

reverxii

Playing Reversi in T_EX

Bruno Le Floch

November 6, 2021

Abstract

`reverxii.tex` is a 983 character long T_EX program which lets you play Reversi against your favorite typesetting engine. To play, run plain T_EX or L^AT_EX (either the dvi or the pdf engines), on the command line, on the file `reverxii.tex`. In most distributions, this can mean running `tex reverxii.tex` in a terminal. To typeset the documentation, run L^AT_EX on `reverxii.dtx`, with any engine (*e.g.*, `pdflatex reverxii.dtx`).

1 The code

Line breaks can be removed, and the stripped down code takes 983 characters.

```
\ifx\>\: \documentclass{report}\fi
\vsizem\hsize3cm\newlinechar'\def~#1{\catcode'\#113~}
~IJKLMNOPQRSTUVWXYZjzqz@. [ ] (/+^'";:?,_)!*{ 13\def~#1#2{\let#1#2~}
\def+{\count\ifx\>\:1\fi1}+1=2)*\cr[\ifnum\(\ifcase\or/\else\fiN\number
@ \advanceX\expandafter!\global? \message~\def.#1{ @+1 1\countdef#1+1.}
.IJKLPQRSTUVWXYZ.-1P1T2+44P+55P+45T+54T
~j{[0<Q[9>Q[0<J[9>J^-././././././]}
~^{+NQNJ}~:~#1{#11#12#13#14#15#16#17#18}~M#1{?{#1}#1}
~_#1#2{M#2:{\B#1}&M#2&M{*}}~\B#1#2{&M{(+ #1#2 | -|0]}}
~q{?{Row and column? e.g. E6*}\read.to\EXD\meaning\E ;}
~\D#1->#2#3#4;{Q'#2@Q-'@J'#3@J-'@}(V?{Invalid move #2#3.}q]}
~){V0 (jS1z1z0z.S0z1z.S.z1z0z.]}~;{@R(P|-)]~z#1{{K#1Z1{Y1,}(Z,]}}
~,{@QS@JK[j=T(Y!^P!;2]0,/ [j=P!VV(I(Y/!Z0]]]]~\A#1{Q#1:\C}
~\C#1{J#1)[0<V0[V>WVUQLJ]]~"~#1{(#1|0|1|2|2|2|2|1|0]}
~0{Z"Q\multiplyZ3@Z"J@V(Z9|1|6|1|1|2|6|2|4] }~'~M :{&M}&M{*}}
~~{PXTXTNP\halign{&## *M{*}}'_1A_2B_3C_4D_5E_6F_7G_8H'}\vfil\break
I1W(W./0] : \AI0 [0<W[1=PQUJL/q])^P;1] [. =W?{(RTie/ Player [0>R-/0]
wins by N[0>R-]R].}X\dump]~}~
```

In fact, the first line and `\ifx\>\:1\fi` are only used to support L^AT_EX and can be removed for plain T_EX. This reduces the code down to 938 characters.

To play a two-player game, change `1=P` to `0=P` near the end: this changes the computer player from 0 to 1, hence disabling it.

2 Explanation

2.1 Some comments

The name `revertxii.tex` is of course a reference to the infamous `xii.tex`, also on CTAN, which typesets the lyrics of the *Twelve days of Christmas* using code that very few can understand. In my case, I aimed for the shortest possible code, rather than the most obfuscated. In particular, I did not assign weird catcodes other than making most characters active.

Since I was aiming for short code, the text presented to the player is concise, but hopefully enough to leave the game understandable and usable. Despite aiming for short code, I thought it would be neat to typeset a record of the game as it goes: after all, $\TeX$ is a typesetting program. This used up around 38 characters, putting me above the 900 character line.

One technique used to make the code shorter is to rename any primitive used more than once into a single active character. Also, counters holding the information about the board are accessed directly by number.

A careful reader would notice that changing one of the remaining one-character control sequences to active characters ($\$$ is still unused) would gain one character. I didn't do it, because it messes up the code-highlighting of my editor :). Of course, I chose the control sequences which are not active characters to be those appearing the least, only twice each.

2.1.1 On with the code!

In $\LaTeX 2_{\epsilon}$, detected by comparing the spaces `\>` and `\:` (only one of them is defined in plain $\TeX$), load a document class.

```
\ifx\>\:\documentclass{report}\fi
```

We do not need `\begin{document}` because the text is eventually typeset inside an `\halign` table, which means `\everypar` (and its usual error) is never called.

First set up the page dimensions. This would not be enough for pdf output.

```
\vsize5cm  
\hsize3cm
```

Since plain $\TeX$ does not provide the `\typeout` command, we are using `\message`, hence need to setup a new line character. It is arbitrarily chosen to be `*`, which will be made active and `\let` to `\cr`.

```
\newlinechar'*
```

Then a first loop to make many characters active. The loop ends when reaching the trailing brace group: there we make spaces active, then redefine `~` for the next loop. We still have an annoying `13~` in the input stream. Introduce a short-hand for count registers whose number starts with `1` (or in fact `11` in $\LaTeX$ given the test `\ifx\>\:1\fi`, see later for why we do this). Then remove `13` by setting `\count11` to `213` (this counter is used for allocation later on). The next loop is triggered by the `~` which we left.

```

\def~#1{\catcode'#113~}
~IJKLMNOPQRSTUVWXYZjqz@.[|](/+^'";:?,_)!*
{ 13\def~#1#2{\let#1#2~}\def+{\count\ifx\>\:1\fi1}+1=2}

```

At this stage, `~` is defined to take two arguments, `\let` the first to the second, and repeat. As announced `*` becomes `\cr`, so that it will be a new line both in messages and in alignments. We try to keep a relatively consistent naming scheme: opening conditionals are left delimiters, `\or` and `\else` are middle delimiters, and `\fi` is a right bracket. Other primitives which are used a lot are also given short names. The loop ends by redefining `~` to `\def`.

```

*\cr
[\ifnum
(\ifcase
|\or
/\else
]\fi
N\number
@\advance
X\expandafter
!\global
?\message
~\def

```

It is time to allocate counters. Unfortunately, `\newcount` is `\outer` in plain $\text{\TeX}$, so it is unpractical to use. We have defined `+` to be `\count1` in plain and `\count11` in $\text{\LaTeX}$. This will eventually let us access a table of 8×8 counters `+11` to `+88`, which are thus `\count111` to `\count188` in plain, and `\count1111` to `\count1188` in $\text{\LaTeX}$. This is needed because plain $\text{\TeX}$ without $\text{\eTeX}$ extensions only has 256 counters, while $\text{\LaTeX}$ uses some of these counters already so we must use the higher counters provided by the $\text{\eTeX}$ extensions, always available in $\text{\LaTeX}$. The following text explains the plain $\text{\TeX}$ case; in the other case just replace `\count1` by `\count11`.

We have set `+1`, *aka.* `\count11` to 213, and will allocate counters starting from that number upwards. Repeatedly advance `+1` by 1 and define the next character to be the counter number `+1`, then repeat. As always, the loop is interrupted by making it redefine the looping macro `.` to be a counter. We use the trailing dot to assign it the value `-1`, then assign a couple of counters: starting player, other player, and the initial position: the squares in rows and columns 4 and 5 are initially filled in a cross pattern.

```

.#1{ @+1 1\countdef#1+1.}.IJKLPQRSTUVWXYZ.-1P1T2+44P+55P+45T+54T

```

Let us summarize which counters are used where:

P is the current player (1 for - or 2 for 0);

T is the other player;

Q is the row number;
J is the column number;
S is the step size in the row direction;
K is the step size in the column direction;
R is the score difference, positive when 0 is winning;
V is used when computing the value of placing a piece at the position (Q,J);
W is the value of the best possible move according to the AI, also used to end the game if neither player can move;
U is the row number of the best move;
L is the column number of the best move;
Y is a boolean, true (0) most of the time, it has to do with when we flip or not, but I don't understand it now, help welcome;
Z is a temporary counter, used locally to compute how much a given cell matters (*i.e.* corners are important), and used globally as a boolean to indicate whether to flip pieces or not.

The board is stored in counters `1<row><column>`. An empty square is has the value 0, a square for player - has value 1, and player 0 corresponds to the value 2. The macro `j` retrieves that value, assuming that the row and column numbers are stored as Q and J, and returns `.`, that is, -1, if outside the board. Recall that `[` is `\ifnum`, `/` is `\else`, and `]` is `\fi`. When Q and J are within bounds, the value is retrieved by `^` as `\count1<row><column>`. Note that `j` and `^` are only safe to use on the left-hand side of an `\ifnum` test (because of expansion issues), and that `^` can be used on the left-hand side of an assignment.

```
~j{[0<Q[9>Q[0<J[9>J^/.]/.]/.]/.]}
~~{+NQNJ}
```

We often need to loop over numbers from 1 to 8; here is a macro.

```
~:#1{#11#12#13#14#15#16#17#18}
```

The macros to print the board, both to the dvi and to the console. `M` spews its argument as a `\message` (?), and directly typeset. This is used rather directly to print the first and last lines (`'`), which are simply alignment cells containing each number from 1 to 8, with some care given to spaces and new lines. The `_` macro receives a digit and the corresponding capital letter as arguments, and outputs that row of the board. First place the letter on the left of the board, then loop from 1 to 8, typesetting and `\message`ing a space, - or 0, depending on the value of the relevant count register. Note the two spaces in the definition: the first ends the counter's number, the second is typeset (in plain T_EX, active spaces expand to a normal space).

```

~M#1{?{#1}#1}
~' {M :{&M}&M{*}}
~_#1#2{M#2:{\B#1}&M#2&M{*}}
~\B#1#2{&M{(+ #1#2 | - | 0)}}

```

The input is done by `q`, which queries the user until they give a correct input, so that `Q` and `J` are in the range $[1, 8]$. Prompt the user with a small `\message`, giving an example of what move he could do (only true at the start, but the hope is that the player will understand what the input format is). The code that follows is similar to L^AT_EX 2_ε's `\@onelevel@sanitize`. We extract the two first characters from the the `\meaning` of the user's input (remember, `X` is `\expandafter`), as `#2` and `#3` of `\D`. Grab the character code of each, relative to the characters `@` (row) or `0` (column). The closing parenthesis is where most of the work is done. It sets `V` to the value of placing a piece in the cell (Q, J) , zero if the move does not flip any of the opponent's pieces, or if the cell is outside the board. After performing that calculation, if `V` is zero, the move was not valid, and we query the user again.

```

~q{?{Row and column? e.g. E6*}\read.to\EX\D\meaning\E ;}
~\D#1->#2#3#4;{Q'#2@Q-'@J'#3@J-'0}(V?{Invalid move #2#3.}q)}

```

So... how do we compute the “value” of a move? It is automatically invalid if `j` does not return 0: either the cell is occupied, or it is outside the board. Then for each of the 8 directions around the cell, we count the number of pieces that are flipped in that direction. The direction is stored as two counters, `S` and `K`, each ± 1 or 0. We call `z`, a first time to test whether flipping should happen in that direction, and, if it is (as indicated by the global value of `Z`), call it again to actually flip. The call to `z`, must happen within a group, because it directly changes the row and column numbers `Q` and `J`, following which cell is being queried.

```

~){V0 (jS1z1z0z.S0z1z.S.z1z0z.]}
~z#1{ {K#1Z1{Y1,}(Z,]}

```

The macro `z`, is recursive. At each step, move (Q, J) in the direction (S, K) . Then, if that cell (j) contains a piece belonging to the other player (`T`), do some stuff (`Y!~P!;2]0` and repeat. What is it that we do? Well, if the condition `Y` is met (I don't remember how that works), we set the current cell to belong to the player, globally, and change the score difference by 2 (see `;`), also globally. Then, we compute with `0` the value corresponding to the cell that we might be flipping (see `0`).

Otherwise (`/`), if the cell (j) contains a piece from the current player (`P`), it means we have reached the end of a run of the form $\langle \text{initial cell} \rangle \langle \text{opponent's pieces} \rangle \langle \text{own piece} \rangle$, hence the $\langle \text{opponent's pieces} \rangle$ should count as flipped if we place our piece in the $\langle \text{initial cell} \rangle$. Until there, all changes to `V` were local, returning to the old value at the end of the group that `z`, is contained in. Since the run in that direction was successful, we escape the value of `V` from the group with `\global V=V`. Under some conditions, we set the boolean `Z` to true, globally

(!Z0), which triggers a second call to , with different setting, and actually flips the opponent's pieces.

```
~,{@QS@JK[j=T(Y!^P!;2]0,/ [j=P!VV(I(Y/!Z0]]]]}
```

I moved those 0 and " guys a little bit in this explanation, but not in the original implemenation, because it is hard to sync. We assign weights to each of the 64 cells:

```
9 1 6 6 6 6 1 9
1 1 2 2 2 2 1 1
6 2 4 4 4 4 2 6
6 2 4 4 4 4 2 6
6 2 4 4 4 4 2 6
6 2 4 4 4 4 2 6
1 1 2 2 2 2 1 1
9 1 6 6 6 6 1 9
```

All weights are positive, so that every move which flips a piece ends up with a positive overall value. The AI would be better if the places next to corners had a negative weight, but I would have too much code to rewrite for that to work. We really have three types of rows and three types of columns. Converting from Q or J is done by ", then we assemble the two as a number in the range [0, 8], and use another \ifcase construction to produce the weights.

```
~0{Z"Q\multiplyZ3@Z"J@V(Z9|1|6|1|1|2|6|2|4] }
~"#1{(#1|0|1|2|2|2|2|1|0]]}
```

The counter R keeps track of the score difference, and is updated with ;2 (when flipping a piece) or ;1 (when adding a piece). The counter R should be \advanced (@) by a positive amount when the current player P is player 0, and a negative amount for player -.

```
~;{@R(P|-]}
```

After printing the board, we go through every cell and find the one with the highest value. The macro \A, does one row, hence \:\A does all the rows. Store the argument as the row number Q, then loop over columns. After setting the column number J to its argument, \C calls), which as explained above computes the value of placing a piece there, throws away that case if it flips nothing, otherwise also counts the weight of the current cell. Then update the best value W and the best row U and column L if the new V is larger than W.

```
~\A#1{Q#1:\C}
~\C#1{J#1}[0<V0[V>WWWUQLJ]]}
```

We won't need ~ as \def anymore, so we reuse it as the main command.

- First exchange the players: set P equal to T, but first expand the value of P after T to set that as well.

- Secondly, open an alignment, with a repeating preamble adding a space at its end. Then message a new-line (we should be using ? rather than M here, I think) to keep a clean output. Afterwards, print the top line with ', the eight lines of the bulk with _, and the bottom line, which happens to end with ?{*}*, *i.e.*, ends with \cr: we can thus close the alignment, and cause T_EX to output the page.
- After printing, it is time to check whether there is a move or not. We don't want to flip pieces at this stage, hence set the boolean I to false (1). And we reset the best value to 0, unless it was already 0 (which means that the previous player had no available move), in which case we set it to -1. Then loop over rows, finding the best value (see \A). Reset the boolean I to be true.
- If a move was found in the previous step ($W > 0$), either use it if the player is the AI, or query the user. The various booleans are set up in such a way as to do the flipping, so calling) does it. Then also put a player's piece in the current cell ^, and increase the score difference by 1.
- Finally, if neither player could move, declare the game ended, give the score, and \dump the run (not \end because we want to support both plain T_EX and L^AT_EX). Otherwise, repeat.

Of course, after defining ~, we call it. Let's play!

```

~~{
  PXTXTNP
  \halign{&## *M{*}'_1A_2B_3C_4D_5E_6F_7G_8H'}\vfil\break
  I1 W(W./0) : \AI0
  [0<W
    [1=PQUJL/q]
    )
    ^P;1
  ]
  [.=W
    ?{(RTie/ Player [0>R-/0] wins by N[0>R-]R).}
    X\dump
  ]
  ~
}
~

```