Go is run without command line options. They can be overridden with the run time options:

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

Alternatively you can configure GNU Go as follows:

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

then recompile GNU Go. This changes the defaults in ‘`config.h`’, so that you do not have to pass any command line options to GNU Go at run time to get the experimental owl extension turned on and the experimental break-in code turned off.

If you want to find out what experimental options were compiled into your GNU Go binary you can run `gnugo --options` to find out. Here is a list of experimental options in GNU Go.

- `experimental-break-in`. Experimental break-in code (see section Break Ins). You should not need to configure this because the break in code is enabled by default in level 10, and is turned off at level 9. If you don't want the breakin code just play at level 9.
- `cosmic-gnugo`. An experimental style which plays a center oriented game and has a good winning rate against standard GNU Go, though it makes GNU Go weaker against other opponents.
- `large-scale`. Attempt to make large-scale captures. See:

  http://lists.gnu.org/archive/html/gnugo-devel/2003-07/msg00209.html

  for the philosophy of this option. This option makes the engine slower.
- `metamachine`. Enables the metamachine, which allows you to run the engine in an experimental mode whereby it forks a new `gnugo` process which acts as an “oracle.” Has no effect unless combined with the ‘`--metamachine`’ run-time option.

Other options are not experimental, and can be changed as configure or runtime options.

- `chinese-rules` Use Chinese (area) counting.
- `resignation-allowed` Allow GNU Go to resign games. This is on by default.

2.3 Compiling GNU Go on Microsoft platforms

2.3.1 Building with older visual studio

The distribution directories contain some .dsp and .dsw files with GNU Go. These have been brought up to date in the sense that they should work if you have the older VC++ with Visual Studio 6 but the distributed .dsp and .dsw files will only be of use with older version of Visual Studio.

In most cases (unless you are building in Cygwin) the preferred way to build GNU Go on Windows platforms is to use CMake. CMake understands about many versions of Visual C/Visual Studio, and will generate project/solution files for the tools installed on your system. So even if you have Visual Studio 6 you may use CMake and dispense with the distributed .dsp and .dsw files.

2.3.2 Building with Visual Studio project files

Before you compile the GNU Go source, you need to run CMake first, to generate the build files you'll give to Visual Studio.

From the cmd.exe command prompt, CD into the GNU Go source directory. To confirm you're in the right place, you should see the file 'CMakeLists.txt' in the top-level directory of the GNU Go code (as well as others in lower subdirectories).

Direct CMake to generate the new Visual Studio build files by typing:

```
    cmake CMakeLists.txt
```

Compile the code by invoking the newly-created Solution file:

```
    vcbuild GNUGo.sln
```

This will take a few moments, as CMake generates 4 debug/retail targets:

```
    debug
    release
    minsizerel
    relwithdebinfo
```

For each of these targets, Visual Studio is generating a version of gnugo.exe:

```
    interface\debug\gnugo.exe
    interface\release\gnugo.exe
    interface\minsizerel\gnugo.exe
    interface\relwithdebinfo\gnugo.exe
```

Additionally, there is an 'Install' target available, that will copy the the gnugo.exe into the %ProgramFiles% directory. To do this, type:

```
    vcbuild INSTALL.vcproj
```

This should result in copying GNU/Go into:

```
    "%ProgramFiles%\GNUGo\bin\gnugo.exe" --options
```

In addition to command line use, CMake also has a GUI version. Users of the Visual Studio GUI might prefer to use that.

2.3.3 Building with Nmake makefiles

GNU Go will also build using NMake makefiles. Optionally, instead of Visual Studio project/solution files, you may direct CMake to generate NMake makefiles. To generate the makefiles:

```
    cmake -G "NMake Makefiles" CMakeLists.txt
```

The default rule for the makefile is 'all'. Use the 'help' rule to show a list of available targets.

```
    nmake -f Makefile help
```

To compile GNU Go:

```
    nmake -f Makefil, all
```

One sysand 2009 tems, GNU GO may fail to build when using NMake makefiles. only fails the first time run, run NMake again with the 'clean all' targets, and it will compile the second and subsequent times.

```
    nmake -f Makefile clean all
```

Which will successfully generate a gnugo.exe.

```
    interface\gnugo.exe --options
```

2.3.4 Building with MinGW Makefiles

GNU Go can be built on Windows systems using MinGW.

This development environment uses: the GCC compiler (gcc.exe, not cl.exe), the Microsoft C runtime libraries (MSCRT, not GLibC), the GNU Make build tool (`mingw32-make.exe`, not NMake), all from the Windows shell (`cmd.exe`, not sh/bash).

For CMake to work, in addition to the base MinGW installation, the C++ compiler (g++.exe) and GNU Make (mingw32-make.exe) need to be installed. This was tested using GCC v3, not the experimental v4. To debug, use GDB, as the GCC-generated symbols won't work with NTSD/Windbg/Visual Studio.

To create the makfiles, run CMake with the MinGW generator option:

```
    cmake -G "MinGW Makefiles" CMakeLists.txt
```

To build GNU Go, from a cmd.exe shell, run GNU Make (against the newly-created 'Makefile' and it's default 'all' target):

```
    mingw32-make
    ..\interface\gnugo.exe --options
```

2.3.5 Building with MSYS makefiles (MinGW)

GNU Go can be built on Windows systems using MSYS.

This development environment uses: the GCC compiler (gcc.exe, not cl.exe), the Microsoft C runtime libraries (MSCRT, not GLibC), the GNU Make build tool (make, not NMake), all from the GNU Bash (sh.exe, not cmd.exe).

To create the makfiles, run CMake with the MSYS generator option:

```
    cmake -G "MSYS Makefiles" CMakeLists.txt
```

Start MSYS's Bash shell, either clicking on a shortcut on from the command line:

```
    cd /d c:\msys\1.0
    msys.bat
```

To build GNU Go, from a Bash shell, run GNU Make (against the newly-created 'Makefile' and it's default 'all' target):

```
    make
    ../interface/gnugo.exe --options
```

To debug, use GDB, as the GCC-generated symbols won't work with NTSD/Windbg/Visual Studio.

2.3.6 Building on cygwin

With Cygwin, you should be able to

```
  tar zxvf gnugo-3.8.tar.gz
  cd gnugo-3.8
  env CC='gcc -mno-cygwin' ./configure
  make
```

2.3.7 Testing on Windows:

'`regression/regress.cmd`' is a simplified cmd.exe-centric port of the main gnugo Unix shell script regress.sh. It can be used to help verify that the generated binary might be operational. Read the script's comment header for more information. For access to the full GNU Go tests, use Unix, not Windows.

To test:

```
    cd regression
    regress.cmd ..\interface\gnugo.exe
```

2.4 Macintosh

If you have Mac OS X you can build GNU Go using Apple's compiler, which is derived from GCC. You will need Xcode.

One issue is that the configure test for socket support is too conservative. On OS/X, the configure test fails, but actually socket support exists. So if you want to be able to connect to the engine through tcp/ip (using gtp) you may `configure --enable-socket-support`. There will be an error message but you may build the engine and socket support should work.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

3. Using GNU Go

3.1 Getting Documentation

You can obtain a printed copy of the manual by running `make gnugo.pdf` in the 'doc/' directory, then printing the resulting 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. This will create a file called 'gnugo.info' (and a few others) and copy them into a system directory such as '/usr/local/share/info'. You may then add them to your info directory tree with the command `install-info --info-file=[path to gnugo.info] --info-dir=[path to dir]`. 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`, `make gnugo.ps`, or `make gnugo.pdf` in the 'doc/' directory. (`make gnugo.pdf` only works after you have converted all .eps-files in the doc/ directory to .pdf files, e.g. with the utility epstopdf.)

You can make an HTML version with the command `makeinfo --html gnugo.texi`. If you have `texi2html`, better HTML documentation may be obtained by `make gnugo_ovr.html` in the 'doc/' directory.

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

There are two different programs called CGoban, both written by William Shubert. In this documentation, CGoban means CGoban 1.x, the older program. You should get a copy with version number 1.12 or higher.

CGoban is an extremely nice way to run GNU Go. CGoban provides a beautiful graphic user interface under X Window System.

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 Other Clients

In addition to CGoban (see section Running GNU Go via CGoban) there are a number of other good clients that are capable of running GNU Go. Here are the ones that we are aware of that are Free Software. This list is part of a larger list of free Go programs that is maintained at http://www.gnu.org/software/gnugo/free_go_software.html.

- Quarry (http://home.gna.org/quarry/) is a GPL'd client that supports GTP. Works under GNU/Linux and requires GTK+ 2.x and librsvg 2.5. Supports GNU Go as well as other engines. Can play not only Go, but also a few other board games.
- qGo (http://sourceforge.net/projects/qgo/) is a full featured Client for playing on the servers, SGF viewing/editing, and GNU Go client written in C++ for GNU/Linux, Windows and Mac OS X. Can play One Color Go. Licensed GPL and QPL.
- ccGo (http://ccdw.org/~cjj/prog/ccgo/) is a GPL'd client written in C++ capable of playing with GNU Go, or on IGS.
- RubyGo (http://rubygo.rubyforge.org/) is a GPL'd client by J.-F. Menon for IGS written in the scripting language Ruby. RubyGo is capable of playing with GNU Go using the GTP.
- Dingoui (http://dingoui.sourceforge.net/) is a free GMP client written in GTK+ which can run GNU Go.
- Jago (http://www.rene-grothmann.de/jago/) is a GPL'd Java client which works for both Microsoft Windows and X Window System.
- Sente Software's FreeGoban (http://www.sente.ch/software/goban/freegoban.html) is a well-liked user interface for GNU Go (and potentially other programs) distributed under the GPL.
- Mac GNU Go (http://www1.u-netsurf.ne.jp/~future/HTML/macgnugo_ovr.html) is a front end for GNU Go 3.2 with both English and Japanese versions. License is GPL.
- Quickiego (http://www.geocities.com/secretmojo/QuickieGo/) is a Mac interface to GNU Go 2.6.
- Gogui (http://sourceforge.net/projects/gogui/) from Markus Enzenberger is a Java workbench that allows you to play with a gtp (http://www.lysator.liu.se/~gunnar/gtp) engine such as GNU Go. Licence is GPL. Gogui does not support gmp or play on servers but is potentially very useful for programmers working on GNU Go or other engines.

3.4 Ascii Interface

Even if you do not have any client program, 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.

You may play games with a time limit against GNU Go in ascii mode. For this, the Canadian time control system is used. (See http://en.wikipedia.org/wiki/Byoyomi and http://senseis.xmp.net/?CanadianByoyomi.) That is, you have a main time to be followed by byo-yomi periods. After the main time is exhausted you have a certain number of moves to be made in a certain number of seconds. (see section Invoking GNU Go: Command line options)

3.5 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 or with the mouse, and you can have a nice graphical display of the board within emacs.

You will need the file ‘`interface/gnugo.el`’. There is a version of this distributed with GNU Go but it only works with Emacs 21. Most Emacsen are Emacs 22 however. Therefore you should get the latest version of gnugo.el by Thien-Thi Nguyen, which you can find at http://www.gnuvola.org/software/j/gnugo/ or http://www.emacswiki.org/emacs/gnugo.el.

You will also need some xpm files for the graphical display. You can either use those distributed by Thien-Thi Nguyen (at the first URL above) or those distributed with GNU Go, either the file ‘`interface/gnugo-xpms.el`’ or (for high resolution displays) ‘`interface/gnugo-big-xpms.el`’.

Load the file ‘`interface/gnugo.el`’ and ‘`interface/gnugo-xpms.el`’. You may do this using the Emacs `M-x load-file` command.

When you start a game with `M-x gnugo`, you will first see an ascii board. However typing `i' toggles a graphical board display which is very nice. This is a pleasant way to play GNU Go. You may get help by typing `C-x m`.

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 introduced a new protocol, the Go Text Protocol (see section The Go Text Protocol) which we hope can serve the functions currently used by the GMP. The GTP is becoming increasingly adopted by other programs as a method of interprocess communication, both by computer programs and by clients. Still the GMP is widely used in tournaments.

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 Game Format

The Smart Game 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. GNU Go will generate a move for the player who is about to move. If you want to override this and generate a move for the other player you may add the option '`--color <color>`' where <color> is `black` or `white`.

'`-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

'`-O`' , '`--output-flags flags`'

Add useful information to the sgf file. Flags can be 'd', 'v' or both (i.e. 'dv'). If 'd' is specified, dead and critical dragons are marked in the sgf file. If 'v' is specified, move valuations around the board are indicated.

'`--mode mode`'

Force the playing mode ('ascii', 'emacs,' '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.

'`--resign-allowed`'

GNU Go will resign games if this option is enabled. This is the default unless you build the engine with the configure option '`--disable-resignation-allowed`' . Unfortunately the Go Modem Protocol has no provision for passing a resignation, so this option has no effect in GMP mode.

'`--never-resign`'

GNU Go will not resign games.

'`--resign-allowed`'

GNU Go will resign lost games. This is the default.

3.9.2 Monte Carlo Options

GNU Go can play Monte Carlo Go on a 9x9 board. (Not available for larger boards.) It makes quite a strong engine. Here are the command line options.

'`--monte-carlo`'

Use Monte Carlo move generation (9x9 or smaller).

'`--mc-games-per-level <number>`'

Number of Monte Carlo simulations per level. Default 8000. Thus at level 10, GNU Go simulates 80,000 games in order to generate a move.

'`--mc-list-patterns`'

list names of builtin Monte Carlo patterns

'`--mc-patterns <name>`'

Choose a built in Monte Carlo pattern database. The argument can be '`mc_mogo_classic`' , '`mc_montegnu_classic`' or '`mc_uniform`' .

'`--mc-load-patterns <filename>`'

read Monte Carlo patterns from file

3.9.3 Other general options

'`-M`' , '`--cache-size megs`'

Memory in megabytes used for caching of read results. The default size is 8 unless you configure gnugo with the command `configure --enable-cache-size=size` before compiling to make size the default (see section Installation). GNU Go stores results of its reading calculations in a hash table (see section Hashing of Positions). 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 you may want to increase the size of the 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) or if either player passes during the game.

'`--japanese-rules`'

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

'`--play-out-aftermath`'
'`--capture-all-dead`'

These options cause GNU Go to play out moves that are usually left unplayed after the end of the game. Such moves lose points under Japanese rules but not Chinese rules. With '`--play-out-aftermath`', GNU Go may play inside its territory in order to reach a position where it considers every group demonstrably alive or dead. The option '`--capture-all-dead`' causes GNU Go to play inside its own territory to remove dead stones.

'`--forbid-suicide`'

Do not allow suicide moves (playing a stone so that it ends up without liberties and is therefore immediately removed). This is the default.

'`--allow-suicide`'

Allow suicide moves, except single-stone suicide. The latter would not change the board at all and pass should be used instead.

'`--allow-all-suicide`'

Allow suicide moves, including single-stone suicide. This is only interesting in exceptional cases. Normally the '`--allow-suicide`' option should be used instead.

'`--simple-ko`'

Do not allow an immediate recapture of a ko so that the previous position is recreated. Repetition of earlier positions than that are allowed. This is default.

'`--no-ko`'

Allow all kinds of board repetition.

'`--positional-superko`'

Forbid repetition of any earlier board position. This only applies to moves on the board; passing is always allowed.

'`--situational-superko`'

Forbid repetition of any earlier board position with the same player to move. This only applies to moves on the board; passing is always allowed.

'`--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`' and '`-L`' to create a diagram of the board from 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.

'`--options`'

Print which experimental configure options were compiled into the program (see section Other Options).

'`--orientation n`'

Combine with '`-l`'. The Go board can be oriented in 8 different ways, counting reflections and rotations of the position; this option selects an orientation (default 0). The parameter '`n`' is an integer between 0 and 7.

3.9.4 Other options affecting strength and speed

- '`--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.

This single parameter '`--level`' is the best way of choosing whether to play stronger or faster. It controls a host of other parameters which may themselves 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 the remaining options in this category. Instead, just adjust the single variable '`--level`'. The following options are of use to developers tuning the program for performance and accuracy. For completeness, here they are.

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

  Deep reading cutoff. When reading beyond this depth (default 16) 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.

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

  Deep reading cutoff. Beyond this depth (default 12) 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 7) 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`'

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. Below this depth (default 13), GNU Go still tries to attack strings with only 3 liberties, but only tries one move at each node.

- '`--break-chain-depth depth`'

  Set the `break_chain_depth`. Beyond this depth, GNU Go abandons some attempts to defend groups by trying to capture part of the surrounding chain.

- '`--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 Superstrings) 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 Reading Basics).

- '`--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 1000. We caution the user that increasing `owl_node_limit` does not necessarily increase the strength of the program.

- '`--owl-node-limit n`'

  If the number of variations exceeds this limit, Owl assumes the dragon can make life. Default 1000. We caution the user that increasing `owl_node_limit` does not necessarily increase the strength of the program.

- '`--owl-distrust n`'

  Below this limit some owl reading is truncated.

3.9.5 Ascii mode options

'`--color color`'

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

'`--handicap number`'

Choose the number of handicap stones (0–9)

For more information about the following clock options see See section Ascii Interface.

'`--clock seconds`'

Initialize the timer.

'`--byo-time seconds`'

Number of seconds per (Canadian) byo-yomi period

'`--byo-period stones`'

Number of stones per (Canadian) byo-yomi period

3.9.6 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 a speedup or other optimization has not affected the behavior of the engine. Note that when 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 section Regression testing.

'-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 Colored Display)

'--showtime'

Print timing information to stderr.

'-E' , '--printeyes' : colored display of eye spaces

Use rxvt, xterm or Linux Console. (see section Colored Display)

'-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' . A list of these may be produced using '--debug-flags' . Here they are in hexadecimal:

```
DEBUG_INFLUENCE             0x0001
DEBUG_EYES                  0x0002
DEBUG_OWL                   0x0004
DEBUG_ESCAPE                0x0008
DEBUG_MATCHER               0x0010
DEBUG_DRAGONS               0x0020
DEBUG_SEMEAI                0x0040
DEBUG_LOADSGF               0x0080
DEBUG_HELPER                0x0100
DEBUG_READING               0x0200
DEBUG_WORMS                 0x0400
DEBUG_MOVE_REASONS          0x0800
DEBUG_OWL_PERFORMANCE       0x1000
DEBUG_LIFE                  0x2000
DEBUG_FILLLIB               0x4000
DEBUG_READING_PERFORMANCE   0x8000
DEBUG_SCORING               0x010000
DEBUG_AFTERMATH             0x020000
DEBUG_ATARI_ATARI           0x040000
DEBUG_READING_CACHE         0x080000
DEBUG_TERRITORY             0x100000
DEBUG_OWL_PERSISTENT_CACHE  0X200000
DEBUG_TOP_MOVES             0x400000
DEBUG_MISCELLANEOUS         0x800000
DEBUG_ORACLE_STREAM         0x1000000
```

These debug flags are additive. If you want to turn on both dragon and worm debugging you can use '-d0x420' .

'--debug-flags'

Print the list of debug flags

'-w' , '--worms'

Print more information about worm data.

'-m' , '--moyo level'

moyo debugging, show moyo board. The level is fully documented elsewhere (see section Colored display and debugging of influence).

'-b' , '--benchmark number'

benchmarking mode - can be used with '-l' . Causes GNU Go to play itself repeatedly, seeding the start of the game with a few random moves. This method of testing the program is largely superceded by use of the `twogtp` program.

'-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 section Tactical reading 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-owl location'

Invoke the owl code (see section The Owl Code) 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.

'`--decide-connection location1/location2`'

Decide whether dragons at location1 and location2 can be connected. Useful in connection with '`-o`' to write the variations to an SGF file.

'`--decide-dragon-data location`'

Print complete information about the status of the dragon at location.

'`--decide-semeai location1/location2`'

At location1 and location2 are adjacent dragons of the opposite color. Neither is aliveby itself, and their fate (alive, dead or seki) depends on the outcome of a semeai (capturing race). Decide what happens. Useful in connection with '`-o`' to write the variations to an SGF file.

'`--decide-tactical-semeai location1/location2`'

Similar to '`--decide-semeai`', except that moves proposed by the owl code are not considered.

'`--decide-position`'

Try to attack and defend every dragon with dragon.escape<6. If used with '`-o`', writes the variations to an sgf file.

'`--decide-eye location`'

Evaluates the eyespace at location and prints a report. You can get more information by adding '`-d0x02`' to the command line. (see section Eye Local Game Values.)

'`--decide-surrounded location`'

A dragon is surrounded if it is contained in the convex hull of its unfriendly neighbor dragons. This does not mean that it cannot escape, but it is often a good indicator that the dragon is under attack. This option draws the convex hull of the neighbor dragons and decides whether the dragon at location is surrounded.

'`--decide-combination`'

Calls the function `atari_atari` to decide whether there exist combinations on the board.

'`--score method`'

Requires '`-l`' to specify which game to score and '`-L`' if you want to score anywhere else than at the end of the game record. method can be "estimate", "finish", or "aftermath". "finish" and "aftermath" are appropriate when the game is complete, or nearly so, and both try to supply an accurate final score. Notice that if the game is not already finished it will be played out, which may take quite a long time if the game is far from complete. The "estimate" method may be used to get a quick estimate during the middle of the game. Any of these options may be combined with '`--chinese-rules`' if you want to use Chinese (Area) counting.

If the option '`-o outputfilename`' is provided, the result will also be written as a comment in the output file. For the "finish" and "aftermath" scoring algorithms, the selfplayed moves completing the game are also stored.

finish

Finish the game by selfplaying until two passes, then determine the status of all stones and compute territory.

aftermath

Finish the game by selfplaying until two passes, then accurately determine status of all stones by playing out the "aftermath", i.e. playing on until all stones except ones involved in seki have become either unconditionally (in the strongest sense) alive or unconditionally dead (or captured). Slower than '`--score finish`', and while these algorithms usually agree, if they differ, '`--score aftermath`' is most likely to be correct.

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

This combination mandates **Tromp-Taylor** scoring. The Tromp-Taylor ruleset requires the game to be played out until all dead stones are removed, then uses area (Chinese) scoring. The option '`--capture-all-dead`' requires the aftermath code to finish capturing all dead stones.

3.9.7 Experimental options

Most of these are available as configure options and are described in Other Options.

- '`--options`'

  Print which experimental configure options were compiled into the program.

- '`--with-break-in`'
- '`--without-break-in`'

  Use or do not use the experimental break-in code. This option has no effect at level 9 or below. The break in code is enabled by default at level 10, and the only difference between levels 9 and level 10 is that the break in code is disabled at level 9.

- '`--cosmic-gnugo`'

  Use center oriented influence.

- '`--nofusekidb`'

  Turn off the fuseki database.

- '`--nofuseki`'

Turn off fuseki moves entirely

- '`--nojosekidb`'

  Turn off the joseki database.

- '`--mirror`'

  Try to play mirror go.

- '`--mirror-limit n`'

  Stop mirroring when n stones are on the board.

[<<] [>>] [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

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.

GNU Go starts by trying to understand the current board position as good as possible. Using the information found in this first phase, and using additional move generators, a list of candidate moves is generated. Finally, each of the candidate moves is valued according to its territorial value (including captures or life-and-death effects), and possible strategical effects (such as strengthening a weak group).

Note that while GNU Go does, of course, do a lot of reading to analyze possible captures, life and death of groups etc., it does not (yet) have a fullboard lookahead.

4.1 Gathering Information

This is by far the most important phase in the move generation. Misunderstanding life-and-death situations can cause gross mistakes. Wrong territory estimates will lead to inaccurate move valuations. Bad judgement of weaknesses of groups make strategic mistakes likely.

This information gathering is done by the function `examine_position()`. It first calls `make_worms()`.

Its first steps are very simple: it identifies sets of directly connected stones, called worms, and notes their sizes and their number of liberties.

Soon after comes the most important step of the worm analysis: the tactical reading code (see section Tactical reading) is called for every worm. It tries to read out which worms can be captured directly, giving up as soon as a worm can reach 5 liberties. If a worm can be captured, the engine of course looks for moves defending against this capture. Also, a lot of effort is made to find virtually all moves that achieve the capture or defense of a worm.

After knowing which worms are tactically stable, we can make a first picture of the balance of power across the board: the Influence Function code is called for the first time.

This is to aid the next step, the analysis of dragons. By a dragon we mean a group of stones that cannot be disconnected.

Naturally the first step in the responsible function `make_dragons()` is to identify these dragons, i.e. determine which worms cannot be disconnected from each other. This is partly done by patterns, but in most cases the specialized readconnect code is called. This module does a minimax search to determine whether two given worms can be connected with, resp. disconnected from each other.

Then we compute various measures to determine how strong or weak any given dragon is:

- A crude estimate of the number of eyes is made.
- The results of the influence computations is used to see which dragons are adjacent to own territory or a moyo.
- A guess is made for the potential to escape if the dragon got under attack.

For those dragons that are considered weak, a life and death analysis is made (see section The Owl Code). If two dragons next to each other are found that are both not alive, we try to resolve this situation with the semeai module.

For a more detailed reference of the worm and dragon analysis (and explanations of the data structures used to store the information), see See section Worms and Dragons.

The influence code is then called second time to make a detailed analysis of likely territory. Of course, the life-and-death status of dragons are now taken into account.

The territorial results of the influence module get corrected by the break-in module. This specifically tries to analyze where an opponent could break into an alleged territory, with sequences that would be too difficult to see for the influence code.

4.2 Move Generators

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.

For a list and explanation of move reasons used in GNU Go, and how they are evaluated, see See section Move generation.

There are a couple of move generators that only extract data found in the previous phase, examining the position:

- `worm_reasons()`

  Moves that have been found to capture or defend a worm are proposed as candidates.

- `owl_reasons()`

  The status of every dragon, as it has been determined by the owl code (see section The Owl Code) in the previous phase, is reviewed. If the status is critical, the killing or defending move gets a corresponding move reason.

- `semeai_move_reasons()`

  Similarly as `owl_reasons`, this function proposes moves relevant for semeais.

- `break_in_move_reasons()`

  This suggests moves that have been found to break into opponent's territory by the break-in module.

The following move generators do additional work:

- `fuseki()`

  Generate a move in the early fuseki, either in an empty corner of from the fuseki database.

- `shapes()`

  This is probably the most important move generator. It finds patterns from 'patterns/patterns.db', 'patterns/patterns2.db', 'patterns/fuseki.db', and the joseki files 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 section The Pattern Code, for a complete documentation of patterns.)

- `combinations()`

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

- `revise_thrashing_dragon()`

  This module does not directly propose move: If we are clearly ahead, and the last move played by the opponent is part of a dead dragon, we want to attack that dragon again to be on the safe side. This is done be setting the status of this thrashing dragon to unkown and repeating the shape move generation and move valution.

- `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.

4.3 Move Valuation

After the move generation modules have run, each proposed candidate move goes through a detailed valuation by the function `review_move_reasons`. This invokes some analysis to try to turn up other move reasons that may have been missed.

The most important value of a move is its territorial effect. see section Influence and Territory explains in detail how this is determined.

This value is modified for all move reasons that cannot be expressed directly in terms of territory, such as combination attacks (where it is not clear which of several strings will get captured), strategical effects, connection moves, etc. A large set heuristics is necessary here, e.g. to avoid duplication of such values. This is explained in more detail in Valuation of suggested moves.

4.4 Detailed Sequence of Events

First comes the sequence of events when `examine_position()` is run from `genmove()`. This is for reference only.

```
purge_persistent_caches()
make_worms():
  compute_effective_sizes()
  compute_unconditional_status()
  find_worm_attacks_and_defenses():
    for each attackable worm:
      set worm.attack
      change_attack() to add the attack point
    find_attack_patterns() to find a few more attacks
    for each defensible worm:
      set worm.attack
      change_defense() to add the defense point
    find_defense_patterns() to find a few more defense moves
    find additional attacks and defenses by testing all
      immediate liberties
  find higher order liberties (for each worm)
  find cutting stones (for each worm)
  improve attacks and defenses: if capturing a string defends
    another friendly string, or kills an unfriendly one, we
    add points of defense or attack. Make repairs if adjacent
    strings can both be attacked but not defended.
  find worm lunches
  find worm threats
```

```
    identify inessential worms (such as nakade stones)
  compute_worm_influence():
    find_influence_patterns()
    value_influence()
    segment_influence()
  make_dragons():
    find_cuts()
    find_connections()
    make_domains() (determine eyeshapes)
    find_lunches() (adjacent strings that can be captured)
    find_half_and_false_eyes()
    eye_computations(): Compute the value of each eye space.
      Store its attack and defense point.
    analyze_false_eye_territory()
    for each dragon compute_dragon_genus()
    for each dragon compute_escape() and set escape route data
    resegment_initial_influence()
    compute_refined_dragon_weaknesses() (called again after owl)
    for each dragon compute_crude_status()
    find_neighbor_dragons()
    for each dragon compute surround status
    for each weak dragon run owl_attack() and owl_defend()
      to determine points of attack and defense
    for each dragon compute dragon.status
    for each thrashing dragon compute owl threats
    for each dragon compute dragon.safety
    revise_inessentiality()
    semeai():
      for every semeai, run owl_analyze_semeai()
      find_moves_to_make_seki()
    identify_thrashing_dragons()
    compute_dragon_influence():
      compute_influence()
      break_territories() (see section Break Ins)
    compute_refined_dragon_weaknesses()
```

Now a summary of the sequence of events during the move generation and selection phases of `genmove()`, which take place after the information gathering phase has been completed:

```
  estimate_score()
  choose_strategy()
  collect_move_reasons():
    worm_reasons(): for each attack and defense point add a move reason
    semeai_reasons(): for each dragon2.semeai point add a move reason
    owl_reasons(): for each owl attack and defense point add a move reason
    break_in_reasons(): for each breakin found add a move reason
  fuseki()
  break_mirror_go()
  shapes(): match patterns around the board (see section Overview)
  combinations(): look for moves with a double meaning and other tricks
    find_double_threats()
    atari_atari()
  review_move_reasons()
  if ahead and there is a thrashing dragon, consider it
    alive and reconsider the position
  endgame_shapes()
  endgame()
  if no move found yet, revisit any semeai, change status of dead opponent
    to alive, then run shapes() and endgame_shapes() again
  if no move found yet, run fill_liberty()
```

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 Game Format files, and testing are found in the directories ‘`interface/`’, ‘`sgf/`’, and ‘`regression/`’. Code borrowed from other GNU programs is contained in ‘`utils/`’. That directory also includes some code developed within GNU Go which is not go specific. 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.

- ‘`breakin.c`’

  Code to detect moves which can break into supposed territory and moves to prevent this.

- ‘`cache.c`’ and ‘`cache.h`’

  As a means of speeding up reading, computed results are cached so that they can be quickly reused if the same position is encountered through e.g. another move ordering. This is implemented using a hash table.

- ‘`clock.c`’ and ‘`clock.h`’

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

- ‘`combination.c`’

  When something can (only) be captured through a series of ataris or other threats we call this a combination attack. This file contains code to find such attacks and moves to prevent them.

- ‘`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.

- ‘`endgame.c`’

  Code to find certain types of endgame moves.

- ‘`filllib.c`’

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

- ‘`fuseki.c`’

  Generates fuseki (opening) moves from a database. Also generates moves in empty corners.

- ‘`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 ‘`hash.h`’

  Hashing code implementing Zobrist hashing. (see section Hashing of Positions) The code in ‘`hash.c`’ provides a way to hash board positions into compact descriptions which can be efficiently compared. The caching code in ‘`cache.c`’ makes use of the board hashes when storing and retrieving read results.

- ‘`influence.c`’ and ‘`influence.h`’.

  This code determines which regions of the board are under the influence of either player. (see section Influence Function)

- ‘`liberty.h`’

  Header file for the engine. The name “liberty” connotes freedom (see section Copying).

- ‘`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`’ and ‘`move_reasons.h`’

  Code for keeping track of move reasons.

- ‘`movelist.c`’

  Supporting code for lists of moves.

* 'optics.c'

  This file contains the code to recognize eye shapes, documented in See section Eyes and Half Eyes.

* 'oracle.c'

  Code to fork off a second GNU Go process which can be used to simulate reading with top level information (e.g. dragon partitioning) available.

* 'owl.c'

  This file does life and death reading. Move generation is pattern based and the code in 'optics.c' is used to evaluate the eyespaces for vital moves and independent life. A dragon can also live by successfully escaping. Semeai reading along the same principles is also implemented in this file.

* 'persistent.c'

  Persistent cache which allows reuse of read results at a later move or with additional stones outside an active area, which are those intersections thought to affect the read result.

* 'printutils.c'

  Print utilities.

* 'readconnect.c' and 'readconnect.h'

  This file contains code to determine whether two strings can be connected or disconnected.

* 'reading.c'

  This file contains code to determine whether any given string can be attacked or defended. See section Tactical reading, for details.

* 'semeai.c'

  This file contains `semeai()`, the module which detects dragons in semeai. To determine the semeai results the semeai reading in 'owl.c' is used.

* 'sgfdecide.c'

  Code to generate sgf traces for various types of reading.

* 'shapes.c'

  This file contains `shapes()`, the module called by `genmove()` which tries to find moves which match a pattern (see section The Pattern Code).

* '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.

* 'surround.c'

  Code to determine whether a dragon is surrounded and to find moves to surround with or break out with.

* 'utils.c'

  An assortment of utilities, described in greater detail below.

* 'value_moves.c'

  This file contains the code which assigns values to every move after all the move reasons are generated. It also tries to generate certain kinds of additional move reasons.

* '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

---

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 The Owl Code).
* Connection patterns in 'conn.db' (see section The Connections Database)
* Influence patterns in 'influence.db' and 'barriers.db' (see section Influence Function)
* eye patterns in 'eyes.db' (see section Eyes and Half Eyes).

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 Game 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 section Eyes and Half Eyes, for details.

- ‘helpers.c’

  These are helper functions to assist in evaluating moves by matchpat.

- ‘hoshi.sgf’

  Smart Game 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 Game Format, and produces a pattern database.

- ‘komoku.sgf’

  Smart Game 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 Game Format file containing 5-3 point openings

- ‘mokuhazushi.db’

  Pattern database compiled from mokuhadzushi.sgf

- ‘sansan.sgf’

  Smart Game Format file containing 3-3 point openings

- ‘sansan.db’

  Pattern database compiled from ‘sansan.sgf’

- ‘takamoku.sgf’

  Smart Game 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.

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-3.6/`’ and execute the command:

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

This will build a file called ‘`gnugo-3.6/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-3.6/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.

In practice, most tuning of GNU Go is done in conjunction with maintaining the ‘regression/’ directory (see section Regression testing).

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 3.6, 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 Tuning the Pattern databases) 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 worms and dragons around the board. This type of information is available by different methods, however (see section Debugging on a Graphical Board, see section Colored Display).

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 section Tactical reading). 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. The data in parentheses tell 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 section The Go Text Protocol) 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 Debugging the reading code. 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 Debugging on a Graphical Board

The quickest way to analyze most positions is to use the tool ‘`view.pike`’ in the ‘`regression`’ directory. It can be started with a testcase specified, e.g. `pike view.pike strategy:40` or at a move in an sgf file, e.g. `pike view.pike mistake.sgf:125`. When started it shows the position on a grapical board on which it also marks information like move values, dragon status, and so on. By clicking on the board further information about the valuation of moves, contents of various data structures, and other data can be made available.

Specific information on how to use ‘`view.pike`’ for influence tuning can be found in See section Influence Tuning with `view.pike`.

5.7 Scoring the game

GNU Go can score the game. 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 (see section Invoking GNU Go: Command line options).

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 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`. On Mac OS X put `setenv TERM xterm-color` in your ‘`.tcshrc`’ file to enable color in the terminal.

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 Dragons) 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 section Eyes and Half Eyes).

6. Move generation

6.1 Introduction

GNU Go 3.0 introduced a move generation scheme substantially different from earlier versions. In particular, it was 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.

6.2 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. Here is a partial list of of move reasons considered by GNU Go. (The complete list may be found in 'move_reasons.h'.)

ATTACK_MOVE
DEFEND_MOVE

Attack or defend a worm.

ATTACK_THREAT_MOVE
DEFEND_THREAT_MOVE

Threaten to attack or defend a worm.

EITHER_MOVE

A move that either achieves one goal or another (at the moment this only used for attacks on worms).

ALL_MOVE

At the moment this is used for a move that defends two worms threatened by a double attack.

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
EXPAND_MOYO_MOVE

Move expanding our territory/moyo. These reasons are at the moment treated identically.

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 Pattern Attributes) 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. This means that the owl code could not decide the outcome, because the owl node limit was reached.

`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.

The attack and defend move types can have a suffix to denote moves whose result depends on a ko, e.g. `OWL_ATTACK_MOVE_GOOD_KO`. Here `..._GOOD_KO` and `..._BAD_KO` correspond to `KO_A` and `KO_B` as explained in Ko Handling. See ‘`engine/move_reasons.h`’ for the full of move reasons.

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

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

6.3 Detailed Descriptions of various Move Reasons

6.3.1 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.)

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.3.5 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.

6.3.6 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.)

6.3.7 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.

6.3.8 Territorial moves

Any move that increases territory gets a move reason. This is the expand territory move reason. That move reason is added by the ‘`e`’ patterns in ‘`patterns/patterns.db`’. Similarly the ‘`E`’ patterns attempt to generate or mitigate a moyo, which is a region of influence not yet secure territory, yet valuable. Such a pattern sets the “expand moyo” move reason.

6.3.9 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[pos].owl_attack_point` and `dragon[pos].owl_second_attack_point`.

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.

6.3.10 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`. Once such a sequence of ataris is found, it tries to shorten it by rejecting irrelevant moves.

6.4 Valuation of suggested moves

At the end of the move generation process, the function `value_move_reasons()` tries to assign values to the moves for the purpose of selecting the best move. The single purpose of the move valuation is to try to rank the moves so that the best move gets the highest score. In principle these values could be arbitrary, but in order to make it easier to evaluate how well the valuation performs, not to mention simplify the tuning, we try to assign values which are consistent with the usual methods of counting used by human Go players, as explained for example in The Endgame by Ogawa and Davies.

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

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

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

The territorial value is the total change of expected territory caused by this move. This includes changes in the status of groups if the move is an attack or a defense move.

Beginning with GNU Go 3.0, the influence function plays an important role in estimating territory (see section Influence and Territory). It is used to make a guess at each intersection how likely it is that it will become black or white territory. The territorial value sums up the changes in these valuations.

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. An important role of this measure is to offset mistakes made by the estimation of territorial 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.

6.4.1 Territorial Value

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

It considers all groups that are changed from alive to death or vice-versa due to this move. Also, it makes an assumption whether the move should be considered safe. If so, the influence module is called: The function `influence_delta_territory` estimates the territorial effect of both the stone played and of the changes of group status'.

The result returned by the influence module is subject to a number of corrections. This is because some move reasons cannot be evaluated by a single call to the influence function, such as moves depending on a ko.

6.4.2 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.

6.4.3 Shape Factor

In the value field of a pattern (see section Pattern Attributes) 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.

6.4.4 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.

In the fuseki, there are often several moves with identical minimum value. GNU Go chooses randomly between such moves, which ensures some indeterminacy of GNU Go's play. Later in the game, GNU Go's genuine valuation of such a move is used as a secondary criterion.

6.4.5 Secondary Value

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

6.4.6 Threats and Followup Value

Followup value refers to value which may acrue if we get two moves in a row in a local area. It is assigned for moves that threaten to attack or defend a worm or dragon. Also, since GNU Go 3.2 the influence module makes an assessment of the possible purely territorial followup moves. In cases where these two heuristics are not sufficient we add patterns with a `followup_value` autohelper macro.

Usually, the followup value gives only a small contribution; e.g. if it the followup value is very large, then GNU Go treats the move as sente by doubling its value. However, 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 full followup value is taken into account.

6.5 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.

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

7. 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 section Eyes and Half Eyes.

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 stones on the board which are connected along the horizontal and vertical lines, and are of the same color. We often say string instead of worm.

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.

7.1 Worms

The array `struct worm_data worm[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 origin;
  int liberties;
  int liberties2;
  int liberties3;
  int liberties4;
  int lunch;
  int cutstone;
  int cutstone2;
  int genus;
  int inessential;
  int invincible;
  int unconditional_status;
  int attack_points[MAX_TACTICAL_POINTS];
  int attack_codes[MAX_TACTICAL_POINTS];
  int defense_points[MAX_TACTICAL_POINTS];
  int defend_codes[MAX_TACTICAL_POINTS];
  int attack_threat_points[MAX_TACTICAL_POINTS];
  int attack_threat_codes[MAX_TACTICAL_POINTS];
  int defense_threat_points[MAX_TACTICAL_POINTS];
  int defense_threat_codes[MAX_TACTICAL_POINTS];
};
```

- `color`

  The color of the worm.

- `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.

- `origin`

  Each worm has a distinguished member, called its origin. 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[pos].origin == pos.
  ```

- `liberties`
- `liberties2`
- `liberties3`
- `liberties4`

  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 empty space. 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.

- `lunch`

  If nonzero, `lunch` points to a boundary worm which can be easily captured. (It does not matter whether or not the string can be defended.)

We have two distinct notions of cutting stone, which we keep track of in the separate fields `worm.cutstone` and `worm.cutstone2`. We use currently use both concepts in parallel.

`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[].cutstone=2`. For potential cutting strings we set `worm[].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()`.

`genus`

  There are two separate notions of genus for worms and dragons. The dragon notion is more important, so `dragon[pos].genus` is a far more useful field than `worm[pos].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.

`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 remaining cavity only borders worms of the opposite color.

`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_TERRITORY` or `BLACK_TERRITORY` 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.

`int attack_points[MAX_TACTICAL_POINTS]`

`attack_codes[MAX_TACTICAL_POINTS]`

```
int defense_points[MAX_TACTICAL_POINTS];
int defend_codes[MAX_TACTICAL_POINTS];
```

If the tactical reading code (see section Tactical reading) finds that the worm can be attacked, `attack_points[0]` is a point of attack, and `attack_codes[0]` is the attack code, `WIN`, `KO_A` or `KO_B`. If multiple attacks are known, `attack_points[k]` and `attack_codes[k]` are used. Similarly with the defense codes and defense points.

`int attack_threat_points[MAX_TACTICAL_POINTS];`

`int attack_threat_codes[MAX_TACTICAL_POINTS];`

```
int defense_threat_points[MAX_TACTICAL_POINTS];
int defense_threat_codes[MAX_TACTICAL_POINTS];
```

These are points that threaten to attack or defend a worm.

The function `makeworms()` will generate data for all worms.

7.2 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 worms in GNU Go 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`’.

7.3 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.

7.4 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 Topology of Half Eyes and False Eyes).

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; the fields `attack_point[]` point to up to 4 points to attack the half eye, and similarly `defense_point[]` gives points to defend the half eye.

```
struct half_eye_data half_eye[MAX_BOARD];

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

The array `struct half_eye_data half_eye[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.

7.5 Dragons

The array `struct dragon_data dragon[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. (Fields will be discussed below.)

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

extern struct dragon_data dragon[BOARDMAX];
```

Other fields attached to the dragon are contained in the `dragon_data2` struct array. (Fields will be discussed below.)

```
struct dragon_data2 {
  int origin;
  int adjacent[MAX_NEIGHBOR_DRAGONS];
  int neighbors;
  int hostile_neighbors;
  int moyo_size;
  float moyo_territorial_value;
  int safety;
  float weakness;
  float weakness_pre_owl;
  int escape_route;
  struct eyevalue genus;
  int heye;
  int lunch;
  int surround_status;
  int surround_size;
  int semeais;
  int semeai_margin_of_safety;
  int semeai_defense_point;
  int semeai_defense_certain;
  int semeai_attack_point;
  int semeai_attack_certain;
  int owl_threat_status;
  int owl_status;
  int owl_attack_point;
  int owl_attack_code;
  int owl_attack_certain;
  int owl_second_attack_point;
  int owl_defense_point;
  int owl_defense_code;
  int owl_defense_certain;
  int owl_second_defense_point;
  int owl_attack_kworm;
  int owl_defense_kworm;
};

extern struct dragon_data2 *dragon2;
```

The difference between the two arrays is that the `dragon` array is indexed by the board, and there is a copy of the dragon data at every stone in the dragon, while there is only one copy of the dragon2 data. The dragons are numbered, and the `id` field of the dragon is a key into the dragon2 array. Two macros DRAGON and DRAGON2 are provided for gaining access to the two arrays.

```
#define DRAGON2(pos) dragon2[dragon[pos].id]
#define DRAGON(d) dragon[dragon2[d].origin]
```

Thus if you know the position `pos` of a stone in the dragon you can access the dragon array directly, for example accessing the origin with `dragon[pos].origin`. However if you need a field from the dragon2 array, you can access it using the DRAGON2 macro, for example you can access its neighor dragons by

```
for (k = 0; k < DRAGON2(pos).neighbors; k++) {
  int d = DRAGON2(pos).adjacent[k];
  int apos = dragon2[d].origin;
  do_something(apos);
}
```

Similarly if you know the dragon number (which is `dragon[pos].id`) then you can access the `dragon2` array directly, or you can access the `dragon` array using the DRAGON macro.

Here are the definitions of each field in the `dragon` arrray.

- `color`

  The color of the dragon.

- `id`

  The dragon number, used as a key into the `dragon2` array.

- origin

  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.

- size

  The number of stones in 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.

- crude_status

  (ALIVE, DEAD, UNKNOWN, CRITICAL). An early measure of the life potential of the dragon. It is computed before the owl code is run and is superceded by the status as soon as that becomes available.

- status

  The dragon status is the best measure of the dragon's health. It is computed after the owl code is run, then revised again when the semeai code is run.

Here are definitions of the fields in the `dragon2` array.

- origin

  The origin field is duplicated here.

- adjacent
- `adjacent[MAX_NEIGHBOR_DRAGONS]`

  Dragons of either color near the given one are called neighbors. They are computed by the function `find_neighbor_dragons()`. The `dragon2.adjacent` array gives the dragon numbers of these dragons.

- `neighbors`

  Dragons of either color near the given one are called neighbors. They are computed by the function `find_neighbor_dragons()`. The `dragon2.adjacent` array gives the dragon numbers of these dragons.

- neighbors

  The number of neighbor dragons.

- hostile_neighbors

  The number of neighbor dragons of the opposite color.

- moyo_size
- float moyo_territorial_value

  The function `compute_surrounding_moyo_sizes()` assigns a size and a territorial value to the moyo around each dragon (see section Territory, Moyo and Area). This is the moyo size. They are recorded in these fields.

- safety

  The dragon safety can take on one of the values

      - TACTICALLY_DEAD - a dragon consisting of a single worm found dead by the reading code (very reliable)
      - ALIVE - found alive by the owl or semeai code
      - STRONGLY_ALIVE - alive without much question
      - INVINCIBLE - definitively alive even after many tenukis
      - ALIVE_IN_SEKI - determined to be seki by the semeai code
      - CRITICAL - lives or dies depending on who moves first
      - DEAD - found to be dead by the owl code
      - INESSENTIAL - the dragon is unimportant (e.g. nakade stones) and dead

- weakness
- weakness_pre_owl

  A floating point measure of the safety of a dragon. The dragon weakness is a number between 0. and 1., higher numbers for dragons in greater need of safety. The field `weakness_pre_owl` is a preliminary computation before the owl code is run.

- escape_route

  A measure of the dragon's potential to escape towards safety, in case it cannot make two eyes locally. Documentation may be found in Escape.

- struct eyevalue genus

  The approximate number of eyes the dragon can be expected to get. Not guaranteed to be accurate. The eyevalue struct, which is used throughout the engine, is declared thus:

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

- heye

  Location of a half eye attached to the dragon.

- lunch

  If nonzero, this is the location of a boundary string which can be captured. In contrast with worm lunches, a dragon lunch must be able to defend itself.

- surround_status
- surround_size

  In estimating the safety of a dragon it is useful to know if it is surrounded. See Surrounded Dragons and the comments in ‘`surround.c`’ for more information about the algorithm. Used in computing the escape_route, and also callable from patterns (currently used by CB258).

- semeais
- semeai_defense_point
- semeai_defense_certain
- semeai_attack_point
- semeai_attack_certain

  If two dragons of opposite color both have the status CRITICAL or DEAD they are in a semeai (capturing race), and their status must be adjudicated by the function `owl_analyze_semeai()` in 'owl.c', which attempts to determine which is alive, which dead, or if the result is seki, and whether it is important who moves first. The function 'new_semeai()' in 'semeai.c' attempts to revise the statuses and to generate move reasons based on these results. The field `dragon2.semeais` is nonzero if the dragon is an element of a semeai, and equals the number of semeais (seldom more than one). The semeai defense and attack points are locations the defender or attacker must move to win the semeai. The field `semeai_margin_of_safety` is intended to indicate whether the semeai is close or not but currently this field is not maintained. The fields `semeai_defense_certain` and `semeai_attack_certain` indicate that the semeai code was able to finish analysis without running out of nodes.

- owl_status

  This is a classification similar to `dragon.crude_status`, but based on the life and death reading in 'owl.c'. The owl code (see section The Owl Code) is skipped for dragons which appear safe by certain heuristics. If the owl code is not run, 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_attack_point

  If the dragon can be attacked this is the point to attack the dragon.

- owl_attack_code

  The owl attack code, It can be WIN, KO_A, KO_B or 0 (see Return Codes).

- owl_attack_certain

  The owl reading is able to finish analyzing the attack without running out of nodes.

- owl_second_attack_point

  A second attack point.

- owl_defense_point

  If the dragon can be defended, this is the place to play.

- owl_defense_code

  The owl defense code, It can be WIN, KO_A, KO_B or 0 (see Return Codes).

- owl_defense_certain

  The owl code is able to finish analyzing the defense without running out of nodes.

- owl_second_defense_point

  A second owl defense point.

7.6 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 section Eyes and Half Eyes).
- the option -m 0x0180 to display territory, moyo and area (see section Territory, Moyo and Area).

The colored displays are documented elsewhere (see section Colored Display).

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

8. Eyes and Half Eyes

The purpose of this Chapter is to describe the algorithm used in GNU Go to determine eyes.

8.1 Local games

The fundamental paradigm of combinatorial game theory is that games can be added and in fact form a group. If 'G' and 'H' are games, then 'G+H' is a game in which each player on his turn has the option of playing in either move. We say that the game 'G+H' is the sum of the local games 'G' and 'H'.

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
```

8.2 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 Amalgamation). 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 ‘`optics.c`’ .

The eyespaces can be conveniently displayed using a colored ascii diagram by running `gnugo -E`.

8.3 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 548

  x
 xX.!

 :0111
```

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 model.

```
  |- 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.

8.4 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!


. !
.!
```

8.5 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 identified by the topological analysis (see section Topology of Half Eyes and False Eyes).

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
```

```
XXXXXXX
```

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 Topology of Half Eyes and False Eyes).

The graph of the eye_shape, ostensibly '....' is modified by replacing the left '.' by '!.' during graph matching.

A false eye is isomorphic to the pattern (!) . To see this, consider the following eye shape:

```
XXXOOOOOO
X.Oa....O
XXXOOOOOO
```

This is equivalent to the two previous eyeshapes, with an isomorphic local game {2|1}.

This eyeshape has a false eye at 'a'. This is also found by the topological criterion.

The graph of the eye_shape, ostensibly '.....' is modified by replacing the left '.' by '!'. This is made directly in the eye data, not only during graph matching.

8.6 Eye shape analysis

The patterns in 'patterns/eyes.db' are compiled into graphs represented essentially by arrays 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}.

8.7 Eye Local Game Values

The game values in 'eyes.db' are given in a simplified scheme which is flexible enough to represent most possibilities in a useful way.

The colon line below the pattern gives the eye value of the matched eye shape. This consists of four digits, each of which is the number of eyes obtained during the following conditions:

1. The attacker moves first and is allowed yet another move because the defender plays tenuki.
2. The attacker moves first and the defender responds locally.
3. The defender moves first and the attacker responds locally.
4. The defender moves first and is allowed yet another move because the attacker plays tenuki.

The first case does **not** necessarily mean that the attacker is allowed two consecutive moves. This is explained with an example later.

Also, since two eyes suffice to live, all higher numbers also count as two.

The following 15 cases are of interest:

- 0000 0 eyes.
- 0001 0 eyes, but the defender can threaten to make one eye.
- 0002 0 eyes, but the defender can threaten to make two eyes.
- 0011 1/2 eye, 1 eye if defender moves first, 0 eyes if attacker does.
- 0012 3/4 eyes, 3/2 eyes if defender moves first, 0 eyes if attacker does.
- 0022 1* eye, 2 eyes if defender moves first, 0 eyes if attacker does.
- 0111 1 eye, attacker can threaten to destroy the eye.
- 0112 1 eye, attacker can threaten to destroy the eye, defender can threaten to make another eye.
- 0122 5/4 eyes, 2 eyes if defender moves first, 1/2 eye if attacker does.
- 0222 2 eyes, attacker can threaten to destroy both.
- 1111 1 eye.
- 1112 1 eye, defender can threaten to make another eye.
- 1122 3/2 eyes, 2 eyes if defender moves first, 1 eye if attacker does.
- 1222 2 eyes, attacker can threaten to destroy one eye.

- 2222 2 eyes.

The 3/4, 5/4, and 1* eye values are the same as in Howard Landman's paper Eyespace Values in Go. Attack and defense points are only marked in the patterns when they have definite effects on the eye value, i.e. pure threats are not marked.

Examples of all different cases can be found among the patterns in this file. Some of them might be slightly counterintuitive, so we explain one important case here. Consider

```
Pattern 6141

 X
XX.@x

:1122
```

which e.g. matches in this position:

```
.OOOXXX
OOXOXOO
OXXba.O
OOOOOOO
```

Now it may look like 'X' could take away both eyes by playing 'a' followed by 'b', giving 0122 as eye value. This is where the subtlety of the definition of the first digit in the eye value comes into play. It does not say that the attacker is allowed two consecutive moves but only that he is allowed to play "another move". The crucial property of this shape is that when 'X' plays at a to destroy (at least) one eye, 'O' can answer at 'b', giving:

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

Now 'X' has to continue at 'c' in order to keep 'O' at one eye. After this 'O' plays tenuki and 'X' cannot destroy the remaining eye by another move. Thus the eye value is indeed 1122.

As a final note, some of the eye values indicating a threat depend on suicide to be allowed, e.g.

```
Pattern 301

X.X

:1222
```

We always assume suicide to be allowed in this database. It is easy enough to sort out such moves at a higher level when suicide is disallowed.

8.8 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:

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

A FALSE EYE is an eye vertex which cannot become a proper eye. Here 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. In this section we ignore the possibility of ko.

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.

8.9 Eye Topology with Ko

This section extends the topological eye analysis to handle ko. We distinguish between a ko in favor of 'O' and one in favor of 'X':

```
.?O?   good for O
OO.O
O.O?
XOX.
.X..
```

```
.?O?   good for X
OO.O
OXO?
X.X.
.X..
```

Preliminarily we give the former the symbolic diagonal value `a` and the latter the diagonal value `b`. We should clearly have `0 < a < 1 < b < 2`. Letting `e` be the topological eye value (still the sum of the four diagonal values), we want to have the following properties:

```
e <= 2     - proper eye
2 < e < 3  - worse than proper eye, better than half eye
e = 3      - half eye
3 < e < 4  - worse than half eye, better than false eye
e >= 4     - false eye
```

In order to determine the appropriate values of `a` and `b` we analyze the typical cases of ko contingent topological eyes:

```
      .X..      (slightly) better than proper eye
(a)   ..OO          e < 2
      OO.O
      O.OO      e = 1 + a
      XOX.
      .X..


      .X..      better than half eye, worse than proper eye
(a')  ..OO      2 < e < 3
      OO.O
      OXOO      e = 1 + b
      X.X.
      .X..


      .X..      better than half eye, worse than proper eye
(b)   .XOO      2 < e < 3
      OO.O
      O.OO      e = 2 + a
      XOX.
      .X..


      .X..      better than false eye, worse than half eye
(b')  .XOO      3 < e < 4
      OO.O
      OXOO      e = 2 + b
      X.X.
      .X..


      .X..
      XOX.      (slightly) better than proper eye
(c)   O.OO          e < 2
      OO.O
      O.OO      e = 2a
      XOX.
      .X..


      .X..
      XOX.      proper eye, some aji
(c')  O.OO      e ~ 2
      OO.O
      OXOO      e = a + b
      X.X.
      .X..


      .X..
      X.X.      better than half eye, worse than proper eye
(c'') OXOO      2 < e < 3
      OO.O
      OXOO      e = 2b
      X.X.
      .X..
```

```
       .X..
       XOX..     better than half eye, worse than proper eye
 (d)   O.O.X     2 < e < 3
       OO.O.
       O.OO.     e = 1 + 2a
       XOX..
       .X..


       .X..
       XOX..     half eye, some aji
 (d')  O.O.X     e ~ 3
       OO.O.
       OXOO.     e = 1 + a + b
       X.X..
       .X..


       .X..
       X.X..     better than false eye, worse than half eye
 (d'') OXO.X     3 < e < 4
       OO.O.
       OXOO.     e = 1 + 2b
       X.X..
       .X..


       .X..
       XOX..     better than false eye, worse than half eye
 (e)   O.OXX     3 < e < 4
       OO.O.
       O.OO.     e =  2 + 2a
       XOX..
       .X..


       .X..
       XOX..     false eye, some aji
 (e')  O.OXX     e ~ 4
       OO.O.
       OXOO.     e = 2 + a + b
       X.X..
       .X..


       .X..
       X.X..     (slightly) worse than false eye
 (e'') OXOXX     4 < e
       OO.O.
       OXOO.     e = 2 + 2b
       X.X..
       .X..
```

It may seem obvious that we should use

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

but this turns out to have some drawbacks. These can be solved by using either of

```
 (ii)  a=2/3, b=4/3
 (iii) a=3/4, b=5/4
 (iv)  a=4/5, b=6/5
```

Summarizing the analysis above we have the following table for the four different choices of `a` and `b`.

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

We can notice that (i) fails for the cases (c''), (d), (d''), and (e). The other three choices get all values in the correct intervals. The main distinction between them is the relative ordering of (c'') and (d) (or analogously (d'') and (e)). If we do a more detailed analysis of these we can see that in both cases `'O'` can secure the eye unconditionally if he moves first while `'X'` can falsify it with ko if he moves first. The difference is that in (c''), `'X'` has to make the first ko threat, while in (d), O has to make the first ko threat. Thus (c'') is better for O and ought to have a smaller topological eye value than (d). This gives an indication that (iv) is the better choice.

We can notice that any value of `a`, `b` satisfying `a+b=2` and `3/4<a<1` would have the same qualities as choice (iv) according to the analysis above. One interesting choice is `a=7/8, b=9/8` since these allow exact computations with floating point values having a binary mantissa. The latter property is shared by `a=3/4` and `a=1/2`.

When there are three kos around the same eyespace, things become more complex. This case is, however, rare enough that we ignore it.

8.10 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!)`.

8.11 Functions in ‘`optics.c`’

The public function `make_domains()` calls the function `make_primary_domains()` which is static in ‘`optics.c`’. It's purpose is to compute the domains of influence of each color, used in determining eye shapes. **Note**: the term influence as used here is distinct from the influence in influence.c.

For this algorithm the strings which are not lively are invisible. Ignoring these, the algorithm assigns friendly influence to

1. every vertex which is occupied by a (lively) friendly stone,
2. every empty vertex adjoining a (lively) friendly stone,
3. every empty vertex for which two adjoining vertices (not on the first line) in the (usually 8) surrounding ones have friendly influence, with two CAVEATS explained below.

Thus in the following diagram, ‘e’ would be assigned friendly influence if ‘a’ and ‘b’ have friendly influence, or ‘a’ and ‘d’. It is not sufficent for ‘b’ and ‘d’ to have friendly influence, because they are not adjoining.

```
uabc
 def
 ghi
```

The constraint that the two adjoining vertices not lie on the first line prevents influence from leaking under a stone on the third line.

The first CAVEAT alluded to above is that even if ‘a’ and ‘b’ have friendly influence, this does not cause ‘e’ to have friendly influence if there is a lively opponent stone at ‘d’. This constraint prevents influence from leaking past knight's move extensions.

The second CAVEAT is that even if ‘a’ and ‘b’ have friendly influence this does not cause ‘e’ to have influence if there are lively opponent stones at ‘u’ and at ‘c’. This prevents influence from leaking past nikken tobis (two space jumps).

The corner vertices are handled slightly different.

```
+---
|ab
|cd
```

We get friendly influence at ‘a’ if we have friendly influence at ‘b’ or ‘c’ and no lively unfriendly stone at ‘b’, ‘c’ or ‘d’.

Here are the public functions in ‘`optics.c`’, except some simple access functions used by autohelpers. The statically declared functions are documented in the source code.

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

  This function is called from `make_dragons()` and from `owl_determine_life()`. It marks the black and white domains (eyeshape regions) and collects some statistics about each one.

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

  Find connected eyespace components and compute relevant statistics.

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

  propagate_eye(origin) copies the data at the (origin) to the rest of the eye (invariant fields only).

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

  Find the dragon or dragons surrounding an eye space. Up to max_dragons dragons adjacent to the eye space are added to the dragon array, and the number of dragons found is returned.

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

  Given an eyespace with origin `pos`, this function computes the minimum and maximum numbers of eyes the space can yield. If max and min are different, then vital points of attack and defense are also generated. 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 pos, struct eyevalue *value, int *pessimistic_min, int *attack_point, int *defense_point, struct eye_data eye[BOARDMAX], struct half_eye_data heye[BOARDMAX])`

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

  turns a proper eyespace into a margin.

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

  These functions are used from constraints to identify eye spaces, primarily for late endgame moves.

- `int max_eye_value(int pos)`

  Return the maximum number of eyes that can be obtained from the eyespace at `(i, j)`. This is most useful in order to determine whether the eyespace can be assumed to produce any territory at all.

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

  These functions simply return information about an eyeshape that has already been analyzed. (They do no real work.)

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

  Find topological half eyes and false eyes by analyzing the diagonal intersections, as described in the Texinfo documentation (Eyes/Eye Topology).

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

  See Texinfo documentation (Eyes:Eye Topology). Returns:

  - 2 or less if `pos` is a proper eye for `color`;
  - between 2 and 3 if the eye can be made false only by ko
  - 3 if `pos` is a half eye;
  - between 3 and 4 if the eye can be made real only by ko
  - 4 or more if `pos` is a false eye.

  Attack and defense points for control of the diagonals are stored in the `heye[]` array. `my_eye` is the eye space information with respect to `color`.

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

  Conservative relative of `topological_eye()`. Essentially the same algorithm is used, but only tactically safe opponent strings on diagonals are considered. This may underestimate the false/half eye status, but it should never be overestimated.

- `void set_eyevalue(struct eyevalue *e, int a, int b, int c, int d)`

  set parameters into the `struct eyevalue` **as follows:** (see section Eye Local Game Values):

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

- `int min_eye_threat(struct eyevalue *e)`

  Number of eyes if attacker plays first twice (the threat of the first move by attacker).

- `int min_eyes(struct eyevalue *e)`

  Number of eyes if attacker plays first followed by alternating play.

- `int max_eyes(struct eyevalue *e)`

  Number of eyes if defender plays first followed by alternating play.

- `int max_eye_threat(struct eyevalue *e)`

  Number of eyes if defender plays first twice (the threat of the first move by defender).

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

  Add the eyevalues `*e1` and `*e2`, leaving the result in *sum. It is safe to let `sum` be the same as `e1` or `e2`.

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

  Produces a string containing the eyevalue. **Note**: the result string is stored in a statically allocated buffer which will be overwritten the next time this function is called.

- `void test_eyeshape(int eyesize, int *eye_vertices)` /* Test whether the optics code evaluates an eyeshape consistently. */
- `int analyze_eyegraph(const char *coded_eyegraph, struct eyevalue *value, char *analyzed_eyegraph)`

  Analyze an eye graph to determine the eye value and vital moves. The eye graph is given by a string which is encoded with ‘`%`’ for newlines and ‘`O`’ for spaces. E.g., the eye graph

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

is encoded as `00!%0.X%!....` (The encoding is needed for the GTP interface to this function.) The result is an eye value and a (nonencoded) pattern showing the vital moves, using the same notation as eyes.db. In the example above we would get the eye value 0112 and the graph (showing ko threat moves)

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

If the eye graph cannot be realized, 0 is returned, 1 otherwise.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

9. The Pattern Code

9.1 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 The Joseki Compiler). 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 section Eyes and Half Eyes).

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:

```
|oOO?
|...X
|..*?
|..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`’.

9.2 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.

9.2.1 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 status (see section Dragons) `ALIVE` or `UNKNOWN`. The `CRITICAL` matcher status is excluded. It is possible for a string to have `ALIVE` 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 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`

9.2.2 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.

‘`e`’

The move makes or secures territory.

‘`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 minimum accepted value of 27.

'F'

This indicates a fuseki pattern. This is only effective together with either the 'j' or 't' classification, and is used to ensure indeterministic play during fuseki.

'j'

Slightly less urgent joseki move. These moves will be made after those with the 'J' classification. This adds the 'e' and 'E' classifications. If the move has the 'F' classification, it also gets a fixed value of 20.1, otherwise it gets a minimum accepted value of 20. (This makes sure that GNU Go chooses randomly between different moves that have 'jF' as classification.)

't'

Minor joseki move (tenuki OK). This is equivalent to adding the 'e' and 'E' classifications together with either a fixed value of 15.07 (if the move has 'F' classification) or a minimum value of 15 (otherwise).

'U'

Urgent joseki move (never tenuki). This is equivalent to the 'd' and 'a' classifications together with a shape bonus of 15 and a minimum accepted value of 40.

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.

9.3 Pattern Attributes

The second and following fields in the ':' line of a pattern are 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 least 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 least/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 section Move generation.

9.4 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 a hypothetical helper called `wedge_helper`. Such a helper used to exist, but has been replaced by a constraint. Due to its simplicity it's still a good example. The helper begins with a comment:

```
/*

?O.           ?Ob
.X*           aX*
?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 `apos`, `bpos`, and `cpos`. The position of *, the point where GNU Go will move if the pattern is adopted, is passed through the parameter `move`.

```
int
wedge_helper(ARGS)
{
  int apos, bpos, cpos;
  int other = OTHER_COLOR(color);
  int success = 0;

  apos = OFFSET(0, -2);
  bpos = OFFSET(-1, 0);
  cpos = OFFSET(1, 0);

  if (TRYMOVE(move, color)) {
    if (TRYMOVE(apos, other)) {
      if (board[apos] == EMPTY || attack(apos, NULL))
        success = 1;
      else if (TRYMOVE(bpos, color)) {
        if (!safe_move(cpos, other))
          success = 1;
        popgo();
      }
      popgo();
    }
    popgo();
  }

  return success;
}
```

The `OFFSET` lines tell GNU Go the positions of the three stones at 'a', 'b', and 'c'. To decide whether the pattern guarantees a connection, we do some reading. First we use the `TRYMOVE` macro to place an 'O' at 'move' and let 'X' draw back to 'a'. Then we ask 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()`.

9.5 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.

9.6 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.

Here is a partial list of the autohelper macros which are typically called from action lines. This list is not complete. If you cannot find an autohelper macro in an action line in this list, consult 'mkpat.c' to find out what function in the engine is actually called. If no macro exists which does what you want, you can add macros by editing the list in 'mkpat.c'.

`antisuji(a);`
: Mark 'a' as an antisuji, that is, a move that must never be played.

`replace(a,*)`
: This is appropriate if the move at '*' is definitely better than the move at 'a'. The macro adds a point redistribution rule. Any points which are assigned to 'a' during the move generation by any move reason are redistributed to '*'.

`prevent_attack_threat(a)`
: Add a reverse followup value because the opponent's move here would threaten to capture 'a'.

`threaten_to_save(a)`
: Add a followup value because the move at '*' threatens to rescue the dead string at 'a'.

`defend_against_atari(a)`
: Estimate the value of defending the string 'a' which can be put into atari and add this as a reverse followup value.

`add_defend_both_move(a,b);`

`add_cut_move(c,d);`
: Add to the move reasons that the move at '*' threatens to cut 'c' and 'd'.

9.7 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 section Worms and Dragons, 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 General Utilities) 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 section Worms and Dragons).

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

Each function true if the dragon has the corresponding matcher status and false otherwise (see section Worms and Dragons).

```
status(x)
```

Returns the status of the dragon at 'x' (see section Worms and Dragons).

```
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)
```

These functions call `whose_territory()`, `whose_moyo()` and `whose_area()` (see section Territory, Moyo and Area). For example `xarea(x)` evaluates to true if 'x' is either a living enemy stone or an empty point within the opponent's "area". The function `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.

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

The function `xplay_connect(a,b,c,...,y,z)` begins with an 'X' move at 'a', then an 'O' move at 'b', and so forth, then finally calls `string_connect()` to determine whether 'x' and 'y' can be connected. The other functions are similar (see section Connection Reading).

```
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 section Worms and Dragons).

```
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 Eye spaces).

```
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 Topology of Half Eyes and False Eyes, 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.

9.8 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.

9.9 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 section Worms and Dragons.

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.

9.10 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 The Connections Database). 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 The Connections Database). When such a function is found, the function `cut_connect_callback` is invoked.

9.11 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 Game 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 Game 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.

9.12 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 GNU Go 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)

9.13 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 ‘\’.

9.14 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’ and some contain no ‘X’.)
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.

9.15 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.

9.16 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,*)
```

9.17 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.

9.18 Corner Matcher

GNU Go uses a special matcher for joseki patterns. It has certain constraints on the patterns it can match, but is much faster and takes far less space to store patterns than the standard matcher.

Patterns used with corner matcher have to qualify the following conditions:

- They must be matchable only at a corner of the board (hence the name of the matcher).
- They can consist only of 'O', 'X' and '.' elements.
- Of all pattern values (see section Pattern Attributes), corner matcher only support `shape(x)`. This is not because the matcher cannot handle other values in principle, just they are currently not used in joseki databases.

Corner matcher was specifically designed for joseki patterns and they of course satisfy all the conditions above. With some modifications corner matcher could be used for fuseki patterns as well, but fullboard matcher does its work just fine.

The main idea of the matcher is very same to the one of DFA matcher (see section Pattern matching with DFA): check all available patterns at once, not a single pattern at a time. A modified version of DFA matcher could be used for joseki pattern matching, but its database would be very large. Corner matcher capitalizes on the fact that there are relatively few stones in each such pattern.

Corner pattern database is organized into a tree. Nodes of the tree are called "variations". Variations represent certain sets of stones in a corner of the board. Root variation corresponds to an empty corner and a step down the tree is equivalent to adding a stone to the corner. Each variation has several properties:

- stone position relative to the corner,
- a flag determining whether the stone color must be equal to the first matched stone color,
- number of stones in the corner area (see below) of the variation stone.

By corner area we define a rectangle which corners are the current corner of the board and the position of the stone (inclusive). For instance, if the current board corner is A19 then corner area of a stone at C18 consists of A18, A19, B18, B19, C18 and C19.

Variation which is a direct child of the root variation matches if there is any stone at the variation position and the stone is alone in its corner area.

Variation at a deeper level of the tree matches if there is a stone of specified color in variation position and the number of stones in its corner area is equal to the number specified in variation structure.

When a certain variation matches, all its children has to be checked recursively for a match.

All leaf variations and some inner ones have patterns attached to them. For a pattern to match, it is required that its parent variation matches. In addition, it is checked that pattern is being matched for the appropriate color (using its variation "stone color" field) and that the number of stones in the area where the pattern is being matched is indeed equal to the number of stones in the pattern. The "stone position" property of the pattern variation determines the move suggested by the pattern.

Consider this joseki pattern which has four stones:

```
------+
......|
......|
.O*...|
.XXO..|
......|
......|
```

To encode it for the corner matcher, we have to use five variations, each next being a child of previous:

Tree level	Position	Color	Number of stones
1	R16	"same"	1
2	P17	"same"	1
3	Q16	"other"	2
4	P16	"other"	4
5	Q17	"same"	1

The fifth variation should have an attached pattern. Note that the stone color for the fifth variation is "same" because the first matched stone for this pattern is 'O' which stands for the stones of the player to whom moves are being suggested with '*'.

The tree consists of all variations for all patterns combined together. Variations for each patterns are sorted to allow very quick tree branch rejection and at the same time keep the database small enough. More details can be found in comments in file 'mkpat.c'

Corner matcher resides in 'matchpat.c' in two functions: `corner_matchpat()` and `do_corner_matchpat()`. The former computes `num_stones[]` array which holds number of stones in corner areas of different intersections of the board for all possible transformations. `corner_matchpat()` also matches top-level variations. `do_corner_matchpat()` is responsible for recursive matching on the variation tree and calling callback function upon pattern match.

Tree-like database for corner matcher is generated by `mkpat` program. Database generator consists of several functions, most important are: `corner_best_element()`, `corner_variation_new()`, `corner_follow_variation()` and `corner_add_pattern()`.

9.19 Emacs Mode for Editing Patterns

If you use GNU Emacs (XEmacs might work too), you can try a special mode for editing GNU Go pattern databases. The mode resides in 'patterns/gnugo-db.el'.

Copy the file to 'emacs/site-lisp' directory. You can then load the mode with `(require 'gnugo-db)`. It makes sense to put this line into your configuration file ('~/.emacs'). You can either use `gnugo-db-mode` command to switch to pattern editing mode, or use the following code snippet to make Emacs do this automatically upon opening a file with '.db' suffix:

```
(setq auto-mode-alist
      (append
       auto-mode-alist
       '(("\\.db\\'" . gnugo-db-mode))))
```

Pattern editing mode provides the following features:

- highlighting of keywords (`Pattern`, `goal_elements` and `callback_data`) and comments,
- making paragraphs equal to patterns (`M-h`, `M-{`, `M-}` and others operate on patterns),
- commands for pattern creation with automatic name selection (`C-c C-p`) and copying main diagram to constraint diagram (`C-c C-c`),
- automated indentation of constraints and side comments (pattern descriptions).

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

10. The DFA pattern matcher

In this chapter, we describe the principles of the GNU Go DFA pattern matcher. The aim of this system is to permit a fast pattern matching when it becomes time critical like in owl module (The Owl Code). Since GNU Go 3.2, this is enabled by default. You can still get back the traditional pattern matcher by running `configure --disable-dfa` and then recompiling GNU Go.

Otherwise, a finite state machine called a Deterministic Finite State Automaton (What is a DFA) will be built off line from the pattern database. This is used at runtime to speedup pattern matching (Pattern matching with DFA and Incremental Algorithm). The runtime speedup is at the cost of an increase in memory use and compile time.

10.1 Introduction to the DFA

The general idea is as follows:

For each intersection of the board, its neighbourhood is scanned following a predefined path. The actual path used does not matter very much; GNU Go uses a spiral as shown below.

path

In each step of the path, the pattern matcher jumps into a state determined by what it has found on the board so far. If we have successfully matched one or several patterns in this step, this state immediately tells us so (in its attribute). But the state also implicitly encodes which further patterns can still get matched: The information stored in the state contains in which state to jump next, depending on whether we find a black, white or empty intersection (or an intersection out of board) in the next step of the path. The state will also immediately tell us if we cannot find any further pattern (by telling us to jump into the error state).

These sloppy explanations may become clearer with the definitions in the next section (What is a DFA).

Reading the board following a predefined path reduces the two dimentional pattern matching to a linear text searching problem. For example, this pattern

```
?X?
.O?
?OO
```

scanned following the path

```
 B
C4A
5139
628
 7
```

gives the string **"OO?X.?*O*?*?"** where **"?"** means **'don't care'** and **"*"** means **'don't care, can even be out of board'**.

So we can forget that we are dealing with two dimensional patterns and consider linear patterns.

10.2 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 GNU Go 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:

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 (Building the DFA). Consider the following DFA:

dfa2

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 (Building the DFA) is only possible on CDFA's.

cdfa

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;
```

10.3 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.

10.4 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:

sync-prod1

Give by synchronization product the following one:

sync-prod2

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.

10.5 Incremental Algorithm

The incremental version of the DFA pattern matcher is not yet implemented in GNU Go 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.

10.6 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.

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

11. 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 connection reading. In this Chapter, we document the tactical reading code, which is in 'engine/reading.c'.

11.1 Reading Basics

What we call Tactical Reading is the analysis whether there is a direct capture of a single string, or whether there is a move to prevent such a direct capture.

If the reading module finds out that the string can get captured, this answer should (usually) be trusted. However, if it says it can be defended, this does not say as much. It is often the case that such a string has no chance to make a life, but that it cannot be captured within the horizon (and the cutoff heuristics) of the tactical reading.

The tactical reading is done by the functions in 'engine/reading.c'. It is a minimax search that declares win for the attacker once he can physically take the string off board, whereas the defense is considered successful when the string has sufficiently many liberties. A string with five liberties is always considered alive. At higher depth within the search tree even fewer liberties cause GNU Go to give up the attack, See depthparams.

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.

11.1.1 Organization of the reading code

The function `do_attack` and `do_find_defense` are wrappers themselves and call `attack1`, `attack2`, `attack3` or `attack4` resp. `defend1`, `defend1`, `defend1` or `defend1` depending on the number of liberties.

These are fine-tuned to generate and try out the moves in an efficient order. They generate a few moves themselves (mostly direct liberties of the string), and then call helper functions called `..._moves` which suggest less obvious moves. Which of these functions get called depends on the number of liberties and of the current search depth.

11.1.2 Return Codes

The return codes of the reading (and owl) functions and owl can be `0`, `KO_B`, `KO_A` or `WIN`. Each reading function determines whether a particular player (assumed to have the move) can solve a specific problem, typically attacking or defending a string.

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.

If GNU Go is compiled with the configure option '--enable-experimental-owl-ext' then the owl functions also have possible return codes of `GAIN` and `LOSS`. A code of `GAIN` means that the attack (or defense) does not succeed, but that in the process of trying to attack or defend, an opponent's worm is captured. A code of `LOSS` means that the attack or defense succeeds, but that another friendly worm dies during the attack or defense.

11.1.3 Reading cutoff and depth parameters

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 16 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 12. 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 7. 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.

- `int attack(int str, int *move)`

  Determines if the string at `str` can be attacked, and if so, `*move` returns the attacking move, unless `*movei` is a null pointer. (Use null pointers if you are interested in the result of the attack but not the attacking move itself.) Returns `WIN`, if the attack succeeds, 0 if it fails, and `KO_A` or `KO_B` if the result depends on ko Return Codes.

- `find_defense(int str, int *move)`

  Attempts to find a move that will save the string at `str`. It returns true if such a move is found, with `*move` the location of the saving move (unless `*move` is a null pointer). It is not checked that tenuki defends, so this may give an erroneous answer if `!attack(str)`. Returns `KO_A` or `KO_B` if the result depends on ko See Return Codes.

- `safe_move(int str, int color)` :

  The function `safe_move(str, color)` checks whether a move at `str` 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.

11.2 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 Invoking GNU Go: Command line options.

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()`.

11.2.1 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).

11.2.2 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.

11.2.3 Hash Structures

The basic hash structures are declared in ‘`engine/hash.h`’ and ‘`engine/cache.c`’

```
typedef struct hashposition_t {
  Compacttype  board[COMPACT_BOARD_SIZE];
  int          ko_pos;
} 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 data1;
  unsigned int data2;

  struct read_result_t *next;
} Read_result;
```

The data1 field packs into 32 bits the following fields:

```
komaster:  2 bits (EMPTY, BLACK, WHITE, or GRAY)
kom_pos : 10 bits (allows MAX_BOARD up to 31)
routine :  4 bits (currently 10 different choices)
str1    : 10 bits
stackp  :  5 bits
```

The data2 field packs into 32 bits the following fields:

```
status :   2 bits (0 free, 1 open, 2 closed)
result1:   4 bits
result2:   4 bits
move   :  10 bits
str2   :  10 bits
```

The `komaster` and `(kom_pos)` field are documented in See section Ko Handling.

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.

11.3 Persistent Reading Cache

Some calculations can be safely saved from move to move. If the opponent's move is not close to our worm or dragon, we do not have to reconsider the life or death of that group on the next move. So the result is saved in a persistent cache. Persistent caches are used for are used in the engine for several types of read results.

- Tactical reading
- Owl reading
- Connection reading
- Breakin code

In this section we will discuss the persistent caching of tactical reading but the same principles apply to the other persistent caches.

Persistent caching is an important performance feature. However it can lead to mistakes and debugging problems—situations where GNU Go generates the right move during debugging but plays a wrong move during a game. If you suspect a persistent cache effect you may try loading the sgf file with the ‘`--replay`’ option and see if the mistake is repeated (see section Invoking GNU Go: Command line options).

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.

11.4 Ko Handling

The principles of ko handling are the same for tactical reading and owl reading.

We have already mentioned (see section Reading Basics) 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` consult parameters `komaster`, `kom_pos`, which are declared static in ‘`board.c`’. These mediate ko captures to prevent the occurrence of infinite loops. During reading, the komaster values are pushed and popped from a stack.

Normally `komaster` is `EMPTY` but it can also be ‘`BLACK`’, ‘`WHITE`’, `GRAY_BLACK`, `GRAY_WHITE` or `WEAK_KO`. The komaster is set to `color` when `color` makes a conditional ko capture. In this case `kom_pos` 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_pos`. 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_pos`, 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_pos`, 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_BLACK` or `GRAY_WHITE` and after this further ko captures are even further restricted.

If the ko at `kom_pos` is filled, then the komaster reverts to `EMPTY`.

In detail, the komaster scheme is as follows. Color ‘O’ is to move. This scheme is known as scheme 5 since in versions of GNU Go through 3.4, several different schemes were included.

- 1. Komaster is EMPTY.
  - 1a. Unconditional ko capture is allowed.

    Komaster remains EMPTY if previous move was not a ko capture. Komaster is set to WEAK_KO if previous move was a ko capture and kom_pos is set to the old value of board_ko_pos.

  - 1b) Conditional ko capture is allowed.

    Komaster is set to O and kom_pos to the location of the ko, where a stone was just removed.

- 2. Komaster is O:
  - 2a) Only nested ko captures are allowed. Kom_pos is moved to the new removed stone.
  - 2b) If komaster fills the ko at kom_pos then komaster reverts to EMPTY.
- 3. Komaster is X:

  Play at kom_pos is not allowed. Any other ko capture is allowed. If O takes another ko, komaster becomes GRAY_X.

- 4. Komaster is GRAY_O or GRAY_X:

  Ko captures are not allowed. If the ko at kom_pos is filled then the komaster reverts to EMPTY.

- 5. Komaster is WEAK_KO:
  - 5a) After a non-ko move komaster reverts to EMPTY.
  - 5b) Unconditional ko capture is only allowed if it is nested ko capture.

    Komaster is changed to WEAK_X and kom_pos to the old value of board_ko_pos.

  - 5c) Conditional ko capture is allowed according to the rules of 1b.

11.5 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 --decide-dragon 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_pos: A2
 6 . . O . . . . .
 5 O O O . . . . .
```

```
    4 O X O O . . . .
    3 X . X O O O O .
    2 . X X X O . . .
    1 X O . . . . . .
      A B C D E F G H
```

W takes the ko at B3 after which the komaster is `GRAY` and ko recaptures are not allowed.

```
    8 . . . . . . . .         komaster: GRAY
    7 . . O . . . . .         kom_pos: B4
    6 . . O . . . . .
    5 O O O . . . . .
    4 O . O O . . . .
    3 X O X O O O O .
    2 . X X X O . . .
    1 X O . . . . . .
      A B C D E F G H
```

Since B 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.

11.6 Another Ko Example

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 --decide-string L1 -o vars.sgf
```

The correct resolution is that H1 attacks L1 unconditionally 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.

11.7 Alternate Komaster Schemes

The following alternate schemes have been proposed. It is assumed that ‘`O`’ is the player about to move.

11.7.1 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_pos` 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_pos`. Komaster parameters unchanged.

11.7.2 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_pos` 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_pos, X)` returns false. In that case, `kom_pos` 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_pos`. Komaster parameters unchanged.

11.8 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.

11.9 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
```

11.10 Connection Reading

GNU Go does reading to determine if strings can be connected. The algorithms for this are in ‘`readconnect.c`’. As with the reading code, the connection code is not pattern based.

The connection code is invoked by the engine through the functions:

`int string_connect(int str1, int str2, int *move)`

    Returns `WIN` if `str1` and `str2` can be connected.

`int disconnect(int str1, int str2, int *move)`

    Returns `WIN` if `str1` and `str2` can be disconnected.

To see the connection code in action, you may try the following example.

```
gnugo --quiet -l connection3.sgf --decide-connection M3/N7 -o vars.sgf
```

(The file ‘`connection3.sgf`’ is in ‘`regression/games`’.) Examine the sgf file produced by this to see what kind of reading is done by the functions `string_connect()` and `string_disconnect()`, which are called by the function `decide_connection`.

One use of the connection code is used is through the autohelper macros `oplay_connect`, `xplay_connect`, `oplay_disconnect` and `xplay_disconnect` which are used in the connection databases.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

12. Pattern Based Reading

In the tactical reading code in ‘`reading.c`’ , the code generating the moves which are tried are all hand coded in C, for efficiency. There is much to be said for another type of reading, in which the moves to be tried are generated from a pattern database.

GNU Go does three main types of pattern based 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 section Eyes and Half Eyes) can be used to evaluate it. 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.

Secondly, there is the ‘`engine/combination.c`’ which attempts to find combinations—situations where a series of threats eventually culminates in one that cannot be parried.

Finally there is the semeai module. A **semeai** is a capturing race between two adjacent DEAD or CRITICAL dragons of opposite colors. The principal function, `owl_analyze_semeai()` is contained in ‘`owl.c`’ . Due to the complex nature of semeais, the results of this function are more frequently wrong than the usual owl code.

12.1 The Owl Code

The life and death code in ‘`optics.c`’ , described elsewhere (see section Eyes and Half Eyes), 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 ‘`owl_attackpats.db`’ or ‘`owl_defendpats.db`’ , or if the function `compute_eyes_pessimistic()` reports that the group is definitely alive. 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, `0`. 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 ‘`--decide-owl location`’ option. 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. ‘`--decide-position`’ performs the same analysis for all dragons with small escape route.

12.2 Combination reading

It may happen that no single one of a set of worms can be killed, yet there is a move that guarantees that at least one can be captured. The simplest example is a double atari. The purpose of the code in ‘`combination.c`’ is to find such moves.

For example, consider the following situation:

```
+---------
|....OOOOX
|....OOXXX
|..O.OXX..
|.OXO.OX..
|.OX..OO..
|.XXOOOXO.
|..*XXOX..
|....XOX..
|.XX..X...
```

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

Every ‘`X`’ stone in this position is alive. However the move at ‘`*`’ produces a position in which at least one of four strings will get captured. This is a combination.

The driving function is called `atari_atari` because typically a combination involves a sequence of ataris culminating in a capture, though sometimes the moves involved are not ataris. For example in the above example, the first move at ‘`*`’ is not an atari, though after ‘`O`’ defends the four stones above, a sequence of ataris ensues resulting in the capture of some string.

Like the owl functions `atari_atari` does pattern-based reading. The database generating the attacking moves is ‘`aa_attackpats.db`’. One danger with this function is that the first atari tried might be irrelevant to the actual combination. To detect this possibility, once we've found a combination, we mark that 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.

- `void combinations(int color)`

  Generate move reasons for combination attacks and defenses against them. This is one of the move generators called from genmove().

- `int atari_atari(int color, int *attack_move, char defense_moves[BOARDMAX], int save_verbose)`

  Look for a combination for `color`. For the purpose of the move generation, returns the size of the smallest of the worms under attack.

- `int atari_atari_confirm_safety(int color, int move, int *defense, int minsize, const char saved_dragons[BOARDMAX], const char saved_worms[BOARDMAX])`

  Tries to determine whether a move is a blunder. Wrapper around atari_atari_blunder_size. Check whether a combination attack of size at least `minsize` appears after move at `move` has been made. The arrays `saved_dragons[]` and `saved_worms[]` should be one for stones belonging to dragons or worms respectively, which are supposedly saved by `move`.

- `int atari_atari_blunder_size(int color, int move, int *defense, const char safe_stones[BOARDMAX])`

  This function checks whether any new combination attack appears after move at (move) has been made, and returns its size (in points). `safe_stones` marks which of our stones are supposedly safe after this move.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

13. Influence Function

13.1 Conceptual Outline of Influence

We define call stones lively if they cannot be tactically attacked, or if they have a tactical defense and belong to the player whose turn it is. Similarly, stones that cannot be strategically attacked (in the sense of the life-and-death analysis), or that have a strategical defense and belong to the player to move, 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.

From the influence of both colors we compute a territorial value between -1.0 and +1.0 for each intersection, which can be seen as the likely hood of it becoming territory for either color.

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. These intrusions depend on whose turn we assume it to be.

All these extra influence sources, as well as connections, are controlled by a pattern database, which consists of the two files patterns/influence.db and patterns/barriers.db. The details are explained in Patterns used by the Influence module.

13.2 Territory, Moyo and Area

Using the influence code, empty regions of the board are partitioned in three ways. A vertex may be described as White or Black's territory, moyo or area. The functions `whose_territory()`, `whose_moyo()` and `whose_area()` will return a color, or EMPTY if it belongs to one player or the other in one of these classifications.

- Territory

  Those parts of the board which are expected to materialize as actual points for one player or the other at the end of the game are considered territory.

- Moyo

  Those parts of the board which are either already territory or more generally places where a territory can easily materialize if the opponent neglects to reduce are considered moyo. moyo.

- Area

  Those parts of the board where one player or the other has a stronger influence than his opponent are considered area.

Generally territory is moyo and moyo is area. To get a feeling for these concepts, load an sgf file in a middle game position with the option '`-m 0x0180`' and examine the resulting diagrams (see section Colored display and debugging of influence).

13.3 Where influence gets used in the engine

The information obtained from the influence computation is used in a variety of places in the engine, and the influence module is called several times in the process of the move generation. The details of the influence computation vary according to the needs of the calling function.

After GNU Go has decided about the tactical stability of strings, the influence module gets called the first time. Here all lively stones act as an influence source of default strength 100. The result is stored in the variables `initial_influence` and `initial_opposite_influence`, and it is used as an important information for guessing the strength of dragons. For example, a dragon that is part of a moyo of size 25 is immediately considered alive. For dragons with a smaller moyo size, a life-and-death analysis will be done by the owl code (see Pattern Based Reading). A dragon with a moyo size of only 5 will be considered weak, even if the owl code has decided that it cannot be killed.

As a tool for both the owl code and the strength estimate of dragons, an "escape" influence gets computed for each dragon (see section Escape).

Once all dragons have been evaluated, the influence module is called again and the variables `initial_influence` and `initial_opposite_influence` get overwritten. Of course, the dragon status', which are available now, are taken into account. Stones belonging to a dead dragon will not serve as an influence source, and the strengths of other stones get adjusted according to the strength of their respective dragon.

The result of this run is the most important tool for move evaluation. All helper functions of patterns as explained in The Pattern Code that refer to influence results (e. g. `olib(*)` etc.) actually use these results. Further, `initial_influence` serves as the reference for computing the territorial value of a move. That is, from the influence strengths stored in `initial_influence`, a territory value is assigned to each intersection. This value is supposed to estimate the likelyhood that this intersection will become white or black territory.

Then, for each move that gets considered in the function `value_moves`, the influence module is called again via the function `compute_move_influence` to assess the likely territorial balance after this move, and the result is compared with the state before that move.

An additional influence computation is done in order to compute the followup value of a move. Some explainations are in Details of the Territory Valuation.

Some of the public functions from '`influence.c`' which are used throughout the engine are listed in Utilities from '`engine/influence.c`'.

13.4 Influence and Territory

In this section we consider how the influence function is used to estimate territory in the function `estimate_territorial_value()`.

A move like '`*`' by '`O`' below is worth one point:

```
OXXX.
OX.XX
O*a.X
OX.XX
OXXX.
```

This is evaluated by the influence function in the following way: We first assign territory under the assumption that X moves first in all local positions in the original position; then we reassing territory, again under the assumption that '`X`' moves first in all local positions, but after we let '`O`' make the move at '`*`'. These two territory assignments are compared and the difference gives the territorial value of the move.

Technically, the assumption that '`X`' plays first everywhere is implemented via an asymmetric pattern database in `barriers.db`. What exactly is a safe connection that stops hostile influence from passing through is different for '`O`' and '`X`'; of course such a connection has to be tighter for stones with color '`O`'. Also, additional intrusion influence sources are added for '`X`' in places where '`X`' stones have natural followup moves.

In this specific example above, the asymmetry (before any move has been made) would turn out as follows: If '`X`' is in turn to move, the white influence would get stopped by a barrier at '`*`', leaving 4 points of territory for '`X`'. However, if '`O`' was next to move, then a followup move for the white stones at the left would be assumed in the form of an extra ("intrusion") influence source at '`*`'. This would get stopped at '`a`', leaving three points of territory.

Returning to the valuation of a move by '`O`' at '`*`', we get a value of 1 for the move at '`*`'. However, of course this move is sente once it is worth playing, and should therefore (in miai counting) be awarded an effective value of 2. Hence we need to recognize the followup value of a move. GNU Go 3.0 took care of this by using patterns in `patterns.db` that enforced an explicit followup value. Versions from 3.2 on instead compute a seperate followup influence to each move considered. In the above example, an intrusion source will be added at '`a`' as a followup move to '`*`'. This destroys all of Black's territory and hence gives a followup value of 3.

The pattern based followup value are still needed at some places, however.

To give another example, consider this position where we want to estimate the value of an '`O`' move at '`*`':

```
OOOXXX
..OX..
..OX..
...*..
------
```

Before the move we assume '`X`' moves first in the local position (and that '`O`' has to connect), which gives territory like this (lower case letter identify territory for each player):

```
OOOXXX
ooOXxx
o.OXxx
o...xx
------
```

Then we let '`O`' make the move at '`*`' and assume '`X`' moves first again next. The territory then becomes ('`X`' is also assumed to have to connect):

```
OOOXXX
ooOXxx
ooOX.x
oo.O.x
------
```

We see that this makes a difference in territory of 4, which is what influence_delta_territory() should report. Then again, we have followup value, and here also a reverse followup value. The reverse followup value, which in this case will be so high that the move is treated as reverse sente, is added by an explicit pattern. Other sources for followup or reverse followup values are threats to capture a rescue a string of stones. See the code and comments in the function `value_move_reaons` for how followup and reverse followup values are used to adjust the effective move value.

To give an example of territorial value where something is captured, consider the '`O`' move at '`*`' here,

```
XXXXXXXO
X.OOOOXO
X.O..O*O
--------
```

As before we first let the influence function determine territory assuming X moves first, i.e. with a captured group:

```
XXXXXXXO
```

```
XxyyyyXO
Xxyxxy.O
--------
```

Here ‘y’ indicates ‘X’ territory + captured stone, i.e. these count for two points. After the ‘O’ move at ‘*’ we instead get

```
XXXXXXXO
X.OOOOXO
X.OooOOO
--------
```

and we see that ‘X’ has 16 territory fewer and ‘O’ has two territory more, for a total difference of 18 points.

That the influence function counts the value of captured stones was introduced in GNU Go 3.2. Previously this was instead done using the effective_size heuristic. The effective size is the number of stones plus the surrounding empty spaces which are closer to this string or dragon than to any other stones. Here the ‘O’ string would thus have effective size 6 (number of stones) + 2 (interior eye) + 2*0.5 (the two empty vertices to the left of the string, split half each with the surrounding X string) + 1*0.33 (the connection point, split between three strings) = 9.33. As noted this value was doubled, giving 18.67 which is reasonably close to the correct value of 18. The effective size heuristic is still used in certain parts of the move valuation where we can't easily get a more accurate value from the influence function (e. g. attacks depending on a ko, attack threats).

Note that this section only describes the territorial valuation of a move. Apart from that, GNU Go uses various heuristics in assigning a strategical value (weakening and strengthening of other stones on the board) to a move. Also, the influence function isn't quite as well tuned as the examples above may seem to claim. But it should give a fairly good idea of how the design is intended.

Another matter is that so far we have only considered the change in secure territory. GNU Go 3.2 and later versions use a revised heuristic, which is explained in the next section, to assign probable territory to each player.

13.5 Details of the Territory Valuation

This section explains how GNU Go assigns a territorial value to an intersection once the white and black influence have been computed. The intention is that an intersection that has a chance of xx% of becoming white territory is counted as 0.xx points of territory for white, and similar for black.

The algorithm in the function `new_value_territory` goes roughly as follows:

If `wi` is the white influence at a point, and `bi` the black influence, then `value = ( (wi-bi)/ (wi+bi) )^3` (positive values indicates likley white territory, negative stand for black territory) turns out to be very simple first guess that is still far off, but reasonable enough to be useful.

This value is then suspect a number of corrections. Assume that this first guess resulted in a positive value.

If both `bi` and `wi` are small, it gets reduced. What exactly is "small" depends on whether the intersection is close to a corner or an edge of the board, since it is easier to claim territory in the corner than in the center.

Then the value at each intersection is degraded to the minimum value of its neighbors. This can be seen as a second implementation of the proverb saying that there is no territory in the center of the board. This step substantially reduces the size of spheres of territory that are open at several sides.

Finally, there are a number of patterns that explicitly forbid GNU Go to count territory at some intersections. This is used e. g. for false eyes that will eventually have to be filled in. Also, points for prisoners are added.

To fine tune this scheme, some revisions have been made to the influence computations that are relevant for territorial evaluation. This includes a reduced default attenuation and some revised pattern handling.

13.6 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 (with an attenuation of 3.0):

```
  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  1  1  1  0  0  0  0
  0  0  0  1  2  3  2  1  0  0  0
  0  0  1  3  5 11  5  3  1  0  0
  0  1  2  5 16 33 16  5  2  1  0
  0  1  3 11 33  X 33 11  3  1  0
  0  1  2  5 16 33 16  5  2  1  0
  0  0  1  3  5 11  5  3  1  0  0
  0  0  0  1  2  3  2  1  0  0  0
  0  0  0  0  1  1  1  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0
```

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.

13.7 The Influence Algorithm

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.

13.8 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.

13.9 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.

The `escape_route` field is modified by the code in 'surround.c' (see section Surrounded Dragons). It is divided by two for weakly surrounded dragons, and set to zero for surrounded ones.

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.

13.10 Break Ins

The code in '`breakin.c`' break-ins into territories that require deeper tactical reading and are thus impossible to detect for the influence module. It gets run after the influence module and revises its territory valuations.

The break-in code makes use of two public functions in '`readconnect.c`',

- int break_in(int str, const char goal[BOARDMAX], int *move)

  Returns WIN if `str` can connect to the area `goal[]` (which may or may not contain stones), if the string's owner gets the first move.

- int block_off(int str, const char goal[BOARDMAX], int *move)

  Returns WIN if `str` cannot connect to the area `goal[]` (which may or may not contain stones), if the other color moves first.

These functions are public front ends to their counterparts `recursive_break_in` and `recursive_block_off`, which call each other recursively.

The procedure is as follows: We look at all big (>= 10) territory regions as detected by the influence code. Using the computation of connection distances from readconnect.c, we compute all nearby vertices of this territory. We look for the closest safe stones belonging to the opponent.

For each such string `str` we call

- `break_in(str, territory)` if the opponent is assumed to be next to move,
- `block_off(str, territory)` if the territory owner is next.

If the break in is successful resp. the blocking unsuccessful, we shrink the territory, and see whether the opponent can still break in. We repeat this until the territory is shrunk so much that the opponent can no longer reach it.

To see the break in code in action run GNU Go on the file '`regression/games/break_in.sgf`' with the option `-d0x102000`. Among the traces you will find:

```
  Trying to break in from D7 to:
E9 (1)  F9 (1)  G9 (1)  E8 (1)  F8 (1)  G8 (1)
H8 (1)  G7 (1)  H7 (1)  J7 (1)  H6 (1)  J6 (1)
H5 (1)  J5 (1)  H4 (1)  J4 (1)  H3 (1)  J3 (1)
H2 (1)  J2 (1)
block_off D7, result 0 PASS (355, 41952 nodes, 0.73 seconds)
E9 (1)  F9 (1)  G9 (1)  E8 (1)  F8 (1)  G8 (1)
H8 (1)  G7 (1)  H7 (1)  J7 (1)  H6 (1)  J6 (1)
H5 (1)  J5 (1)  H4 (1)  J4 (1)  H3 (1)  J3 (1)
H2 (1)  J2 (1)
```

```
B:F4
  Erasing territory at E8 -b.
  Erasing territory at G3 -b.
  Now trying to break to smaller goal:
F9 (1)  G9 (1)  F8 (1)  G8 (1)  H8 (1)  G7 (1)
H7 (1)  J7 (1)  H6 (1)  J6 (1)  H5 (1)  J5 (1)
H4 (1)  J4 (1)  H3 (1)  J3 (1)  H2 (1)  J2 (1)
```

This means that the function `break_in` is called with the goal marked 'a' in the following diagram. The code attempts to find out whether it is possible to connect into this area from the string at `D7`.

```
   A B C D E F G H J
 9 . . . . a a a . . 9
 8 . . . . a a a a . 8
 7 . . . X O O a a a 7
 6 . . . X X X O a a 6
 5 . . . . + . . a a 5
 4 . . . X . . O a a 4
 3 . . . . X . . a a 3
 2 . . . . . . O a a 2
 1 . . . . . . . . . 1
   A B C D E F G H J
```

A breakin is found, so the goal is shrunk by removing `E9` and `J2`, then break_in is called again.

In order to see what reading is actually done in order to do this break in, you may load GNU Go in gtp mode, then issue the commands:

```
loadsgf break_in.sgf
= black

start_sgftrace
=

break_in D7 E9 F9 G9 E8 F8 G8 H8 G7 H7 J7 H6 J6 H5 J5 H4 J4 H3 J3 H2 J2
= 1 E8

finish_sgftrace vars.sgf
=

start_sgftrace
=

break_in D7 F9 G9 F8 G8 H8 G7 H7 J7 H6 J6 H5 J5 H4 J4 H3 J3 H2 J2
= 1 G7

finish_sgftrace vars1.sgf
```

This will produce two sgf files containing the variations caused by these calls to the breakin code. The second file, ‘`vars1.sgf`’ will contain quite a few variations.

The break in code makes a list of break ins which are found. When it is finished, the function `add_expand_territory_move` is called for each break in, adding a move reason.

The break in code is slow, and only changes a few moves by the engine per game. Nevertheless we believe that it contributes substantially to the strength of the program. The break in code is enabled by default in GNU Go 3.6 at level 10, and disabled at level 9. In fact, this is the **only** difference between levels 9 and 10 in GNU Go 3.6.

13.11 Surrounded Dragons

When is a dragon surrounded?

As has been pointed out by Bruce Wilcox, the geometric lines connecting groups of the opposite color are often important. It is very hard to prevent the escape of this ‘`O`’ dragon:

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

On the other hand, this dragon is in grave danger:

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

The difference between these two positions is that in the first, the ‘`O`’ dragon crosses the line connecting the top two ‘`X`’ stones.

Code in ‘`surround.c`’ implements a test for when a dragon is surrounded. The idea is to compute the convex hull of the surround set, that is, the set stones belonging to unfriendly neighbor dragons. If the dragon is contained within that hull. If it is, it is said to be surrounded.

In practice this scheme is modified slightly. The implementation uses various algorithms to compute distances and hostile stones are discarded from the surround set when a pair other hostile ones can be found which makes the considered one useless. For example, in the following position the bottom ‘O’ stone would get discarded.

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

Also, points are added to the surround set below stones on the second and third lines. This should account for the edge being a natural barrier.

In order to compute distances between corners of the convex hull a sorting by angle algorithm has been implemented. If the distance between a pair enclosing stones is large, the surround status gets decreased to `WEAKLY_SURROUNDED`, or even 0 for very large ones.

The sorting by angle must be explained. A small diagram will probably help :

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

The sorting algorithm will generate this:

```
.4.5.
3...6
..X..
2...7
.1.8.
```

That is, the points are sorted by ascending order of the measure of the angle S-G-O, where S is SOUTH, G the (approximated) gravity center of the goal, and O the position of the considered hostile stones.

The necessity of such sorting appears when one tries to measure distances between enclosing stones without sorting them, just by using directly the existing left and right corners arrays. In some positions, the results will be inconsistent. Imagine, for example a position where for instance the points 1,2,3,4,6 and 7 were in the left arrary, leaving only 5 and 8 in the right array. Because of the large distance between 5 and 8, the dragon would have declared weak surrounded or not surrounded at all. Such cases are rare but frequent enough to require the angle sorting.

The following position:

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

This is "more" surrounded than the following position:

```
O.XXXXXX.O
..........
.O......O.
```

In the second case, the surround status would be lowered to `WEAKLY_SURROUNDED`.

The surround code is used to modify the escape_route field in the dragon2 data array. When a dragon is WEAKLY_SURROUNDED, the escape_route is divided by 2. If the dragon is SURROUNDED, escape_route is simply set to 0.

13.12 Patterns used by the Influence module

This section explains the details of the pattern databases used for the influence computation.

First, we have the patterns in ‘`influence.db`’, which get matched symmetrically for both colors.

‘`E`’

These patterns add extra influence sources close to some shapes like walls. This tries to reflect their extra strength. These patterns are not used in the influence computations relevant for territory valuations, but they are useful for getting a better estimate of strengths of groups.

‘`I`’

These patterns add extra influence sources at typical invasion points. Usually they are of small strength. If they additionally have the class ‘`s`’, the extra influence source is added for both colors. Otherwise, only the player assumed to be next to move gets the benefit.

The patterns in ‘`barriers.db`’ get matched only for ‘`O`’ being the player next to move.

‘`A`’

Connections between ‘`X`’ stones that stop influence of ‘`O`’. They have to be tight enough that ‘`O`’ cannot break through, even though he is allowed to move first.

‘`D`’

Connections between ‘`O`’ stones that stop influence of ‘`X`’. The stones involved can be more loosely connected than those in ‘`A`’ patterns.

‘`B`’

These indicate positions of followup moves for the ‘`O`’ stone marked with ‘`Q`’ in the pattern. They are used to reduce the territory e. g. where a monkey jump is possible. Also, they are used in the computation of the followup influence, if the ‘`Q`’ stone was the move played (or a stone saved by the move played).

‘`t`’

These patterns indicate intersections where one color will not be able to get territory, for example in a false eye. The points are set with a call to the helper non_oterritory or non_xterritory in the action of the pattern.

The intrusion patterns (‘`B`’) are more powerful than the description above might suggest. They can be very helpful in identifying weak shapes (by adding an intrusion source for the opponent where he can break through). A negative inference for this is that a single bad ‘`B`’ pattern, e. g. one that has a wrong constraint, typically causes 5 to 10 `FAIL`s in the regression test suite.

Influence Patterns can have autohelper constraints as usual. As for the constraint attributes, there are (additionally to the usual ones ‘`O`’, ‘`o`’, ‘`X`’ and ‘`x`’), attributes ‘`Y`’ and ‘`FY`’. A pattern marked with ‘`Y`’ will only be used in the influence computations relevant for the territory valuation, while ‘`FY`’ patterns only get used in the other influence computations.

The action of an influence pattern is at the moment only used for non-territory patterns as mentioned above, and as a workaround for a problem with ‘`B`’ patterns in the followup influence.

To see why this workaround is necessary, consider the follwoing situation:

```
..XXX
.a*.O
.X.O.
..XXO
```

(Imagine that there is ‘`X`’ territory on the left.)

The move by ‘`O`’ at ‘`*`’ has a natural followup move at ‘`a`’. So, in the computation of the followup influence for ‘`*`’, there would be an extra influence source for ‘`O`’ at ‘`a`’ which would destroy a lot of black territory on the left. This would give a big followup value, and in effect the move ‘`*`’ would be treated as sente.

But of course it is gote, since ‘`X`’ will answer at ‘`a`’, which both stops the possible intrusion and threatens to capture ‘`*`’. This situation is in fact quite common.

Hence we need an additional constraint that can tell when an intrusion pattern can be used in followup influence. This is done by misusing the action line: An additional line

```
>return <condition>;
```

gets added to the pattern. The `condition` should be true if the intrusion cannot be stopped in sente. In the above example, the relevant intrusion pattern will have an action line of the form

```
>return (!xplay_attack(a,b));
```

where ‘`b`’ refers to the stone at ‘`*`’. In fact, almost all followup-specific constraints look similar to this.

13.13 Colored display and debugging of influence

There are various ways to obtain detailed information about the influence computations. Colored diagrams showing influence are possible from a colored xterm or rxvt window.

There are two options controlling when to generate diagrams:

- ‘`-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.

- ‘`--debug-influence 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.

The other options control which diagrams should be generated in these situations. You have to specify at least one of the options above and at least one of the options below to generate any output.

The options below must be combined with one of the two previous ones, or the diagram will not be printed. For example to print the influence diagram, you may combine 0x08 and 0x010, and use the option ‘`-m 0x018`’.

- ‘`-m 0x010`’ or ‘`-m 16`’

  Show colored display of territory/moyo/area regions.

    - territory: cyan
    - moyo: yellow
    - area: red

  This feature is very useful to get an immediate impression of the influence regions as GNU Go sees them.

- ‘`-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.

- '`-m 0x40`' or '`-m 64`'

  This generates two diagrams showing the permeability for black and white influence on the board.

- '`-m 0x80`' or '`-m 128`'

  This shows the strength of the influence sources for black and white across the board. You will see sources at each lively stone (with strength depending on the strength of this stone), and sources contributed by patterns.

- '`-m 0x100`' or '`-m 256`'

  This shows the attenuation with which the influence sources spread influence across the board. Low attenuation indicates far-reaching influence sources.

- '`-m 0x200`' or '`-m 512`'

  This shows the territory valuation of GNU Go. Each intersection is shown with a value between -1.0 and +1.0 (or -2 resp. +2 if there is a dead stone on this intersection). Positive values indicate territory for white. A value of -0.5 thus indicates a point where black has a 50% chance of getting territory.

Finally, there is the debug option '`-d 0x1`' which turns on on `DEBUG_INFLUENCE`. This gives a message for each influence pattern that gets matched. Unfortunately, these are way too many messages making it tedious to navigate the output. However, if you discover an influence source with '`-m 0x80`' that looks wrong, the debug output can help you to quickly find out the responsible pattern.

13.14 Influence Tuning with `view.pike`

A useful program in the regression directory is `view.pike`. To run it, you need Pike, which you may download from http://pike.ida.liu.se/.

The test case '`endgame:920`' fails in GNU Go 3.6. We will explain how to fix it.

Start by firing up view.pike on testcase endgame:920, e.g. by running `pike view.pike endgame:920` in the regression directory.

We see from the first view of move values that filling dame at P15 is valued highest with 0.17 points while the correct move at C4 is valued slightly lower with 0.16. The real problem is of course that C4 is worth a full point and thus should be valued about 1.0.

Now click on C4 to get a list of move reasons and move valuation information. Everything looks okay except that change in territory is 0.00 rather than 1.00 as it ought to be.

We can confirm this by choosing the "delta territory for..." button and again clicking C4. Now B5 should have been marked as one point of change in territory, but it's not.

Next step is to enter the influence debug tool. Press the "influence" button, followed by "black influence, dragons known," and "territory value." This shows the expected territory if black locally moves first everywhere (thus "black influence"). Here we can see that B5 is incorrectly considered as 1.0 points of white territory.

We can compare this with the territory after a white move at C4 (still assuming that black locally moves first everywhere after that) by pressing "after move influence for..." and clicking C4. This looks identical, as expected since delta territory was 0, but here it is correct that B5 is 1.0 points of territory for white.

The most straightforward solution to this problem is to add a non-territory pattern, saying that white can't get territory on B5 if black moves first. The nonterritory patterns are in '`barriers.db`'.

```
Pattern Nonterritory56

...
X.O
?O.

:8,t

eac
XbO
?Od

;oplay_attack(a,b,c,d,d)

>non_xterritory(e);
```

In these patterns it's always assumed that '`O`' moves first and thus it says that '`X`' can't get territory at `B5` ('`e`' in the pattern). Now we need to be a bit careful however since after '`O`' plays at '`a`' and '`X`' cuts in at '`b`', it may well happen that '`O`' needs to defend around '`d`', allowing '`X`' to cut at '`c`', possibly making the nonterritory assumption invalid. It's difficult to do this entirely accurate, but the constraint above is fairly conservative and should guarantee that '`a`' is safe in most, although not all, cases.

[<<] [>>] [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

14. Monte Carlo Go

In Monte Carlo Go the engine plays random games to the end, generating moves from a pattern database within the context of the algorithm UCT (upper confidence bounds applied to trees). This algorithm allowed the program MoGo (http://www.lri.fr/~gelly/MoGo.htm, to become the first computer program to defeat a professional while taking a 9 stone handicap (http://senseis.xmp.net/?MoGo).

GNU Go 3.8 can play 9x9 Go with the option '--monte-carlo' using the UCT algorithm. For command line options, see See section Invoking GNU Go: Command line options.

During reading, the engine makes incremental updates of local 3x3 neighborhood, suicide status, self-atari status, and number of stones captured, for each move.

GNU Go's simulations (Monte Carlo games) are pattern generated. The random playout move generation is distributed strictly proportional to move values computed by table lookup from a local context consisting of 3x3 neighborhood, opponent suicide status, own and opponent self-atari status, number of stones captured by own and opponent move, and closeness to the previous move. Let's call this local context simply "a pattern" and the table "pattern values" or simply "patterns".

There are three built-in databases that you can select using the option '--mc-patterns <name>', where '<name>' is one of

```
mc_montegnu_classic
mc_mogo_classic
mc_uniform
```

The first of these is an approximation of the previous random move generation algorithm. The `mogo_classic` pattern values is an approximation of the simulation policy used by early versions of MoGo, as published in the report odification of UCT with Patterns in Monte-Carlo Go RR-6062, by Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. The uniform pattern values is the so called "light" playout which chooses uniformly between all legal moves except single point proper eyes.

If you're not satisfied with these you can also tune your own pattern values with a pattern database file and load it at runtime with '--mc-load-patterns <name>' adding your own pattern database.

Let's start with the uniform pattern values. Those are defined by the file 'patterns/mc_uniform.db', which looks like this:

```
oOo
O*O
oO?

:0

oOo
O*O
---

:0

|Oo
|*O
+--

:0
```

Patterns are always exactly 3x3 in size with the move at the center point. The symbols are the usual for GNU Go pattern databases:

```
* move
O own stone (i.e. the same color as the color to move)
o own stone or empty
X opponent stone
x opponent stone or empty
? own stone, opponent stone, or empty
| vertical edge
- horizontal edge
+ corner
```

There's also a new symbol:

```
% own stone, opponent stone, empty, or edge
```

After the pattern comes a line starting with a colon. In all these patterns it says that the pattern has a move value of 0, i.e. must not be played. Unmatched patterns have a default value of 1. When all move values are zero for both players, the playout will stop. Including the three patterns above is important because otherwise the playouts would be likely to go on indefinitely, or as it actually happens be terminated at a hard-coded limit of 600 moves. Also place these patterns at the top of the database because when multiple patterns match, the first one is used, regardless of the values.

When using only these patterns you will probably notice that it plays rather heavy, trying hard to be solidly connected. This is because uniform playouts are badly biased with a high probability of non-solid connections being cut apart. To counter this you could try a pattern like

```
?X?
O*O
x.?

:20,near
```

to increase the probability that the one-point jump is reinforced when threatened. Here we added the property "near", which means that the pattern only applies if the previous move was played "near" this move. Primarily "near" means within the surrounding 3x3 neighborhood but it also includes certain cases of liberties of low-liberty strings adjacent to the previous move, e.g. the move to extend out of an atari created by the previous move. You have to read the source to find out the exact rules for nearness.

We could also be even more specific and say

```
?X?
O*O
x.?

:20,near,osafe,xsafe
```

to exclude the cases where this move is a self atari (osafe) or would be a self-atari for the opponent (xsafe).

It may also be interesting to see the effect of capturing stones. A catch-all pattern for captures would be

```
?X%
?*%
%%%

:10,ocap1,osafe
:20,ocap2
:30,ocap3
```

where we have used multiple colon lines to specify different move values depending on the number of captured stones; value 10 for a single captured stone, value 20 for two captured stones, and value 30 for three or more captured stones. Here we also excluded self-atari moves in the case of 1 captured stone in order to avoid getting stuck in triple-ko in the playouts (there's no superko detection in the playouts).

The full set of pattern properties is as follows:

`near`

The move is "near" the previous move.

`far`

The move is not "near" the previous move.

`osafe`

The move is not a self-atari.

`ounsafe`

The move is a self-atari.

`xsafe`

The move would not be a self-atari for the opponent.

`xunsafe`

The move would be a self-atari for the opponent.

`xsuicide`

The move would be suicide for the opponent

`xnosuicide`

The move would not be suicide for the opponent.

`ocap0`

The move captures zero stones.

`ocap1`

The move captures one stone.

`ocap2`

The move captures two stones.

`ocap3`

The move captures three or more stones.

`ocap1+`

The move captures one or more stones.

`ocap1-`

The move captures at most one stone.

`ocap2+`

The move captures two or more stones.

`ocap2-`

The move captures at most two stones.

`xcap0`

An opponent move would capture zero stones.

`xcap1`

An opponent move would capture one stone.

`xcap2`

An opponent move would capture two stones.

`xcap3`

An opponent move would capture three or more stones.

`xcap1+`

An opponent move would capture one or more stones.

`xcap1-`

An opponent move would capture at most one stone.

`xcap2+`

An opponent move would capture two or more stones.

`xcap2-`

An opponent move would capture at most two stones.

These can be combined arbitrarily but all must be satisfied for the pattern to take effect. If contradictory properties are combined, the pattern will never match.

14.0.1 Final Remarks

Move values are unsigned 32-bit integers. To avoid overflow in computations it is highly recommended to keep the values below 10000000 or so.
There is no speed penalty for having lots of patterns in the database. The average time per move is approximately constant (slightly dependent on how often stones are captured or become low on liberties) and the time per game mostly depends on the average game length.
For more complex pattern databases, see ‘`patterns/mc_montegnu_classic.db`’ and ‘`patterns/mc_mogo_classic.db`’.

Nobody really knows how to tune the random playouts to get as strong engine as possible. Please play with this and report any interesting findings, especially if you're able to make it substantially stronger than the ‘`montegnu_classic`’ patterns.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

15. 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.c', which compiles joseki pattern databases from SGF files.

If you want to use the board library in your own program, you need all the .c-files listed under libboard_SOURCES in engine/Makefile.am, and the files in the directories sgf/ and utils/. Then you should include engine/board.h in your code.

The library consists of the following files:

'board.h'

The public interface to the board library.

'board.c'

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

'boardlib.c'

This contains all global variable of the board library.

'hash.c'

Code for hashing go positions.

'sgffile.c'

Implementation of output file in SGF format.

'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.

15.1 Board Data structures

The basic data structures of the board correspond tightly to the board_state struct described in See section The board_state struct. They are all stored in global variables for efficiency reasons, the most important of which are:

```
int           board_size;
Intersection  board[MAXSIZE];
int           board_ko_pos;

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 section Incremental Board data structures). The variable hashdata contains information about the hash value for the current position (see section Hashing of Positions).

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.

15.2 The Board Array

GNU Go represents the board in a one-dimensional array called board. For some purposes a two dimensional indexing of the board by parameters (i, j) might be used.

The board array includes out-of-board markers around the board. To make the relation to the old two-dimensional board representation clear, this figure shows how the 1D indices correspond to the 2D indices when MAX_BOARD is 7.

```
  j  -1   0   1   2   3   4   5   6
i +----------------------------------
-1|   0   1   2   3   4   5   6   7
 0|   8   9  10  11  12  13  14  15
```

```
  1|  16  17  18  19  20  21  22  23
  2|  24  25  26  27  28  29  30  31
  3|  32  33  34  35  36  37  38  39
  4|  40  41  42  43  44  45  46  47
  5|  48  49  50  51  52  53  54  55
  6|  56  57  58  59  60  61  62  63
  7|  64  65  66  67  68  69  70  71  72
```

To convert between a 1D index `pos` and a 2D index `(i, j)`, the macros `POS`, `I`, and `J` are provided, defined as below:

```
#define POS(i, j)    ((MAX_BOARD + 2) + (i) * (MAX_BOARD + 1) + (j))
#define I(pos)       ((pos) / (MAX_BOARD + 1) - 1)
#define J(pos)       ((pos) % (MAX_BOARD + 1) - 1)
```

All 1D indices not corresponding to points on the board have the out of board marker value `GRAY`. Thus if `board_size` and `MAX_BOARD` both are 7, this looks like

```
   j  -1   0   1   2   3   4   5   6
i +----------------------------------
-1|   #   #   #   #   #   #   #   #
 0|   #   .   .   .   .   .   .   .
 1|   #   .   .   .   .   .   .   .
 2|   #   .   .   .   .   .   .   .
 3|   #   .   .   .   .   .   .   .
 4|   #   .   .   .   .   .   .   .
 5|   #   .   .   .   .   .   .   .
 6|   #   .   .   .   .   .   .   .
 7|   #   #   #   #   #   #   #   #   #
```

The indices marked '`#`' have value `GRAY`. If `MAX_BOARD` is 7 and `board_size` is only 5:

```
   j  -1   0   1   2   3   4   5   6
i +----------------------------------
-1|   #   #   #   #   #   #   #   #
 0|   #   .   .   .   .   .   #   #
 1|   #   .   .   .   .   .   #   #
 2|   #   .   .   .   .   .   #   #
 3|   #   .   .   .   .   .   #   #
 4|   #   .   .   .   .   .   #   #
 5|   #   #   #   #   #   #   #   #
 6|   #   #   #   #   #   #   #   #
 7|   #   #   #   #   #   #   #   #   #
```

Navigation on the board is done by the `SOUTH`, `WEST`, `NORTH`, and `EAST` macros,

```
#define NS           (MAX_BOARD + 1)
#define WE           1
#define SOUTH(pos)   ((pos) + NS)
#define WEST(pos)    ((pos) - 1)
#define NORTH(pos)   ((pos) - NS)
#define EAST(pos)    ((pos) + 1)
```

There are also shorthand macros `SW`, `NW`, `NE`, `SE`, `SS`, `WW`, `NN`, `EE` for two step movements.

Any movement from a point on the board to an adjacent or diagonal vertex is guaranteed to produce a valid index into the board array, and the color found is GRAY if it is not on the board. To do explicit tests for out of board there are two macros

```
#define ON_BOARD(pos) (board[pos] != GRAY)
#define ON_BOARD1(pos) (((unsigned) (pos) < BOARDSIZE) && board[pos] != GRAY)
```

where the first one should be used in the algorithms and the second one is useful for assertion tests.

The advantage of a one-dimensional board array is that it gives a significant performance advantage. We need only one variable to determine a board position, which means that many functions need less arguments. Also, often one computation is sufficient for 1D-coordinate where we would need two with two 2D-coordinates: If we, for example, want to have the coordinate of the upper right of `pos`, we can do this with `NORTH(EAST(pos))` instead of `(i+1, j-1)`.

Important: The 2D coordinate `(-1,-1)`, which is used for pass and sometimes to indicate no point, maps to the 1D coordinate `0`, not to `-1`. Instead of a plain `0`, use one of the macros `NO_MOVE` or `PASS_MOVE`.

A loop over multiple directions is straightforwardly written:

```
  for (k = 0; k < 4; k++) {
    int d = delta[k];
    do_something(pos + d);
  }
```

The following constants are useful for loops over the entire board and allocation of arrays with a 1-1 mapping to the board.

```
#define BOARDSIZE    ((MAX_BOARD + 2) * (MAX_BOARD + 1) + 1)
#define BOARDMIN     (MAX_BOARD + 2)
#define BOARDMAX     (MAX_BOARD + 1) * (MAX_BOARD + 1)
```

`BOARDSIZE` is the actual size of the 1D board array, `BOARDMIN` is the first index corresponding to a point on the board, and `BOARDMAX` is one larger than the last index corresponding to a point on the board.

Often one wants to traverse the board, carrying out some function at every vertex. Here are two possible ways of doing this:

```
    int m, n;
    for (m = 0; m < board_size; m++)
      for (n = 0; n < board_size; n++) {
        do_something(POS(m, n));
      }
```

Or:

```
    int pos;
    for (pos = BOARDMIN; pos < BOARDMAX; pos++) {
      if (ON_BOARD(pos))
        do_something(pos);
    }
```

15.3 Incremental Board data structures

In addition to the global board state, the algorithms in ‘`board.c`’ implement a method of incremental 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 1D board 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 origin;                      /* Coordinates of "origin", i.e. */
                                     /* "upper left" stone. */
    int liberties;                   /* Number of liberties. */
    int libs[MAX_LIBERTIES];         /* Coordinates of liberties. */
    int neighbors;                   /* Number of neighbor 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

```
  static int string_number[BOARDMAX];
```

which contains indices into the `string` array. This information is only valid at nonempty vertices, however, so it is necessary to first verify that `board[pos] != EMPTY`.

The `string_data` structure does not include an array of the stone coordinates. This information is stored in a separate array:

```
  static int next_stone[BOARDMAX];
```

This array implements 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[BOARDMAX];
  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[pos]` has the same value as `liberty_mark`, then `pos` 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 ‘`board`’ 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

```
    pos = FIRST_STONE(s);
    do {
        ...
        pos = NEXT_STONE(pos);
    } while (!BACK_TO_FIRST_STONE(s, pos));
```

traverses the stones of the string with number ‘`s`’ exactly once, with `pos` holding the coordinates. In general `pos` is used as board coordinate and ‘`s`’ as an index into the `string` array or sometimes a pointer to an entry in the `string` array.

15.4 Some Board 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(pos, color, ... )) {
       ...    [potentially lots of code here]
       popgo();
  }
```

In case the move is a ko capture, the legality of the capture is subject to the komaster scheme (see section Ko Handling).

- `int trymove(int pos, int color, const char *message)`

  Returns true if `(pos)` 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 `str` if this is not `0`. The value of `str` can be NO_MOVE if it is not needed but otherwise the location of `str` is included in the comment.

- `int tryko(int pos, int color, const char *message)` `tryko()` pushes the position onto the stack, and makes a move `pos` 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 pos, int color, const char *message, int str, 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 Ko Handling).

As you see, `trymove()` plays a move which can be easily retracted (with `popgo()`) and it is call thousands of times per actual game move as GNU Go analyzes the board position. By contrast the function `play_move()` plays a move which is intended to be permanent, though it is still possible to undo it if, for example, the opponent retracts a move.

- `void play_move(int pos, int color)`

  Play a move. 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.

  In spite of the name "permanent move", this move can (usually) be unplayed by `undo_move()`, but it is significantly more costly than unplaying a temporary move. There are limitations on the available move history, so under certain circumstances the move may not be possible to unplay at a later time.

- `int undo_move(int n)`

  Undo '`n`' permanent moves. Returns 1 if successful and 0 if it fails. If '`n`' moves cannot be undone, no move is undone.

Other board functions are documented in See section Board Utilities.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

16. 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.

16.1 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.

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

17. 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, see section The Go Text Protocol.

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 Game Format (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. Games can be saved to SGF files and then later be read back again. These are also within `libengine.a`.

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

17.1 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.

17.2 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
```

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`.

GNU Go uses two different representations of the board, for most purposes a one-dimensional one, but for a few purposes a two dimensional one (see section The Board Library). The one-dimensional board was introduced before GNU Go 3.2, while the two-dimensional board dates back to the ancestral program written by Man Lung Li before 1995. The API still uses the two-dimensional board, so the API functions have not changed much since GNU Go 3.0.

17.3 The board_state struct

A basic data structure in the engine is the `board_state` struct. This structure is internal to the engine and is defined in 'liberty.h'.

```
typedef unsigned char Intersection;

struct board_state {
  int board_size;

  Intersection board[BOARDSIZE];
  int board_ko_pos;
  int black_captured;
  int white_captured;

  Intersection initial_board[BOARDSIZE];
  int initial_board_ko_pos;
  int initial_white_captured;
  int initial_black_captured;
  int move_history_color[MAX_MOVE_HISTORY];
  int move_history_pos[MAX_MOVE_HISTORY];
  int move_history_pointer;

  float komi;
  int move_number;
};
```

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 board state contains an array of `Intersection`'s representing the board. The move history is contained in the struct. Also contained in the struct is the location of a ko (`EMPTY`) if the last move was not a ko capture, the komi, the number of captures, and corresponding data for the initial position at the beginning of the move history.

17.4 Functions which manipulate a Position

All the functions in the engine that manipulate Positions have names prefixed by `gnugo_`. These functions still use the two-dimensional representation of the board (see section The Board Array). Here is a complete list, as prototyped in 'gnugo.h':

`void init_gnugo(float memory)`

Initialize the gnugo engine. This needs to be called once only.

`void gnugo_clear_board(int boardsize)`

Clear the board.

`void gnugo_set_komi(float new_komi)`

Set the komi.

`void gnugo_add_stone(int i, int j, int color)`

Place a stone on the board

`void gnugo_remove_stone(int i, int j)`

Remove a stone from the board

`int gnugo_is_pass(int i, int j)`

Return true if (i,j) is PASS_MOVE

`void gnugo_play_move(int i, int j, int color)`

Play a move and start the clock

`int gnugo_undo_move(int n)`

Undo n permanent moves. Returns 1 if successful and 0 if it fails. If n moves cannot be undone, no move is undone.

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

Perform the moves and place the stones from the SGF node on the board. Return the color of the player whose turn it is to move.

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

Play the moves in ROOT UNTIL movenumber is reached. Return the color of the player whose turn it is to move.

`int gnugo_is_legal(int i, int j, int color)`

Interface to `is_legal()`.

`int gnugo_is_suicide(int i, int j, int color)`

Interface to `is_suicide()`.

`int gnugo_placehand(int handicap)`

Interface to placehand. Sets up handicap pieces and returns the number of placed handicap stones.

`void gnugo_recordboard(SGFNode *root)`

Interface to `sgffile_recordboard()`

`int gnugo_sethand(int handicap, SGFNode *node)`

Interface to placehand. Sets up handicap stones and returns the number of placed handicap stones, updating the sgf file

`float gnugo_genmove(int *i, int *j, int color, int *resign)`

Interface to `genmove()`.

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

Interface to `attack()`.

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

Interface to `find_defense()`.

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

Interface to `who_wins()`.

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

Put upper and lower score estimates into `*upper`, `*lower` and return the average. A positive score favors white. In computing the upper bound, `CRITICAL` dragons are awarded to white; in computing the lower bound, they are awarded to black.

`void gnugo_examine_position(int color, int how_much)`

Interface to `examine_position`.

`int gnugo_get_komi()`

Report the komi.

`void gnugo_get_board(int b[MAX_BOARD][MAX_BOARD])`

Place the board into the 'b' array.

`int gnugo_get_boardsize()`

Report the board size.

`int gnugo_get_move_number()`

Report the move number.

17.5 Game handling

The functions (in see section Functions which manipulate a Position) 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;

  int       to_move;            /* whose move it currently is */
  SGFTree   game_record;        /* Game record in sgf format. */

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

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

The meaning of `handicap` should be obvious. `to_move` is the color of the side whose turn it is to move.

The SGF tree `game_record` is used to store all the moves in the entire game, including a header node which contains, among other things, komi and handicap.

If one or both of the opponents is the computer, the field `computer_player` is used. Otherwise it can be ignored.

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. 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`.

17.5.1 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 prototyped in 'gnugo.h':

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

Initialize the `Gameinfo` structure.

`void gameinfo_print(Gameinfo *ginfo)`

Print a gameinfo.

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

Reads header info from sgf structure and sets the appropriate variables.

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

Make a move in the game. Return 1 if the move was legal. In that case the move is actually done. Otherwise return 0.

`int gameinfo_play_sgftree_rot(Gameinfo *gameinfo, SGFNode *head, const char *untilstr, int orientation)`

Play the moves in an SGF tree. Walk the main variation, actioning the properties into the playing board. Returns the color of the next move to be made. Head is an sgf tree. Untilstr is an optional string of the form either 'L12' or '120' which tells it to stop playing at that move or move number. When debugging, this is the location of the move being examined.

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

Same as previous function, using standard orientation.

18. Utility Functions

In this Chapter, we document some of the utilities which may be called from the GNU Go engine.

18.1 General Utilities

Utility functions from 'engine/utils.c'. Many of these functions underlie autohelper functions (see section Autohelper Functions).

- `void change_dragon_status(int dr, int status)`

  Change the status of all the stones in the dragon at `dr`.

- `int defend_against(int move, int color, int apos)`

  Check whether a move at `move` stops the enemy from playing at (apos).

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

  Returns true if `color` can cut at `pos`, or if connection through `pos` is inhibited. This information is collected by `find_cuts()`, using the B patterns in the connections database.

- `int does_attack(int move, int str)`

  returns true if the move at `move` attacks `str`. This means that it captures the string, and that `str` is not already dead.

- `int does_defend(int move, int str)`

  `does_defend(move, str)` returns true if the move at `move` defends `str`. This means that it defends the string, and that `str` can be captured if no defense is made.

- `int somewhere(int color, int last_move, ...)`

  Example: `somewhere(WHITE, 2, apos, bpos, cpos)`. Returns true if one of the vertices listed satisfies `board[pos]==color`. Here num_moves is the number of moves minus one. If the check is true the dragon is not allowed to be dead. This check is only valid if `stackp==0`.

- `int visible_along_edge(int color, int apos, int bpos)`

  Search along the edge for the first visible stone. Start at apos and move in the direction of bpos. Return 1 if the first visible stone is of the given color. It is required that apos and bpos are at the same distance from the edge.

- `int test_symmetry_after_move(int move, int color, int strict)`

  Is the board symmetric (or rather antisymmetric) with respect to mirroring in tengen after a specific move has been played? If the move is PASS_MOVE, check the current board. If strict is set we require that each stone is matched by a stone of the opposite color at the mirrored vertex. Otherwise we only require that each stone is matched by a stone of either color.

- `int play_break_through_n(int color, int num_moves, ...)`

  The function `play_break_through_n()` 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, ...)`
- `int play_attack_defend2_n(int color, int do_attack, int num_moves, ...)`

  The function `play_attack_defend_n()` 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. Conversely, `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 of these functions is to set up a ladder in an autohelper and see whether it works or not.

- `int play_connect_n(int color, int do_connect, int num_moves, ...)`

  Plays a sequence of moves, alternating between the players and starting with `color`. After having played through the sequence, the two last coordinates give two targets that should be connected or disconnected, depending on the value of do_connect. If there is no stone present to connect or disconnect, it is assumed that the connection has failed. 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 connection/disconnection is tested as if nothing special happened. Ultimately the connection is decided by the functions `string_connect` and `disconnect` (see section Connection Reading).

- `void set_depth_values(int level)`

  It is assumed in reading a ladder if `stackp >= depth` that as soon as a bounding stone is in atari, the string is safe. Similar uses are made of the other depth parameters such as `backfill_depth` and so forth. In short, simplifying assumptions are made when `stackp` is large. Unfortunately any such scheme invites the "horizon effect," in which a stalling move is perceived as a win, by pushing the refutation past the "horizon"—the value of `stackp` in which the reading assumptions are relaxed. To avoid the depth it is sometimes necessary to increase the depth parameters. This function can be used to set the various reading depth parameters. If `mandated_depth_value` is not -1 that value is used; otherwise the depth values are set as a function of level. The parameter `mandated_depth_value` can be set at the command line to force a particular value of depth; normally it is -1.

- `void modify_depth_values(int n)`

  Modify the various tactical reading depth parameters. 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)`

  `modify_depth_values(1)`.

- `void decrease_depth_values(void)`

  `modify_depth_values(-1)`.

- `void restore_depth_values()`

  Sets `depth` and so forth to their saved values.

- `void set_temporary_depth_values(int d, int b, int b2, int bc, int ss, int br, int f, int k)`

  Explicitly set the depth values. This function is currently never called.

- `int confirm_safety(int move, int color, int *defense_point, char safe_stones[BOARDMAX])`

  Check that the move at color doesn't involve any kind of blunder, regardless of size.

- `float blunder_size(int move, int color, int *defense_point, char safe_stones[BOARDMAX])`

  This function will detect some blunders. If the move reduces the number of liberties of an adjacent friendly string, there is a danger that the move could backfire, so the function checks that no friendly worm which was formerly not attackable becomes attackable, and it checks that no opposing worm which was not defendable becomes defendable. It returns the estimated size of the blunder, or 0.0 if nothing bad has happened. The array `safe_stones[]` contains the stones that are supposedly safe after `move`. It may be `NULL`. For use when called from `fill_liberty()`, this function may optionally return a point of defense, which, if taken, will presumably make the move at `move` safe on a subsequent turn.

- `int double_atari(int move, int color, float *value, char safe_stones[BOARDMAX])`

  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. If `safe_stones != NULL`, then only attacks on stones marked as safe are tried. The value of the double atari attack is returned in value (unless value is `NULL`), and the attacked stones are marked unsafe.

- `void unconditional_life(int unconditional_territory[BOARDMAX], 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 worm 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. When no more worm can be removed in this way, the remaining ones are unconditionally alive. After this, unconditionally dead opponent worms and unconditional territory are identified. To find these, we continue from the position obtained at the end of the previous operation (only unconditionally alive strings remain for color) with the following steps:

  1. Play opponent stones on all liberties of the unconditionally alive strings except where illegal. (That the move order may determine exactly which liberties can be played legally is not important. Just pick an arbitrary order).
  2. Recursively extend opponent strings in atari, except where this would be suicide.
  3. Play an opponent stone anywhere it can get two empty neighbors. (I.e. split big eyes into small ones).
  4. an opponent stone anywhere it can get one empty neighbor. (I.e. reduce two space eyes to one space eyes.) Remaining opponent strings in atari and remaining liberties of the unconditionally alive strings constitute the unconditional territory. Opponent strings from the initial position placed on unconditional territory are unconditionally dead. On return, `unconditional_territory[][]` is 1 where color has unconditionally alive stones, 2 where it has unconditional territory, and 0 otherwise.

- `void who_wins(int color, FILE *outfile)`

  Score the game and determine the winner

- `void find_superstring(int str, int *num_stones, int *stones)`

  Find the stones of 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
     ..
     ```

```
00
```

4. Diagonal connection where both adjacent intersections are empty.

```
.0
0.
```

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

- `void find_superstring_liberties(int str, int *num_libs, int *libs, int liberty_cap)`

  This function computes the superstring at `str` as described above, but omitting connections of type 5. Then it constructs a list of liberties of the superstring which are not already liberties of `str`. If `liberty_cap` is nonzero, only liberties of substrings of the superstring which have fewer than `liberty_cap` liberties are generated.

- `void find_proper_superstring_liberties(int str, int *num_libs, int *libs, int liberty_cap)`

  This function is the same as find_superstring_liberties, but it omits those liberties of the string `str`, presumably since those have already been treated elsewhere. If `liberty_cap` is nonzero, only liberties of substrings of the superstring which have at most `liberty_cap` liberties are generated.

- `void find_superstring_stones_and_liberties(int str, int *num_stones, int *stones, int *num_libs, int *libs, int liberty_cap)`

  This function computes the superstring at `str` as described above, but omitting connections of type 5. Then it constructs a list of liberties of the superstring which are not already liberties of `str`. If liberty_cap is nonzero, only liberties of substrings of the superstring which have fewer than liberty_cap liberties are generated.

- `void superstring_chainlinks(int str, int *num_adj, int adjs[MAXCHAIN], int liberty_cap)`

  analogous to chainlinks, this function finds boundary chains of the superstring at `str`, including those which are boundary chains of `str` itself. If `liberty_cap != 0`, only those boundary chains with `<= liberty_cap` liberties are reported.

- `void proper_superstring_chainlinks(int str, int *num_adj, int adjs[MAXCHAIN], int liberty_cap)`

  analogous to chainlinks, this function finds boundary chains of the superstring at `str`, omitting those which are boundary chains of `str` itself. If `liberty_cap != 0`, only those boundary chains with `<= liberty_cap` liberties are reported.

- `void start_timer(int n)`

  Start a timer. GNU Go has four internal timers available for assessing the time spent on various tasks.

- `double time_report(int n, const char *occupation, int move, double mintime)`

  Report time spent and restart the timer. Make no report if elapsed time is less than mintime.

18.2 Print Utilities

Functions in `'engine/printutils.c'` do formatted printing similar to `printf` and its allies. The following formats are recognized:

- `%c`, `%d`, `%f`, `%s`, `%x`

  These have their usual meaning in formatted output, printing a character, integer, float, string or hexadecimal, respectively.

- `%o`

  `Outdent.' Normally output is indented by `2*stackp` spaces, so that the depth can be seen at a glance in traces. At the beginning of a format, this `%o` inhibits the indentation.

- `%H`

  Print a hashvalue.

- `%C`

  Print a color as a string.

- `%m`, `%2m` (synonyms)

  Takes 2 integers and writes a move, using the two dimensional board representation (see section The Board Array)

- `%1m`

  Takes 1 integers and writes a move, using the one dimensional board representation (see section The Board Array)

We list the non statically declared functions in `'printutils.c'`.

- `void gfprintf(FILE *outfile, const char *fmt, ...)`

  Formatted output to `'outfile'`.

- `int gprintf(const char *fmt, ...)`

  Formatted output to stderr. Always returns 1 to allow use in short-circuit logical expressions.

- `int mprintf(const char *fmt, ...)`

Formatted output to stdout.

- `DEBUG(level, fmt, args...)`

  If `level & debug`, do formatted output to stderr. Otherwise, ignore.

- `void abortgo(const char *file, int line, const char *msg, int pos)`

  Print debugging output in an error situation, then exit.

- `const char * color_to_string(int color)`

  Convert a color value to a string

- `const char * location_to_string(int pos)`

  Convert a location to a string

- `void location_to_buffer(int pos, char *buf)`

  Convert a location to a string, writing to a buffer.

- `int string_to_location(int boardsize, char *str, int *m, int *n)`

  Get the `(m, n)` coordinates in the standard GNU Go coordinate system from the string `str`. This means that '`m`' is the nth row from the top and '`n`' is the column. Both coordinates are between 0 and `boardsize-1`, inclusive. Return 1 if ok, otherwise return 0;

- `int is_hoshi_point(int m, int n)` True if the coordinate is a hoshi point.
- `void draw_letter_coordinates(FILE *outfile)` Print a line with coordinate letters above the board.
- `void simple_showboard(FILE *outfile)`

  Bare bones version of `showboard(0)`. No fancy options, no hint of color, and you can choose where to write it.

The following functions are in '`showbord.c`'. Not all public functions in that file are listed here.

`void showboard(int xo)`

Show go board.

```
xo=0:      black and white XO board for ascii game
xo=1:      colored dragon display
xo=2:      colored eye display
xo=3:      colored owl display
xo=4:      colored matcher status display
```

`const char * status_to_string(int status)`

Convert a status value to a string.

`const char * safety_to_string(int status)`

Convert a safety value to a string.

`const char * result_to_string(int result)`

Convert a read result to a string

18.3 Board Utilities

The functions documented in this section are from '`board.c`'. Other functions in '`board.c`' are described in See section Some Board Functions.

- `void store_board(struct board_state *state)`

  Save board state.

- `void restore_board(struct board_state *state)`

  Restore a saved board state.

- `void clear_board(void)`

  Clear the internal board.

- `void dump_stack(void)`

  for use under GDB prints the move stack.

- `void add_stone(int pos, int color)`

  Place a stone on the board and update the board_hash. This operation destroys all move history.

- `void remove_stone(int pos)`

  Remove a stone from the board and update the board_hash. This operation destroys the move history.

- `int is_pass(int pos)`

  Test if the move is a pass or not. Return 1 if it is.

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

  Determines whether the move `color` at `pos` is legal.

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

  Determines whether the move `color` at `pos` would be a suicide. This is the case if

  1. There is no neighboring empty intersection.
  2. There is no neighboring opponent string with exactly one liberty.
  3. There is no neighboring friendly string with more than one liberty.

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

  Determines whether the move `color` at `pos` would be an illegal ko capture.

- `int is_edge_vertex(int pos)`

  Determine whether vertex is on the edge.

- `int edge_distance(int pos)`

  Distance to the edge.

- `int is_corner_vertex(int pos)`

  Determine whether vertex is a corner.

- `int get_komaster()`
- `int get_kom_pos()`

  Public functions to access the variable `komaster` and `kom_pos`, which are static in ‘`board.c`’.

Next we come to `countlib()` and its allies, which address the problem of determining how many liberties a string has. Although `countlib()` addresses this basic question, other functions can often get the needed information more quickly, so there are a number of different functions in this family.

- `int countlib(int str)`

  Count the number of liberties of the string at `pos`. There must be a stone at this location.

- `int findlib(int str, int maxlib, int *libs)`

  Find the liberties of the string at `str`. This location must not be empty. The locations of up to maxlib liberties are written into `libs[]`. 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 `libs[]` accordingly.

- `int fastlib(int pos, int color, int ignore_captures)`

  Count the liberties a stone of the given color would get if played at `pos`. The intent of this function is to be as fast as possible, not necessarily complete. But if it returns a positive value (meaning it has succeeded), the value is guaranteed to be correct. Captures are ignored based if the `ignore_captures` field is nonzero. The location `pos` must be empty. The function fails if there are more than two neighbor strings of the same color. In this case, the return value is -1. Captures are handled in a very limited way, so if ignore_capture is 0, and a capture is required, it will often return -1.

- `int approxlib(int pos, int color, int maxlib, int *libs)`

  Find the liberties a stone of the given color would get if played at `pos`, ignoring possible captures of opponent stones. The location `pos` must be empty. If `libs != NULL`, the locations of up to `maxlib` liberties are written into `libs[]`. The counting of liberties may or may not be halted when `maxlib` is reached. The number of liberties found is returned, which may be less than the total number of liberties if `maxlib` is small. 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 `libs[]` accordingly.

- `int accuratelib(int pos, int color, int maxlib, int *libs)`

  Find the liberties a stone of the given color would get if played at `pos`. This function takes into consideration all captures. Its return value is exact in that sense it counts all the liberties, unless `maxlib` allows it to stop earlier. The location `pos` must be empty. If `libs != NULL`, the locations of up to `maxlib` liberties are written into `libs[]`. The counting of liberties may or may not be halted when `maxlib` is reached. The number of found liberties is returned. This function guarantees that liberties which are not results of captures come first in `libs[]` array. To find whether all the liberties starting from a given one are results of captures, one may use `if (board[libs[k]] != EMPTY)` construction. 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 `libs[]` accordingly.

Next we have some general utility functions.

- `int count_common_libs(int str1, int str2)`

  Find the number of common liberties of the two strings.

- `int find_common_libs(int str1, int str2, int maxlib, int *libs)`

Find the common liberties of the two strings. The locations of up to `maxlib` common liberties are written into `libs[]`. The full number of common liberties is returned. If you want the locations of all common liberties, whatever their number, you should pass `MAXLIBS` as the value for `maxlib` and allocate space for `libs[]` accordingly.

- `int have_common_lib(int str1, int str2, int *lib)`

  Determine whether two strings have at least one common liberty. If they do and `lib != NULL`, one common liberty is returned in `*lib`.

- `int countstones(int str)`

  Report the number of stones in a string.

- `int findstones(int str, int maxstones, int *stones)`

  Find the stones of the string at `str`. The location must not be empty. The locations of up to maxstones stones are written into `stones[]`. The full number of stones is returned.

- `int chainlinks(int str, int adj[MAXCHAIN])`

  This very useful function returns (in the `adj` array) the chains surrounding the string at `str`. The number of chains is returned.

- `int chainlinks2(int str, int adj[MAXCHAIN], int lib)`

  Returns (in `adj` array) those chains surrounding the string at `str`, which has exactly `lib` liberties. The number of such chains is returned.

- `int chainlinks3(int str, int adj[MAXCHAIN], int lib)`

  Returns (in `adj` array) the chains surrounding the string at `str`, which have less or equal `lib` liberties. The number of such chains is returned.

- `int extended_chainlinks(int str, int adj[MAXCHAIN], int both_colors)`

  Returns (in the `adj` array) the opponent strings being directly adjacent to `str` or having a common liberty with `str`. The number of such strings is returned. If the both_colors parameter is true, also own strings sharing a liberty are returned.

- `int find_origin(int str)`

  Find the origin of a string, i.e. the point with the smallest 1D board coordinate. The idea is to have a canonical reference point for a string.

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

  Determine whether a move by color at `pos` would be a self atari, i.e. whether it would get more than one liberty. This function returns true also for the case of a suicide move.

- `int liberty_of_string(int pos, int str)`

  Returns true if `pos` is a liberty of the string at `str`.

- `int second_order_liberty_of_string(int pos, int str)`

  Returns true if `pos` is a second order liberty of the string at str.

- `int neighbor_of_string(int pos, int str)`

  Returns true if `pos` is adjacent to the string at `str`.

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

  Returns true if `pos` has a neighbor of `color`.

- `int same_string(int str1, int str2)`

  Returns true if `str1` and `str2` belong to the same string.

- `int adjacent_strings(int str1, int str2)`

  Returns true if the strings at `str1` and `str2` are adjacent.

- `int is_ko(int pos, int color, int *ko_pos)`

  Return true if the move `pos` by `color` is a ko capture (whether capture is legal on this move or not). If so, and if `ko_pos` is not a `NULL` pointer, then `*ko_pos` returns the location of the captured ko stone. If the move is not a ko capture, `*ko_pos` is set to 0. A move is a ko capture if and only if

  1. All neighbors are opponent stones.
  2. The number of captured stones is exactly one.

- `int is_ko_point(int pos)`

  Return true if `pos` is either a stone, which if captured would give ko, or if `pos` is an empty intersection adjacent to a ko stone.

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

  Returns 1 if at least one string is captured when color plays at `pos`.

- `void mark_string(int str, char mx[BOARDMAX], char mark)`

For each stone in the string at pos, 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 pos are guaranteed to become marked. The rest of the string may or may not become marked. (In the current implementation, it will.)

- `int move_in_stack(int pos, int cutoff)`

  Returns true if at least one move has been played at pos at deeper than level `cutoff` in the reading tree.

- `int stones_on_board(int color)`

  Return the number of stones of the indicated color(s) on the board. This only counts 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.

18.4 Utilities from ‘`engine/influence.c`’

We will only list here a portion of the public functions in `influence.c`. The influence code is invoked through the function `compute_influence` (see section Where influence gets used in the engine). It is invoked as follows.

- `void compute_influence(int color, const char safe_stones[BOARDMAX], const float strength[BOARDMAX], struct influence_data *q, int move, const char *trace_message)`

  Compute the influence values for both colors. The caller must

  - set up the `board[]` state
  - mark safe stones with `INFLUENCE_SAFE_STONE`, dead stones with 0
  - mark stones newly saved by a move with `INFLUENCE_SAVED_STONE` (this is relevant if the influence_data *q is reused to compute a followup value for this move).

  Results will be stored in q. `move` has no effects except toggling debugging. Set it to -1 for no debug output at all (otherwise it will be controlled by the ‘`-m`’ command line option). It is assumed that `color` is in turn to move. (This affects the barrier patterns (class A, D) and intrusions (class B)). Color

Other functions in ‘`influence.c`’ are of the nature of utilities which may be useful throughout the engine. We list the most useful ones here.

- `void influence_mark_non_territory(int pos, int color)`

  Called from actions for ‘`t`’ patterns in ‘`barriers.db`’. Marks `pos` as not being territory for `color`.

- `int whose_territory(const struct influence_data *q, int pos)`

  Return the color of the territory at `pos`. If it's territory for neither color, `EMPTY` is returned.

- `int whose_moyo(const struct influence_data *q, int pos)`

  Return the color who has a moyo at `pos`. If neither color has a moyo there, `EMPTY` is returned. The definition of moyo in terms of the influences is totally ad hoc.

- `int whose_area(const struct influence_data *q, int pos)`

  Return the color who has dominating influence (“area”) at `pos`. If neither color dominates the influence there, EMPTY is returned. The definition of area in terms of the influences is totally ad hoc.

[<<] [>>]   [Top] [Contents] [Index] [?]

19. The Go Text Protocol

19.1 The Go Text Protocol

GNU Go 3.0 introduced a new interface, the Go Text Protocol, abbreviated GTP. The intention was 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.

There are two versions of the protocol. Version 1 was used with GNU Go 3.0 and 3.2. GNU Go 3.4 and later versions use protocol version 2. The specification of GTP version 2 is available at http://www.lysator.liu.se/~gunnar/gtp/. GNU Go 3.4 is the reference implementation for GTP version 2, but all but the most common commands are to be regarded as private extensions of the protocol.

The GTP has a variety of applications. For GNU Go the first use was in regression testing (see section Regression testing), followed by communication with the NNGS go server and for automated test games against itself and other programs. Now there are also many graphical user interfaces available supporting GTP, as well as bridges to other Go servers than NNGS.

19.2 Running GNU Go in GTP mode

To start GNU Go in GTP mode, simply invoke it with the option '`--mode gtp`'. You will not get a prompt or any other output to start with but GNU Go is silently waiting for GTP commands.

A sample GTP session may look as follows:

```
virihaure 462% ./gnugo --mode gtp
1 boardsize 7
=1

2 clear_board
=2

3 play black D5
=3

4 genmove white
=4 C3

5 play black C3
?5 illegal move

6 play black E3
=6

7 showboard
=7
   A B C D E F G
 7 . . . . . . . 7
 6 . . . . . . . 6
 5 . . + X + . . 5
 4 . . . + . . . 4
 3 . . O . X . . 3
 2 . . . . . . . 2     WHITE (O) has captured 0 stones
 1 . . . . . . . 1     BLACK (X) has captured 0 stones
   A B C D E F G

8 quit
=8
```

Commands are given on a single line, starting by an optional identity number, followed by the command name and its arguments.

If the command is successful, the response starts by an equals sign ('`=`'), followed by the identity number of the command (if any) and then the result. In this example all results were empty strings except for command 4 where the answer was the white move at C3, and command 7 where the result was a diagram of the current board position. The response ends by two consecutive newlines.

Failing commands are signified by a question mark ('`?`') instead of an equals sign, as in the response to command 5.

The detailed specification of the protocol can be found at http://www.lysator.liu.se/~gunnar/gtp/. The available commands in GNU Go may always be listed using the command `list_commands`. They are also documented in See section GTP command reference.

19.3 GTP applications

GTP is an asymmetric protocol involving two parties which we call controller and engine. The controller sends all commands and the engine only responds to these commands. GNU Go implements the engine end of the protocol.

With the source code of GNU Go is also distributed a number of applications implementing the controller end. Among the most interesting of these are:

'`regression/regress.awk`'

Script to run regressions. The script sends GTP commands to set up and evaluate positions to the engine and then analyzes the responses from the engine. More information about GTP based regression testing can be found in the regression chapter (see section Regression testing).

'`regression/regress.pl`'

Perl script to run regressions, giving output which together with the CGI script '`regression/regress.plx`' generates HTML views of the regressions.

'`regression/regress.pike`'

Pike script to run regressions. More feature-rich and powerful than '`regress.awk`'.

'`regression/view.pike`'

Pike script to examine a single regression testcase through a graphical board. This gives an easy way to inspect many of the GNU Go internals.

'`interface/gtp_examples/twogtp`'

Perl script to play two engines against each other. The script essentially sets up both engines with desired boardsize, handicap, and komi, then relays moves back and forth between the engines.

'`interface/gtp_examples/twogtp-a`'

An alternative Perl implementation of twogtp.

'`interface/gtp_examples/twogtp.py`'

Implementation of twogtp in Python. Has more features than the Perl variants.

'`interface/gtp_examples/twogtp.pike`'

Implementation of twogtp in Pike. Has even more features than the Python variant.

'`interface/gtp_examples/2ptkgo.pl`'

Variation of twogtp which includes a graphical board.

More GTP applications, including bridges to go servers and graphical user interfaces, are listed at http://www.lysator.liu.se/~gunnar/gtp/.

19.4 The Metamachine

An interesting application of the GTP is the concept of using GNU Go as an "Oracle" that can be consulted by another process. This could be another computer program that asks GNU Go to generate future board positions, then evaluate them.

David Doshay at the University of California at Santa Cruz has done interesting experiments with a parallel engine, known as SlugGo, that is based on GNU Go. These are described in http://lists.gnu.org/archive/html/gnugo-devel/2004-08/msg00060.html.

The "Metamachine" experiment is a more modest approach using the GTP to communicate with a GNU Go process that is used as an oracle. The following scheme is used.

- The GNU Go "oracle" is asked to generate its top moves using the GTP `top_moves` commands.
- Both moves are tried and `estimate_score` is called from the resulting board position.
- The higher scoring position is selected as the engine's move.

This scheme does not produce a stronger engine, but it is suggestive, and the SlugGo experiment seems to show that a more elaborate scheme along the same lines could produce a stronger engine.

Two implementations are distributed with GNU Go. Both make use of `fork` and `pipe` system calls, so they require a Unix-like environment. The Metamachine has been tested under GNU/Linux.

Important: If the Metamachine terminates normally, the GNU Go process will be killed. However there is a danger that something will go wrong. When you are finished running the Metamachine, it is a good idea to run `ps -A|grep gnugo` or `ps -aux|grep gnugo` to make sure there are no unterminated processes. (If there are, just kill them.)

19.4.1 The Standalone Metamachine

In '`interface/gtp_examples/metamachine.c`' is a standalone implementation of the Metamachine. Compile it with `cc -o metamachine metamachine.c` and run it. It forks a `gnugo` process with which it communicates through the GTP, to use as an oracle.

The following scheme is followed:

```
           stdin             pipe a
GTP client ----> Metamachine -----> GNU Go
           <----             <-----
          stdout             pipe b
```

Most commands issued by the client are passed along verbatim to GNU Go by the Metamachine. The exception is gg_genmove, which is intercepted then processed differently, as described above. The client is unaware of this, and only knows that it issued a gg_genmove command and received a reply. Thus to the the Metamachine appears as an ordinary GTP engine.

Usage: no arguments gives normal GTP behavior. `metamachine --debug` sends diagnostics to stderr.

19.4.2 GNU Go as a Metamachine

Alternatively, you may compile GNU Go with the configure option '`--enable-metamachine`'. This causes the file `oracle.c` to be compiled, which contains the Metamachine code. This has no effect on the engine unless you run GNU Go with the runtime option '`--metamachine`'. Thus you must use both the configure and the runtime option to get the Metamachine.

This method is better than the standalone program since you have access to GNU Go's facilities. For example, you can run the Metamachine with CGoban or in Ascii mode this way.

You can get traces by adding the command line '`-d0x1000000`'. In debugging the Metamachine, a danger is that any small oversight in designing the program can cause the forked process and the controller to hang, each one waiting for a response from the other. If this seems to happen it is useful to know that you can attach `gdb` to a running process and find out what it is doing.

19.5 Adding new GTP commands

The implementation of GTP in GNU Go is distributed over three files, '`interface/gtp.h`', '`interface/gtp.c`', and '`interface/play_gtp.c`'. The first two implement a small library of helper functions which can be used also by other programs. In the interest of promoting the GTP they are licensed with minimal restrictions (see section The Go Text Protocol License). The actual GTP commands are implemented in '`play_gtp.c`', which has knowledge about the engine internals.

To see how a simple but fairly typical command is implemented we look at `gtp_countlib()` (a GNU Go private extension command):

```
static int
gtp_countlib(char *s)
{
  int i, j;
  if (!gtp_decode_coord(s, &i, &j))
    return gtp_failure("invalid coordinate");

  if (BOARD(i, j) == EMPTY)
    return gtp_failure("vertex must not be empty");

  return gtp_success("%d", countlib(POS(i, j)));
}
```

The arguments to the command are passed in the string `s`. In this case we expect a vertex as argument and thus try to read it with `gtp_decode_coord()` from '`gtp.c`'.

A correctly formatted response should start with either '`=`' or '`?`', followed by the identity number (if one was sent), the actual result, and finally two consecutive newlines. It is important to get this formatting correct since the controller in the other end relies on it. Naturally the result itself cannot contain two consecutive newlines but it may be split over several lines by single newlines.

The easiest way to generate a correctly formatted response is with one of the functions `gtp_failure()` and `gtp_success()`, assuming that their formatted output does 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() does not support. Then we have to fall back to the construction in e.g. `gtp_genmove()`:

```
static int
gtp_genmove(char *s)
{
  [...]
  gtp_start_response(GTP_SUCCESS);
  gtp_print_vertex(i, j);
  return gtp_finish_response();
}
```

Here `gtp_start_response()` writes the equal sign and the identity number while `gtp_finish_response()` adds the final two newlines. The next example is from `gtp_list_commands()`:

```
static int
gtp_list_commands(char *s)
{
  int k;
  UNUSED(s);

  gtp_start_response(GTP_SUCCESS);

  for (k = 0; commands[k].name != NULL; k++)
    gtp_printf("%s\n", commands[k].name);

  gtp_printf("\n");
  return GTP_OK;
}
```

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 from the last command printed in the loop.

In order to add a new GTP command to GNU Go, the following pieces of code need to be inserted in '`play_gtp.c`':

1. A function declaration using the `DECLARE` macro in the list starting at line 68.
2. An entry in the `commands[]` array starting at line 200.
3. An implementation of the function handling the command.

Useful helper functions in '`gtp.c`'/'`gtp.h`' are:

`gtp_printf()` for basic formatted printing.
`gtp_mprintf()` for printing with special format codes for vertices and colors.
`gtp_success()` and `gtp_failure()` for simple responses.

`gtp_start_response()` and `gtp_end_response()` for more complex responses.
`gtp_print_vertex()` and `gtp_print_vertices()` for printing one or multiple vertices.
`gtp_decode_color()` to read in a color from the command arguments.
`gtp_decode_coord()` to read in a vertex from the command arguments.
`gtp_decode_move()` to read in a move, i.e. color plus vertex, from the command arguments.

19.6 GTP command reference

This section lists the GTP commands implemented in GNU Go along with some information about each command. Each entry in the list has the following fields:

- Function: What this command does.
- Arguments: What other information, if any, this command requires. Typical values include none or vertex or integer (there are others).
- Fails: Circumstances which cause this command to fail.
- Returns: What is displayed after the = and before the two newlines. Typical values include nothing or a move coordinate or some status string (there are others).
- Status: How this command relates to the standard GTP version 2 commands. If nothing else is specified it is a GNU Go private extension.

Without further ado, here is the big list (in no particular order).

Note: if new commands are added by editing 'interface/play_gtp.c' this list could become incomplete. You may rebuild this list in 'doc/gtp-commands.texi' with the command `make gtp-commands` in the 'doc/' directory. This may require GNU sed.

-
- quit: Quit

```
Arguments: none
Fails:     never
Returns:   nothing

Status:    GTP version 2 standard command.
```

- protocol_version: Report protocol version.

```
Arguments: none
Fails:     never
Returns:   protocol version number

Status:    GTP version 2 standard command.
```

- name: Report the name of the program.

```
Arguments: none
Fails:     never
Returns:   program name

Status:    GTP version 2 standard command.
```

- version: Report the version number of the program.

```
Arguments: none
Fails:     never
Returns:   version number

Status:    GTP version 2 standard command.
```

- boardsize: Set the board size to NxN and clear the board.

```
Arguments: integer
Fails:     board size outside engine's limits
Returns:   nothing

Status:    GTP version 2 standard command.
```

- query_boardsize: Find the current boardsize

```
Arguments: none
Fails:     never
Returns:   board_size
```

- clear_board: Clear the board.

```
Arguments: none
Fails:     never
Returns:   nothing

Status:    GTP version 2 standard command.
```

- orientation: Set the orienation to N and clear the board

```
Arguments: integer
Fails:     illegal orientation
Returns:   nothing
```

- query_orientation: Find the current orientation

```
Arguments: none
Fails:     never
Returns:   orientation
```

- komi: Set the komi.

```
Arguments: float
Fails:     incorrect argument
Returns:   nothing

Status:    GTP version 2 standard command.
```

- get_komi: Get the komi

```
Arguments: none
Fails:     never
Returns:   Komi
```

- black: Play a black stone at the given vertex.

```
Arguments: vertex
Fails:     invalid vertex, illegal move
Returns:   nothing

Status:    Obsolete GTP version 1 command.
```

- playwhite: Play a white stone at the given vertex.

```
Arguments: vertex
Fails:     invalid vertex, illegal move
Returns:   nothing

Status:    Obsolete GTP version 1 command.
```

- play: Play a stone of the given color at the given vertex.

```
Arguments: color, vertex
Fails:     invalid vertex, illegal move
Returns:   nothing

Status:    GTP version 2 standard command.
```

- fixed_handicap: Set up fixed placement handicap stones.

```
Arguments: number of handicap stones
Fails:     invalid number of stones for the current boardsize
Returns:   list of vertices with handicap stones

Status:    GTP version 2 standard command.
```

- place_free_handicap: Choose free placement handicap stones and put them on the board.

```
Arguments: number of handicap stones
Fails:     invalid number of stones
Returns:   list of vertices with handicap stones

Status:    GTP version 2 standard command.
```

- set_free_handicap: Put free placement handicap stones on the board.

```
Arguments: list of vertices with handicap stones
Fails:     board not empty, bad list of vertices
Returns:   nothing

Status:    GTP version 2 standard command.
```

- get_handicap: Get the handicap

```
Arguments: none
Fails:     never
Returns:   handicap
```

- loadsgf: Load an sgf file, possibly up to a move number or the first occurence of a move.

```
Arguments: filename + move number, vertex, or nothing
Fails:     missing filename or failure to open or parse file
Returns:   color to play

Status:    GTP version 2 standard command.
```

- color: Return the color at a vertex.

```
Arguments: vertex
Fails:     invalid vertex
Returns:   "black", "white", or "empty"
```

- list_stones: List vertices with either black or white stones.

```
Arguments: color
Fails:     invalid color
Returns:   list of vertices
```

- countlib: Count number of liberties for the string at a vertex.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   Number of liberties.
```

- findlib: Return the positions of the liberties for the string at a vertex.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   Sorted space separated list of vertices.
```

- accuratelib: Determine which liberties a stone of given color will get if played at given vertex.

```
Arguments: move (color + vertex)
Fails:     invalid color, invalid vertex, occupied vertex
Returns:   Sorted space separated list of liberties
```

- accurate_approxlib: Determine which liberties a stone of given color will get if played at given vertex.

```
Arguments: move (color + vertex)
Fails:     invalid color, invalid vertex, occupied vertex
Returns:   Sorted space separated list of liberties

Supposedly identical in behavior to the above function and
can be retired when this is confirmed.
```

- is_legal: Tell whether a move is legal.

```
Arguments: move
Fails:     invalid move
Returns:   1 if the move is legal, 0 if it is not.
```

- all_legal: List all legal moves for either color.

```
Arguments: color
Fails:     invalid color
Returns:   Sorted space separated list of vertices.
```

- captures: List the number of captures taken by either color.

```
Arguments: color
Fails:     invalid color
Returns:   Number of captures.
```

- last_move: Return the last move.

```
Arguments: none
Fails:     no previous move known
Returns:   Color and vertex of last move.
```

- move_history: Print the move history in reverse order

```
Arguments: none
Fails:     never
Returns:   List of moves played in reverse order in format:
           color move (one move per line)
```

- invariant_hash: Return the rotation/reflection invariant board hash.

```
Arguments: none
Fails:     never
Returns:   Invariant hash for the board as a hexadecimal number.
```

- invariant_hash_for_moves: Return the rotation/reflection invariant board hash obtained by playing all the possible moves for the given color.

```
Arguments: color
Fails:     invalid color
Returns:   List of moves + invariant hash as a hexadecimal number,
           one pair of move + hash per line.
```

- trymove: Play a stone of the given color at the given vertex.

```
Arguments: move (color + vertex)
Fails:     invalid color, invalid vertex, illegal move
Returns:   nothing
```

- tryko: Play a stone of the given color at the given vertex, allowing illegal ko capture.

```
Arguments: move (color + vertex)
Fails:     invalid color, invalid vertex, illegal move
Returns:   nothing
```

- popgo: Undo a trymove or tryko.

```
Arguments: none
Fails:     stack empty
Returns:   nothing
```

- clear_cache: clear the caches.

```
Arguments: none.
Fails:     never.
Returns:   nothing.
```

- attack: Try to attack a string.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   attack code followed by attack point if attack code nonzero.
```

- attack_either: Try to attack either of two strings

```
Arguments: two vertices
Fails:     invalid vertex, empty vertex
Returns:   attack code against the strings.  Guarantees there
           exists a move which will attack one of the two
           with attack_code, but does not return the move.
```

- defend: Try to defend a string.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   defense code followed by defense point if defense code nonzero.
```

- does_attack: Examine whether a specific move attacks a string tactically.

```
Arguments: vertex (move), vertex (dragon)
Fails:     invalid vertex, empty vertex
Returns:   attack code
```

- does_defend: Examine whether a specific move defends a string tactically.

```
Arguments: vertex (move), vertex (dragon)
Fails:     invalid vertex, empty vertex
Returns:   attack code
```

- ladder_attack: Try to attack a string strictly in a ladder.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   attack code followed by attack point if attack code nonzero.
```

- increase_depths: Increase depth values by one.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- decrease_depths: Decrease depth values by one.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- owl_attack: Try to attack a dragon.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   attack code followed by attack point if attack code nonzero.
```

- owl_defend: Try to defend a dragon.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   defense code followed by defense point if defense code nonzero.
```

- owl_threaten_attack: Try to attack a dragon in 2 moves.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   attack code followed by the two attack points if
           attack code nonzero.
```

- owl_threaten_defense: Try to defend a dragon with 2 moves.

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   defense code followed by the 2 defense points if
           defense code nonzero.
```

- owl_does_attack: Examine whether a specific move attacks a dragon.

```
Arguments: vertex (move), vertex (dragon)
Fails:     invalid vertex, empty vertex
Returns:   attack code
```

- owl_does_defend: Examine whether a specific move defends a dragon.

```
Arguments: vertex (move), vertex (dragon)
Fails:     invalid vertex, empty vertex
Returns:   defense code
```

- owl_connection_defends: Examine whether a connection defends involved dragons.

```
Arguments: vertex (move), vertex (dragon1), vertex (dragon2)
Fails:     invalid vertex, empty vertex
Returns:   defense code
```

- defend_both: Try to defend both of two strings

```
Arguments: two vertices
Fails:     invalid vertex, empty vertex
Returns:   defend code for the strings.  Guarantees there
           exists a move which will defend both of the two
           with defend_code, but does not return the move.
```

- owl_substantial: Determine whether capturing a string gives a living dragon

```
Arguments: vertex
Fails:     invalid vertex, empty vertex
Returns:   1 if dragon can live, 0 otherwise
```

- analyze_semeai: Analyze a semeai

```
Arguments: dragona, dragonb
Fails:     invalid vertices, empty vertices
Returns:   semeai defense result, semeai attack result, semeai move
```

- analyze_semeai_after_move: Analyze a semeai after a move have been made.

```
Arguments: color, vertex, dragona, dragonb
Fails:     invalid vertices
Returns:   semeai defense result, semeai attack result, semeai move
```

- tactical_analyze_semeai: Analyze a semeai, not using owl

```
Arguments: dragona, dragonb
Fails:     invalid vertices, empty vertices
Returns:   status of dragona, dragonb assuming dragona moves first
```

- connect: Try to connect two strings.

```
Arguments: vertex, vertex
Fails:     invalid vertex, empty vertex, vertices of different colors
Returns:   connect result followed by connect point if successful.
```

- disconnect: Try to disconnect two strings.

```
Arguments: vertex, vertex
Fails:     invalid vertex, empty vertex, vertices of different colors
Returns:   disconnect result followed by disconnect point if successful.
```

- break_in: Try to break from string into area.

```
Arguments: vertex, vertices
Fails:     invalid vertex, empty vertex.
Returns:   result followed by break in point if successful.
```

- block_off: Try to block string from area.

```
Arguments: vertex, vertices
Fails:     invalid vertex, empty vertex.
Returns:   result followed by block point if successful.
```

- eval_eye: Evaluate an eye space

```
Arguments: vertex
Fails:     invalid vertex
Returns:   Minimum and maximum number of eyes. If these differ an
           attack and a defense point are additionally returned.
           If the vertex is not an eye space or not of unique color,
           a single -1 is returned.
```

- dragon_status: Determine status of a dragon.

```
Arguments: optional vertex
Fails:     invalid vertex, empty vertex
Returns:   status ("alive", "critical", "dead", or "unknown"),
           attack point, defense point. Points of attack and
           defense are only given if the status is critical.
           If no vertex is given, the status is listed for all
           dragons, one per row in the format "A4: alive".

FIXME: Should be able to distinguish between life in seki
       and independent life. Should also be able to identify ko.
```

- same_dragon: Determine whether two stones belong to the same dragon.

```
Arguments: vertex, vertex
Fails:     invalid vertex, empty vertex
Returns:   1 if the vertices belong to the same dragon, 0 otherwise
```

- unconditional_status: Determine the unconditional status of a vertex.

```
Arguments: vertex
Fails:     invalid vertex
Returns:   unconditional status ("undecided", "alive", "dead",
           "white_territory", "black_territory"). Occupied vertices can
           be undecided, alive, or dead. Empty vertices can be
           undecided, white territory, or black territory.
```

- combination_attack: Find a move by color capturing something through a combination attack.

```
Arguments: color
Fails:     invalid color
Returns:   Recommended move, PASS if no move found
```

- combination_defend: If color can capture something through a combination attack, list moves by the opponent of color to defend against this attack.

```
Arguments: color
Fails:     invalid color
Returns:   Recommended moves, PASS if no combination attack found.
```

- aa_confirm_safety: Run atari_atari_confirm_safety().

```
Arguments: move, optional int
Fails:     invalid move
Returns:   success code, if failure also defending move
```

- genmove_black: Generate and play the supposedly best black move.

```
Arguments: none
Fails:     never
Returns:   a move coordinate or "PASS"

Status:    Obsolete GTP version 1 command.
```

- genmove_white: Generate and play the supposedly best white move.

```
Arguments: none
Fails:     never
Returns:   a move coordinate or "PASS"

Status:    Obsolete GTP version 1 command.
```

- genmove: Generate and play the supposedly best move for either color.

```
Arguments: color to move
Fails:     invalid color
Returns:   a move coordinate or "PASS" (or "resign" if resignation_allowed)

Status:    GTP version 2 standard command.
```

- reg_genmove: Generate the supposedly best move for either color.

```
Arguments: color to move
Fails:     invalid color
Returns:   a move coordinate (or "PASS")

Status:    GTP version 2 standard command.
```

- gg_genmove: Generate the supposedly best move for either color.

```
Arguments: color to move, optionally a random seed
Fails:     invalid color
Returns:   a move coordinate (or "PASS")

This differs from reg_genmove in the optional random seed.
```

- restricted_genmove: Generate the supposedly best move for either color from a choice of allowed vertices.

```
Arguments: color to move, allowed vertices
Fails:     invalid color, invalid vertex, no vertex listed
Returns:   a move coordinate (or "PASS")
```

- kgs-genmove_cleanup: Generate and play the supposedly best move for either color, not passing until all dead opponent stones have been removed.

```
Arguments: color to move
Fails:     invalid color
Returns:   a move coordinate (or "PASS")

Status:    KGS specific command.

A similar command, but possibly somewhat different, will likely be added
to GTP version 3 at a later time.
```

- level: Set the playing level.

```
Arguments: int
Fails:     incorrect argument
Returns:   nothing
```

- undo: Undo one move

```
Arguments: none
Fails:     If move history is too short.
Returns:   nothing

Status:    GTP version 2 standard command.
```

- gg-undo: Undo a number of moves

```
Arguments: optional int
Fails:     If move history is too short.
Returns:   nothing
```

- time_settings: Set time allowance

```
Arguments: int main_time, int byo_yomi_time, int byo_yomi_stones
Fails:     syntax error
Returns:   nothing

Status:    GTP version 2 standard command.
```

- time_left: Report remaining time

```
Arguments: color color, int time, int stones
Fails:     syntax error
Returns:   nothing

Status:    GTP version 2 standard command.
```

- final_score: Compute the score of a finished game.

```
Arguments: Optional random seed
Fails:     never
Returns:   Score in SGF format (RE property).

Status:    GTP version 2 standard command.
```

- final_status: Report the final status of a vertex in a finished game.

```
Arguments: Vertex, optional random seed
Fails:     invalid vertex
```

```
Returns:   Status in the form of one of the strings "alive", "dead",
           "seki", "white_territory", "black_territory", or "dame".
```

- final_status_list: Report vertices with a specific final status in a finished game.

```
Arguments: Status in the form of one of the strings "alive", "dead",
           "seki", "white_territory", "black_territory", or "dame".
           An optional random seed can be added.
Fails:     missing or invalid status string
Returns:   Vertices having the specified status. These are split with
           one string on each line if the vertices are nonempty (i.e.
           for "alive", "dead", and "seki").

Status:    GTP version 2 standard command.
           However, "dame", "white_territory", and "black_territory"
           are private extensions.
```

- estimate_score: Estimate the score

```
Arguments: None
Fails:     never
Returns:   upper and lower bounds for the score
```

- experimental_score: Estimate the score, taking into account which player moves next

```
Arguments: Color to play
Fails:     Invalid color
Returns:   Score.

This function generates a move for color, then adds the
value of the move generated to the value of the position.
Critical dragons are awarded to the opponent since the
value of rescuing a critical dragon is taken into account
in the value of the move generated.
```

- reset_life_node_counter: Reset the count of life nodes.

```
Arguments: none
Fails:     never
Returns:   nothing

Note: This function is obsolete and only remains for backwards
compatibility.
```

- get_life_node_counter: Retrieve the count of life nodes.

```
Arguments: none
Fails:     never
Returns:   number of life nodes

Note: This function is obsolete and only remains for backwards
compatibility.
```

- reset_owl_node_counter: Reset the count of owl nodes.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- get_owl_node_counter: Retrieve the count of owl nodes.

```
Arguments: none
Fails:     never
Returns:   number of owl nodes
```

- reset_reading_node_counter: Reset the count of reading nodes.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- get_reading_node_counter: Retrieve the count of reading nodes.

```
Arguments: none
Fails:     never
Returns:   number of reading nodes
```

- reset_trymove_counter: Reset the count of trymoves/trykos.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- get_trymove_counter: Retrieve the count of trymoves/trykos.

```
Arguments: none
Fails:     never
Returns:   number of trymoves/trykos
```

- reset_connection_node_counter: Reset the count of connection nodes.

```
Arguments: none
Fails:     never
Returns:   nothing
```

- get_connection_node_counter: Retrieve the count of connection nodes.

```
Arguments: none
Fails:     never
```

```
Returns:   number of connection nodes
```

* test_eyeshape: Test an eyeshape for inconsistent evaluations

```
Arguments: Eyeshape vertices
Fails:     Bad vertices
Returns:   Failure reports on stderr.
```

* analyze_eyegraph: Compute an eyevalue and vital points for an eye graph

```
Arguments: Eyeshape encoded in string
Fails:     Bad eyeshape, analysis failed
Returns:   Eyevalue, vital points
```

* cputime: Returns elapsed CPU time in seconds.

```
Arguments: none
Fails:     never
Returns:   Total elapsed (user + system) CPU time in seconds.
```

* showboard: Write the position to stdout.

```
Arguments: none
Fails:     never
Returns:   nothing

Status:    GTP version 2 standard command.
```

* dump_stack: Dump stack to stderr.

```
Arguments: none
Fails:     never
Returns:   nothing
```

* initial_influence: Return information about the initial influence function.

```
Arguments: color to move, what information
Fails:     never
Returns:   Influence data formatted like:

    0.51   1.34   3.20   6.60   9.09   8.06   1.96   0.00   0.00
    0.45   1.65   4.92  12.19  17.47  15.92   4.03   0.00   0.00
                  .
                  .
                  .
    0.00   0.00   0.00   0.00   0.00 100.00  75.53  41.47  23.41

The available choices of information are:

white_influence (float)
black_influence (float)
white_strength (float)
black_strength (float)
white_attenuation (float)
black_attenuation (float)
white_permeability (float)
black_permeability (float)
territory_value (float)
influence_regions (int)
non_territory (int)

The encoding of influence_regions is as follows:
 4 white stone
 3 white territory
 2 white moyo
 1 white area
 0 neutral
-1 black area
-2 black moyo
-3 black territory
-4 black stone
```

* move_influence: Return information about the influence function after a move.

```
Arguments: move, what information
Fails:     never
Returns:   Influence data formatted like for initial_influence.
```

* move_probabilities: List probabilities of each move being played (when non-zero). If no previous genmove command has been issued, the result of this command will be meaningless.

```
Arguments: none
Fails:     never
Returns:   Move, probabilty pairs, one per row.
```

* move_uncertainty: Return the number of bits of uncertainty in the move. If no previous genmove command has been issued, the result of this command will be meaningless.

```
Arguments: none
Fails:     never
Returns:   bits of uncertainty
```

* followup_influence: Return information about the followup influence after a move.

```
Arguments: move, what information
Fails:     never
Returns:   Influence data formatted like for initial_influence.
```

* worm_data: Return the information in the worm data structure.

```
Arguments: optional vertex
Fails:     never
Returns:   Worm data formatted like:

A19:
color           black
size            10
effective_size  17.83
origin          A19
liberties       8
liberties2      15
liberties3      10
liberties4      8
attack          PASS
attack_code     0
lunch           B19
defend          PASS
defend_code     0
cutstone        2
cutstone2       0
genus           0
inessential     0
B19:
color           white
.
.
.
inessential     0
C19:
...

If an intersection is specified, only data for this one will be returned.
```

- worm_stones: List the stones of a worm

```
Arguments: the location, "BLACK" or "WHITE"
Fails:     if called on an empty or off-board location
Returns:   list of stones
```

- worm_cutstone: Return the cutstone field in the worm data structure.

```
Arguments: non-empty vertex
Fails:     never
Returns:   cutstone
```

- dragon_data: Return the information in the dragon data structure.

```
Arguments: optional intersection
Fails:     never
Returns:   Dragon data formatted in the corresponding way to worm_data.
```

- dragon_stones: List the stones of a dragon

```
Arguments: the location
Fails:     if called on an empty or off-board location
Returns:   list of stones
```

- eye_data: Return the information in the eye data structure.

```
Arguments: color, vertex
Fails:     never
Returns:   eye data fields and values, one pair per row
```

- half_eye_data: Return the information in the half eye data structure.

```
Arguments: vertex
Fails:     never
Returns:   half eye data fields and values, one pair per row
```

- start_sgftrace: Start storing moves executed during reading in an sgf tree in memory.

```
Arguments: none
Fails:     never
Returns:   nothing

Warning: You had better know what you're doing if you try to use this
         command.
```

- finish_sgftrace: Finish storing moves in an sgf tree and write it to file.

```
Arguments: filename
Fails:     never
Returns:   nothing

Warning: You had better know what you're doing if you try to use this
         command.
```

- printsgf: Dump the current position as a static sgf file to filename, or as output if filename is missing or "-"

```
Arguments: optional filename
Fails:     never
Returns:   nothing if filename, otherwise the sgf
```

- tune_move_ordering: Tune the parameters for the move ordering in the tactical reading.

```
Arguments: MOVE_ORDERING_PARAMETERS integers
Fails:     incorrect arguments
Returns:   nothing
```

- echo: Echo the parameter

```
Arguments: string
Fails:     never
Returns:   nothing
```

- echo_err: Echo the parameter to stdout AND stderr

```
Arguments: string
Fails:     never
Returns:   nothing
```

- help: List all known commands

```
Arguments: none
Fails:     never
Returns:   list of known commands, one per line

Status:    GTP version 2 standard command.
```

- known_command: Tell whether a command is known.

```
Arguments: command name
Fails:     never
Returns:   "true" if command exists, "false" if not

Status:    GTP version 2 standard command.
```

- report_uncertainty: Turn uncertainty reports from owl_attack and owl_defend on or off.

```
Arguments: "on" or "off"
Fails:     invalid argument
Returns:   nothing
```

- get_random_seed: Get the random seed

```
Arguments: none
Fails:     never
Returns:   random seed
```

- set_random_seed: Set the random seed

```
Arguments: integer
Fails:     invalid data
Returns:   nothing
```

- advance_random_seed: Advance the random seed by a number of games.

```
Arguments: integer
Fails:     invalid data
Returns:   New random seed.
```

- is_surrounded: Determine if a dragon is surrounded

```
Arguments: vertex (dragon)
Fails:     invalid vertex, empty vertex
Returns:   1 if surrounded, 2 if weakly surrounded, 0 if not
```

- does_surround: Determine if a move surrounds a dragon

```
Arguments: vertex (move), vertex (dragon)
Fails:     invalid vertex, empty (dragon, nonempty (move)
Returns:   1 if (move) surrounds (dragon)
```

- surround_map: Report the surround map for dragon at a vertex

```
Arguments: vertex (dragon), vertex (mapped location)
Fails:     invalid vertex, empty dragon
Returns:   value of surround map at (mapped location), or -1 if
           dragon not surrounded.
```

- set_search_diamond: limit search, and establish a search diamond

```
Arguments: pos
Fails:     invalid value
Returns:   nothing
```

- reset_search_mask: unmark the entire board for limited search

```
Arguments: none
Fails:     never
Returns:   nothing
```

- limit_search: sets the global variable limit_search

```
Arguments: value
Fails:     invalid arguments
Returns:   nothing
```

- set_search_limit: mark a vertex for limited search

```
Arguments: position
Fails:     invalid arguments
Returns:   nothing
```

- draw_search_area: Draw search area. Writes to stderr.

```
Arguments: none
Fails:     never
Returns:   nothing
```

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.

20. 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.

20.1 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. The Pike script ‘`regress.pike`’ can also be used to run all tests or a subset of the tests.

Game records used by the regression tests are stored in the directory ‘`regression/games/`’ and its subdirectories.

20.2 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.

20.3 Running the Regression 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 all_batches` runs all test suites in a sequence using the `regress.sh` script.

20.4 Running regress.pike

A more powerful way to run regressions is with the script '`regress.pike`'. This requires that you have Pike (http://pike.ida.liu.se) installed.

Executing `./regress.pike` without arguments will run all testsuites that `make all_batches` would run. The difference is that unexpected results are reported immediately when they have been found (instead of after the whole file has been run) and that statistics of time consumption and node usage is presented for each test file and in total.

To run a single test suite do e.g. `./regress.pike nicklas3.tst` or `./regress.pike nicklas3`. The result may look like:

```
nicklas3                                2.96    614772    3322     469
Total nodes: 614772 3322 469
Total time: 2.96 (3.22)
Total uncertainty: 0.00
```

The numbers here mean that the test suite took 2.96 seconds of processor time and 3.22 seconds of real time. The consumption of reading nodes was 614772 for tactical reading, 3322 for owl reading, and 469 for connection reading. The last line relates to the variability of the generated moves in the test suite, and 0 means that none was decided by the randomness contribution to the move valuation. Multiple testsuites can be run by e.g. `./regress.pike owl ld_owl owl1`.

It is also possible to run a single testcase, e.g. `./regress.pike strategy:6`, a number of testcases, e.g. `./regress.pike strategy:6,23,45`, a range of testcases, e.g. `./regress.pike strategy:13-15` or more complex combinations e.g. `./regress.pike strategy:6,13-15,23,45 nicklas3:602,1403`.

There are also command line options to choose what engine to run, what options to send to the engine, to turn on verbose output, and to use a file to specify which testcases to run. Run `./regress.pike --help` for a complete and up to date list of options.

20.5 Viewing tests with Emacs

To get a quick regression view, you may use the graphical display mode available with Emacs (see section GNU Go mode in Emacs). You will want the cursor in the regression buffer when you enter `M-x gnugo`, so that GNU Go opens in the correct directory. A good way to be in the right directory is to open the window of the test you want to investigate. Then you can cut and past GTP commands directly from the test to the minibuffer, using the `:` command from Emacs. Although Emacs mode does not have a coordinate grid, you may get an ascii board with the coordinate grid using `: showboard` command.

20.6 HTML Regression Views

Extremely useful HTML Views of the regression tests may be produced using two perl scripts '`regression/regress.pl`' and '`regression/regress.plx`'.

1. The driver program (regress.pl) which:
   - Runs the regression tests, invoking GNU Go.

- Captures the trace output, board position, and pass/fail status, sgf output, and dragon status information.

2. The interface to view the captured output (regress.plx) which:
    - Never invokes GNU Go.
    - Displays the captured output in helpful formats (i.e. HTML).

20.6.1 Setting up the HTML regression Views

There are many ways configuring Apache to permit CGI scripts, all of them are featured in Apache documentation, which can be found at http://httpd.apache.org/docs/2.0/howto/cgi.html

Below you will find one example.

This documentation assumes an Apache 2.0 included in Fedora Core distribution, but it should be fairly close to the config for other distributions.

First, you will need to configure Apache to run CGI scripts in the directory you wish to serve the html views from. In '`/etc/httpd/conf/httpd.conf`' there should be a line:

```
DocumentRoot "/var/www/html"
```

Search for a line `<Directory "/path/to/directory">`, where `/path/to/directory` is the same as provided in `DocumentRoot`, then add `ExecCGI` to list of `Options`. The whole section should look like:

```
<Directory "/var/www/html">
...
    Options ... ExecCGI
...
</Directory>
```

This allows CGI scripts to be executed in the directory used by regress.plx. Next, you need to tell Apache that '`.plx`' is a CGI script ending. Your '`httpd.conf`' file should contain a line:

```
AddHandler cgi-script ...
```

If there isn't already, add it; add '`.plx`' to the list of extensions, so line should look like:

```
AddHandler cgi-script ... .plx
```

You will also need to make sure you have the necessary modules loaded to run CGI scripts; mod_cgi and mod_mime should be sufficient. Your '`httpd.conf`' should have the relevant `LoadModule cgi_module modules/mod_cgi.so` and `LoadModule mime_module modules/mod_mime.so` lines; uncomment them if necessary.

Next, you need to put a copy of '`regress.plx`' in the `DocumentRoot` directory `/var/www/html` or it subdirectories where you plan to serve the html views from.

You will also need to install the Perl module GD (http://search.cpan.org/dist/GD/), available from CPAN.

Finally, run '`regression/regress.pl`' to create the xml data used to generate the html views (to do all regression tests run '`regression/regress.pl -a 1`'); then, copy the '`html/`' directory to the same directory as '`regress.plx`' resides in.

At this point, you should have a working copy of the html regression views.

Additional notes for Debian users: The Perl GD module can be installed by `apt-get install libgd-perl`. It may suffice to add this to the apache2 configuration:

```
<Directory "/var/www/regression">
        Options +ExecCGI
        AddHandler cgi-script .plx
        RedirectMatch ^/regression$ /regression/regress.plx
</Directory>
```

and then make a link from '`/var/www/regression`' to the GNU Go regression directory. The `RedirectMatch` statement is only needed to set up a shorter entry URL.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by Daniel Bump on February, 19 2009 using texi2html 1.78.