# Pushover

a board game playing program written in GNU epsilon
For epsilon version UNKNOWN, 15 February 2018

**Luca Saiu**

This is the manual documenting Pushover (for GNU epsilon version UNKNOWN, last updated on 15 February 2018).

Pushover is a board game playing program written in GNU epsilon and included in its distribution as a nontrivial programming example, as a compiler benchmark and as an interesting diversion. Like the rest of GNU epsilon Pushover is free software, distributed under the GNU General Public License version 3 or later.



Written by Luca Saiu

# Table of Contents

# 1 Introduction

I had not planned to include the Pushover game in the epsilon distribution before actually writing it. Now that it exists I am quite happy about it, as it constitutes the first worthy example of a program written in epsilon which is at the same time nontrivial and not part of the language implementation itself.

Like the rest of GNU epsilon, Pushover is free software.

## 1.1 History

Between late 2015 and early 2016 I was in charge of supervising a "tutored project" for two groups of around twenty first-semester Computer Science students at a technological university institute near Paris.

The students had about four weeks to implement a board game in C, working in pairs. The version they had to turn in was *for two human players only*, without computer play of any kind.
When first reading the specification I recognized the game as a slight variant of something I had played once a few years before during a Summer picnic at Parc de Sceaux, on a wooden set brought along by some friend of a friend.

The problem specification written by Mathieu Lacroix was very clear, and the level of difficulty appropriate for beginners. Students had to satisfy a strict set of requirements meant to guide them into writing first the helper functions they were to be needing later. Everything looked quite sensible.

When I eventually came to see the final work by my first group one Friday afternoon I felt disheartened. Many students had done badly, never overcoming their initial difficulties with pointers; a few programs did not even compile, many more failed to work correctly and almost all crashed when provided with incorrect input.

Walking back home I kept thinking about the problem. It was a neat little game described very clearly, with simple rules but nontrivial in terms of game complexity. I actually felt like learning to play the game well, for fun. Even more I wanted to implement something myself; something more advanced. Could I write a minimax version of it in a couple of days? And in epsilon, where debugging is still crude and unforgiving? That seemed a more interesting challenge, and I decided to spend the weekend working on it. Of course I was not constrained to follow the C specification and at times I used some very different implementation techniques; for example my move logic is based on recursive procedures over heap-allocated garbage-collected lists, which are copied off the board data structures and blitted back on; but my game implements the same rules dictated by the specification. On Sunday afternoon it was finished: 700 lines of quite clean epsilon code. I would only add a few minor things later, such as the command-line interface — at the time epsilon had no support at all for `argv`, and the option parsing library had to be implemented first.

The next morning I had to see my second project group for its last session. I announced at the beginning that I would show my computer-playing implementation of the game to the students who were interested after evaluation, and explain how it worked. Of course nothing of that was mandatory: everyone was free to leave right after showing me the code and answering my questions.
Even knowing from the beginning that it was stronger (students are grouped by grade) I

was quite positively impressed by my second group, which did much better than the first. So I was in a particularly good mood when I started explaining minimax on the whiteboard to the small bunch of students who had remained until the end. Then, seeing the computer beat me in a quick game (even despite my blunders) and play against itself raised expressions of wonder and a few ooohs — Those are the best moments about teaching. At some point another little crowd of students who had left before quietly came back into the room. At my interrogative expression they demanded to see the computer playing program I had promised. I was more than happy to reuse my game tree drawing still on the whiteboard to speak again about minimax, and then show the program once more. Without explaining the epsilon code I rapidly showed what it looked like, just to convey the idea that it was not overly long or complicated.
A few students remained much longer, deeply fascinated, asking questions about programming and languages.

Some days later I had a short discussion with Mathieu about introducing this kind of more advanced problems as optional tasks for the best students: we teachers could have shown something like what I did right at the beginning of the first session. Possibly. We did not come to a satisfactory answer, as it is not easy to motivate promising beginners without scaring off the weaker majority at the same time. Anyway I am glad to have followed my intuition showing the minimax program. There was some real beauty hidden right behind the beginner problem, needing just to be pointed at.

The original French specification named the game "Push Over". The French language tends to discourage neologisms and compounds but I feel no need to follow the example in English; therefore my version of the game will simply be *Pushover*. The only difference in rules with respect to the original specification is that in my version Black always plays the first move — which player began was not stated in the original. Apart from that for all practical purposes Pushover is the same game as "Push Over".

I wish to thank Mathieu for his nice project specification which inspired me to implement the game myself. His web page, also including the specification in French, is `http://www-lipn.univ-paris13.fr/~lacroix`. Mathieu told me he found the original idea on `http://jeuxstrategieter.free.fr/Push_over_presentation.php` (again, in French).

## 1.2 Purpose

Born out of a happy coincidence, Pushover is one of the rare instances of a program being both simple and truly realistic. It can serve at the same time as an example of epsilon programming and as a benchmark: the program does a considerable amount of computation, and the current version also generates quite a lot of garbage, stressing the memory system. Computing the best move is exponential in time, but not in (alive) space.

The program may be optimized (see Section 4.2 [Future development], page 13) but it is also useful as it stands as an example of something written quickly with convenience in mind. Since it can be run deterministically some specific version of it can be used as a benchmark to track the advances of the epsilon implementation — which currently remains quite inefficient.

With comparatively small changes Pushover could become a testbed for more or less advanced epsilon features which are still to come: for example `fork`-level parallelism would

be easy to exploit, and the system could run without garbage collection using manually handled memory regions for quick release of large heap memory blocks, in the style of GNU Libc *obstacks* (see Section "Obstacks" in *The GNU C Library Reference Manual*).

And I also find the game quite satisfying to play.

## 1.3 License

Pushover is free software: you are free to share and change it under the terms of the GNU General Public License, version 3 or later. A copy of the GNU General Public License is distributed in the file `COPYING` along with the software, and the text is also available at `http://www.gnu.org/licenses/gpl.html`. There is no warranty, up to the extent permitted by law.

This Pushover manual is free documentation, and therefore you are free to share and change it as well. The applicable license for the manual is the GNU Free Documentation License. A copy is distributed in the file `doc/COPYING.DOC` in the epsilon sources, and you can also read it on the web at `http://www.gnu.org/licenses/fdl.html`.

## 1.4 Contributing

Pushover is distributed along with GNU epsilon, in the same source archives. As a minor subproject Pushover does not need its own development infrastructure, and shares the resources used for epsilon.

Revision-controlled files, bug and issue trackers are available on GNU Savannah; see `https://savannah.gnu.org/projects/epsilon`. You can find the Pushover source code in `examples/pushover.e`, in the Git repository and the source tarballs. As is the case for most free software development the preferred communication medium about GNU epsilon is a mailing list. Discussions about Pushover are on-topic in epsilon lists.

If you have a bug to report or a patch to submit you can write to `bug-epsilon@gnu.org`; please include all the relevant information about your system, the software version you are using and how it was configured and compiled. When you are in doubt whether to include some detail or not, do.

In case you want to participate in the project development or discuss please make sure to have read Chapter 4 [Implementation], page 13, and then write to `epsilon-devel@gnu.org`. As of early 2016 `bug-epsilon` is an alias for `epsilon-devel` and there is no specific "help" list, but I can easily add more mailing lists should the need ever arise.

I encourage public technical discussion but also believe in respecting privacy. If for some reason you would rather reach me in private you can use my email address `positron@gnu.org`. My personal website is `http://ageinghacker.net`.

# 2 Pushover rules

The rules for playing Pushover can be learned in a few minutes by a child; with respect to the complexity *of the rules* Pushover is even simpler than checkers/draughts. However a game with simple rules is not necessarily easy to play well, as any chess player can witness. See also Section 2.4 [Game theory], page 8 for some preliminary discussion of Pushover's game complexity.

Pushover is game of two players named *Black* and *White*, played on a square board. Starting from an empty board Black and White take turns pushing one *piece* of their color from one of the four edges towards the interior of the board, horizontally or vertically. Whenever a piece is being pushed onto an already occupied case the previous piece is pushed along the same direction. If a whole row or column is occupied pushing one further piece from an edge causes the farthest piece to fall off (or to be *ejected* from) the opposite edge. A player is allowed to eject pieces of her own color, but not her opponent's.

The game ends when one or more rows or columns are completely occupied by pieces of the same color. At that point the player having more completely occupied rows and columns of her own color wins; if the sum of the number of fully occupied rows and columns is the same for each player, the game ends in a draw.

By convention Black moves first.

## 2.1 The board

The board is a square of *cases* having sides of length $n$ cases, for some predetermined value of $n$ in the range from 3 to 9, both extremes included. Each case in the board may be empty or occupied by one *piece* of either color.

```
  1 2 3
1 . . .
2 . . .
3 . . .
```

Figure 2.1: The initial position of a size-3 Pushover board, showing no pieces. The first move by Black will fill any one case in the perimeter; the central case is not reachable by the first move.

A board has four *edges*: *top*, *bottom*, *left* and *right*. A board is drawn as shown in Figure 2.1 for size $n = 3$. Rows and columns are numbered from 1 to $n$ included, with the origin top-left: then row 1 corresponds to the top edge, row $n$ to the bottom edge, column 1 to the left edge and column $n$ to the right edge.

Figure 2.1 shows an empty board, where every case is represented as a dot '`.`'. When some cases are occupied, for example as in Figure 2.3, Black pieces are represented as '`B`' and White pieces as '`W`'. In the program output the characters '`B`' and '`W`' may appear in two distinct colors. We call *position* any specific board configuration such as the one in Figure 2.1 or the left one in Figure 2.3.

## 2.2 Valid moves

The game starts with Black moving in an empty position. After Black plays her move it is White's turn and then again Black's, with the two players always playing one move each

until the game ends. A *move* consists of pushing one piece of the player's color inwards from an edge. The move can be written as two characters, the first representing the edge ('T' for top, 'B' for bottom, 'L' for left or 'R' for right) and the second a row or column number. For example pushing a piece downwards from the top edge into the second column is written as T2; the move to push a piece into the first row from the right edge towards the left is R1.

```
  1 2 3           1 2 3
1 . . .         1 . . .
2 B . .         2 W B .
3 . . .         3 . . .
```

Figure 2.2: Left: the current position right after Black has played L2 in the initial position, on a board of size 3. If White responds with L2, yielding the position shown on the right, the Black piece is pushed to the right by one case to make place. If instead White responds to the first move by Black in any other way the Black piece is not moved as the result of White's move.

Whenever the case into which a player moves is already occupied, one piece is *pushed* to make place, in the same direction of the player's move (see Figure 2.2). This displacement of the outermost piece may cause in its turn other displacements in the same row or column, until either the last displaced piece is pushed into a previously empty case, or the last displaced piece is ejected from the board and falls off the edge opposite to the insertion edge, as in Figure 2.3.

```
  1 2 3 4         1 2 3 4
1 B . W .       1 B . W .
2 W . . W       2 B . . W
3 B . B W       3 W . B W
4 B . . .       4 B . . .
```

Figure 2.3: Black plays T1 on the position shown on the left, ejecting a Black piece out of the bottom edge. The resulting position is shown on the right.

A player is allowed to eject a piece of her own color but *not* a piece of her opponent's color, as shown in Figure 2.4. A move which would cause the ejection of an opponent's piece is simply not allowed to take place, and can not be executed "partially" or in modified form. The Pushover program will stop a player attempting to perform an invalid move: for example if Black attempted to play L2 in the situation of Figure 2.4, the software would simply recognize the move as impossible and ask the player to enter a different one.
Causing a piece to be displaced without it being ejected *is* permitted, independently from the color of the displaced piece: the color restriction only applies to ejection.

```
  1 2 3 4
1 B B . .
2 W B W W
3 . . . .
4 . . . .
```

Figure 2.4: Black is not allowed to play L2 in the shown position, as in doing so she would eject a white piece from the right edge. Each player is allowed to eject her own pieces but not her opponent's.

Ejection only happens when the involved row (for a push from the left or right edge) or column (for a push from the top or bottom edge) is completely occupied: if at least one

free case exists in the row or column then a *hole* will be filled and no place will fall off the board, as shown in Figure 2.5. In every case the free case in the involved row or column which is closest to the edge from which the player is pushing will be occupied.

```
  1 2 3 4 5        1 2 3 4 5
1 B . W . W      1 B B W . W
2 . . . . .      2 . . . . .
3 . . . . B      3 . . . . B
4 . . . . .      4 . . . . .
5 . . . . .      5 . . . . .
```

Figure 2.5: Black plays `L1` in the position shown on the left. By doing this she pushes her own piece to the right, filling a hole. The new inserted piece is on the top-left corner. No other piece is displaced.

## 2.3 Victory and draw

Whenever a move yields a position where at least one row or one column is completely occupied by pieces of the same color, the game ends. At that point the balance of colors determines which player is the winner or if the game ends in a draw.

The Black final *score* is the sum of the number of rows completely occupied by Black pieces and the number of columns completely occupied by Black pieces. For this purpose every row or column counts as one point: the fact that pieces are aligned horizontally or vertically has no consequence. The White final score is determined in the same way, counting how many rows and columns are completely occupied by White pieces. If the Black final score is greater than the White score, then Black wins the game. If the White score is greater then White wins. If the two final scores are equal the game is a draw.

It is possible that a player plays a move yielding a winning state for the other by mistake, as in Figure 2.8. Such a blunder does not constitute an invalid move.

```
  1 2 3 4          1 2 3 4
1 B B . W        1 B B . W
2 W B . W        2 W B . W
3 . . . W        3 . . . W
4 . . B B        4 . B B W
```

Figure 2.6: Starting from the left position White plays `R4` and wins, yielding the right position in which Black has zero points and White has one full column (4).

Forcing a draw may be the best option in some game position, as an alternative to losing.

```
  1 2 3 4          1 2 3 4
1 . . W .        1 . . B .
2 W W B W        2 W W W W
3 B B W B        3 B B B B
4 . W . B        4 . W W B
```

Figure 2.7: Black forces draw by playing `T3` in the left position, which yields the right position where the two players have one full row each (2 and 3).

It is uncommon through not impossible that the final score is different from 0 for both players in a non-drawn game, as shown in Figure 2.8. Draws with both scores greater than 1 appear to be much more common.

```
  1 2 3 4 5        1 2 3 4 5
1 W W W . B      1 W W W . W
2 B B B B W      2 B B B B B
3 . . . . B      3 . . . . W
4 B B B B W      4 B B B B B
5 W W W W .      5 W W W W W
```

Figure 2.8: A blunder resulting in a more complex victory state: White plays `T5` on the left position, yielding the right position which is winning for Black: in the right position Black has two full rows (2 and 4) while White only has one (5).

## 2.4 Game theory

Pushover is a perfect-information, deterministic sequential combinatorial game. Any notion of game complexity for combinatorial games depends on the game *branching factor* and its *game-tree depth*.

*FIXME: the following paragraph is not completely exact. The initial complexity is less than $4n$: the board is empty, therefore pushing "into an angle" has the same effect independently from the direction. This changes later, when rows and columns fill.*

The branching factor for Pushover is maximum at the beginning of a game, being exactly $4n$ with a board of size $n$, decreasing near endgame. Games can last at least tens to hundreds of moves, and the game length grows as the board size grows. Branching factors for other combinatorial games are usually more variable; chess has a branching factor around 35 on average.

Without having developed any formal analysis (estimating typical values for either the game depth or the branching factor at a given depth is nontrivial) I conjecture that Pushover becomes more complex than chess for boards of size 7-8, and possibly already at size 6.

If empirical observation of computer vs. computer games (see [`--tournament`, page 10]) is to be trusted Black enjoys a considerable advantage on smaller boards, which decreases as the board size increases.

Most game theory considerations are moot when considering the current implementation, which is sequential and inefficient even when compiled due to a combination of a naïf compiler, suboptimal algorithms and a memory system fundamentally inadequate to collect short-lived data generated at a fast rate. The computer player uses a simple minimax algorithm without $\alpha$-$\beta$ pruning or any heuristic. All of this in practice limits analysis depth to a few plies.

# 3 Usage guide

The Pushover program has a terminal interface. As of early 2016 it uses ANSI terminal escape sequences to display pieces in different colors by default, even if it would be reasonable to check that `TERM` environment variable and use escapes only where supported, or possibly build upon some higher-level abstraction. Escape sequences can be disabled with a command-line option (see [`--no-color`], page 10). The GNU Readline (see *GNU Readline Library*) library is supported if epsilon has been configured to use it.

The Pushover program is a loop playing every move of a game — or more than one in the case of tournaments; see [`--tournament`], page 10. At the beginning of each turn the computer displays the current position; then a move is played until a final position is reached.

## 3.1 Player types

Being conceived in a quite orthogonal way, the program supports different types of games: *human versus human*, *human versus computer*, and *computer versus computer*. Computer versus computer games are non-interactive, and the user can simply watch games unfold; this is particularly useful to compare how effective the different kinds of algorithms are — see [`--tournament`], page 10.

Each of the two players may be of one of the following three types:

- *human*, in which case the game is interactive. The computer displays the list of every valid move in the current position clearly showing which player is moving in the current turn, and waits until the user types in a move in the notation of [move-notation], page 6, followed by `Enter`. Since no ambiguity is possible the program also accepts edge letters in lower case, as a convenience. If the user enters an invalid move the computer asks again until she eventually types in a valid move which is then played, and control is passed to the other player unless the game is over. At the prompt a human player can also quit the program by typing `C-d` on an empty line.

  A human player may optionally benefit from *computer hints*, computed with the minimax algorithm described right below. If hints are enabled then the computer searches for an optimal move at the specified depth in plies, and presents it to the user; the user is then free to follow the computer suggestion or reject it and play a different move.

- *minimax*, a straightforward implementation of the classical algorithm without $\alpha$-$\beta$ pruning or any heuristic. Analysis *depth* may be specified in plies, from 1 to 9 included — a depth of 9 plies being already impractical with the current implementation. The algorithm is exponential in time, but not in space.

  In minimax play the computer normally chooses a random move within the set of moves which are considered optimal by the algorithm, but computer play can also be made deterministic — in which cases the computer always plays the first optimal move according to some fixed order.

- *dumb*, which consists in considering all valid moves as optimal and playing any one chosen at random or, in deterministic mode, the first one.

  Dumb playing may be of some use for learning the rules of the game and for testing. It comes in handy for benchmarking the algorithm used by another computer player and

I may possibly use it for developing other algorithms in the future, to compare against a baseline.

Two deterministic algorithms played against one another are prone to enter a *game loop*, endlessly playing the same sequence of moves; in rare circumstances nondeterministic algorithms may enter a game loop as well, for example when the set of optimal moves is a singleton for two consecutive plies, and the second move yields the same position as before the first move. The best way to solve this problem would be to introduce a new rule in the spirit of the *threefold repetition* rule in chess; see Section 4.2 [Future development], page 13.

## 3.2 Command-line options

The `pushover` program has no non-option arguments: every argument it recognizes is an option, currently always in the long GNU style (see Section "Command-Line Interfaces" in *The GNU Coding Standards*).

Every option taking a parameter, shown below with a '`=`' sign followed by a parameter placeholder, may be provided either as a single argument including the '`=`' sign followed by the parameter, or as two separate arguments without the '`=`' sign: for example the program will indifferently accept '`--black=m3n`' and '`--black m3n`'.

If the same option is specified multiple times with different parameters *the last parameter* will take precedence, which may be convenient for shell aliases.

No option is mandatory.

The Pushover program accepts the two standard GNU options:

`--version`
: Print version information and legal notices, then exit successfully.

`--help`
: Print a short summary of command-line syntax explaining every option, then exit successfully.

The following option affects board drawing:

`--no-color`
: Display game positions and player names in a single color, without outputting any terminal escape sequence. If the option is not specified then the program uses ANSI terminal sequences; this should probably be changed so that the program only uses such sequences on supported terminals, according to the value of the `TERM` environment variable.

The following two options control game parameters:

`--size=`*n*
: Play on a board of size *n*. The default board size used when the option is not specified is $n = 4$.

`--tournament=`*n*
: Instead of a single game play a *tournament* of *n* games one after another, at the end printing statistics about Black victories, White victories and draws. The statistics format is as follows:

```
Score over 100 games (size 4):
* Black (minimax 4-plies deep): 26%
* White (minimax 4-plies deep): 18%
* Draws: 56%
```

Tournament mode is only enabled for $n > 1$; when the value of $n$ is zero or negative only one game is played, with no final statistics. Tournament mode is off by default.

The last two options below, likely the most commonly used, determine the player type for each side:

`--black=`*playerspec*
`--white=`*playerspec*

Specify the player type for Black or for White as shown in Section 3.1 [Player types], page 9. The value of *playerspec* follows a rigid syntax:

- '`h`' or '`h`*n*': human player. When provided *n* must be a decimal digit from 1 to 9, representing the minimax search depth in plies for hints; hints are disabled if *playerspec* is simply '`h`'.
- '`m`*n*' or '`m`*n*`d`': minimax computer player with *n* plies deep search where, again, *n* is a single digit between 1 and 9 included. The '`m`*n*`d`' version forces deterministic play.
- '`d`' or '`dd`': dumb computer player, in random ('`d`') or deterministic mode ('`dd`').

The default *playerspec* is '`h3`' for `--black` and '`m5`' for `--white`: a human playing as Black with depth 3 hints versus the computer as White with minimax at depth 5.

When the user only specifies one of `--black` and `--white`, only the one corresponding default is overridden. Combining `--black` and `--white` yields different type of games: it is permitted for the two players to be both human, both computer, or one human and one computer in either role.

## 3.3 Performance considerations

???rephrase and expand???

A user in search for a more difficult challenge wishing to play *as White* against a minimax player at the same depth as the default opponent could invoke the program as:

```
pushover --black=m5 --white=h3
```

A user accepting to tolerate the slowness of a depth 7 search could play against a very strong computer opponent:

```
pushover --white=m7
```

Search time being exponential with a branching factor around $4n$ on a board of size $n$, a one-ply deeper minimax player (or hint) slows down computer play by a factor of roughly $4n$, more than one order or magnitude even for $n = 3$. Deepening search by two plies would make search run $16n^2$ times slower.

# 4 Implementation

## 4.1 Minimax

Section 4.2 [Future development], page 13

## 4.2 Future development

??chess and other games, likely depending on functors??

??breaking game loops by preventing repeated positions, in the spirit of the threefold repetition rule in chess???

## 4.3 Rule change

# Index

## H



## I



## L



## M



## N



## O



## P



## Q



## R



## S



## T

## U



## V



## W