Here are a few hints for finding your way around the source code. This doesn't make it less complex than it is, but it gets you started.

Jumping around

Use `ctags -R` to generate a tags file for the `:tag` command. (We recommend universal-ctags instead of the default `ctags` provided by most distros; see also CONTRIBUTING.md.)

To jump to a function or variable definition, move the cursor on the name and use the `CTRL-]` command. Use `CTRL-T` or `CTRL-O` to jump back.

To jump to a file, move the cursor on its name and use the `gf` command.

Most code can be found in a file with an obvious name (incomplete list): * buffer.c manipulating buffers (loaded files) * diff.c diff mode (vimdiff) * eval.c expression evaluation * fileio.c reading and writing files * fold.c folding * getchar.c getting characters and key mapping * mark.c marks * mbyte.c multi-byte character handling * memfile.c storing lines for buffers in a swapfile * memline.c storing lines for buffers in memory * menu.c menus * message.c (error) messages * ops.c handling operators (`d`, `y`, `p`) * option.c options * quickfix.c quickfix commands (`:make`, `:cn`) * regexp.c pattern matching * screen.c updating the windows * search.c pattern searching * spell.c spell checking * syntax.c syntax and other highlighting * tag.c tags * terminal.c integrated terminal emulator * undo.c undo and redo * window.c handling split windows

Important variables

The current mode is stored in `State`. The values it can have are `NORMAL`, `INSERT`, `CMDLINE`, and a few others.

The current window is `curwin`. The current buffer is `curbuf`. These point to structures with the cursor position in the window, option values, the file name, etc.

All the global variables are declared in globals.h.

The main loop

The main loop is implemented in state_enter. The basic idea is that Vim waits for the user to type a character and processes it until another character is needed. Thus there are several places where Vim waits for a character to be typed. The `vgetc()` function is used for this. It also handles mapping.

Updating the screen is mostly postponed until a command or a sequence of commands has finished. The work is done by `update_screen()`, which calls `win_update()` for every window, which calls `win_line()` for every line. See the start of screen.c for more explanations.

Command-line mode

When typing a `:`, `normal_cmd()` will call `getcmdline()` to obtain a line with an Ex command. `getcmdline()` contains a loop that will handle each typed character. It returns when hitting `<CR>` or `<Esc>` or some other character that ends the command line mode.

Ex commands

Ex commands are handled by the function `do_cmdline()`. It does the generic parsing of the `:` command line and calls `do_one_cmd()` for each separate command. It also takes care of while loops.

`do_one_cmd()` parses the range and generic arguments and puts them in the exarg_t and passes it to the function that handles the command.

The `:` commands are listed in ex_cmds_defs.h. The third entry of each item is the name of the function that handles the command. The last entry are the flags that are used for the command.

Normal mode commands

The Normal mode commands are handled by the `normal_cmd()` function. It also handles the optional count and an extra character for some commands. These are passed in a `cmdarg_t` to the function that handles the command.

There is a table `nv_cmds` in normal.c which lists the first character of every command. The second entry of each item is the name of the function that handles the command.

Insert mode commands

When doing an `i` or `a` command, `normal_cmd()` will call the `edit()` function. It contains a loop that waits for the next character and handles it. It returns when leaving Insert mode.

Options

There is a list with all option names in option.c, called `options[]`.

Code Overview (Visualization)

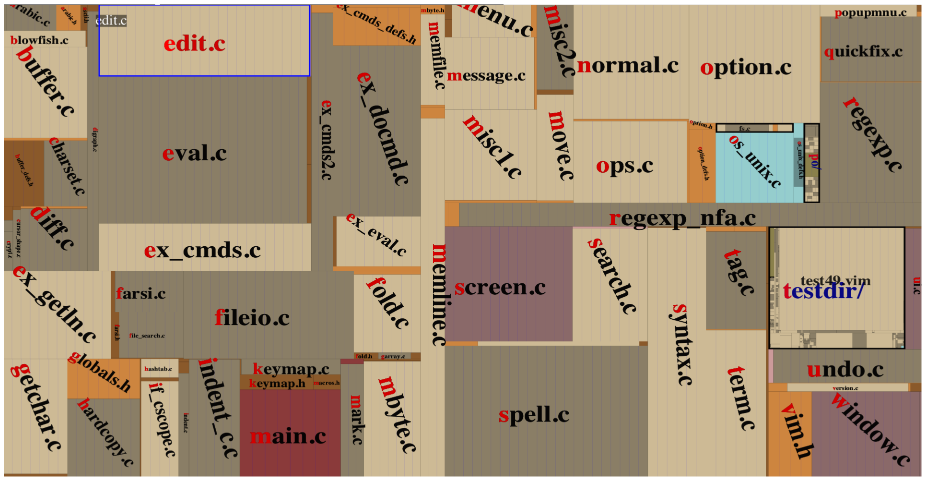

Figure 1: Code visualization generated with facebook/pfff