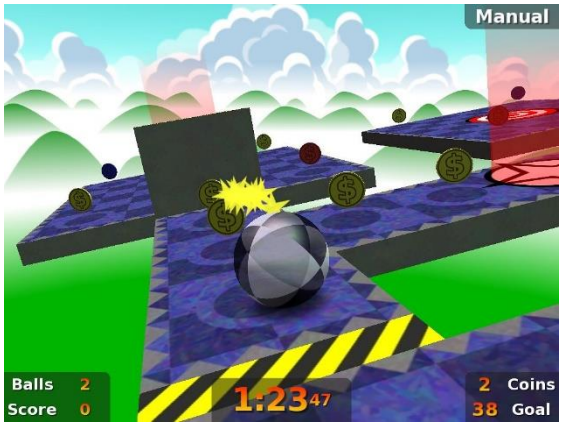
“NeverCruncher” AI
- Explanation in three steps.

Preamble: Neverball is a cult sandbox game, popular on Windows and the Linux platform. A table is tilted to keep a ball on it and to collect coins. It is open source and has a fanatical fan base.

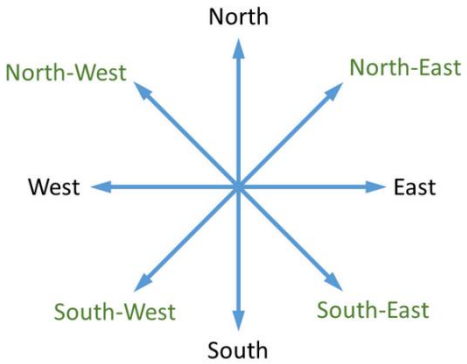


Setup: The codebase was examined, and an agent was written in, having ***no world knowledge***. The information available was.

- The number of coins collected.
- The position and, if this changes, the speed.
- When the ball has left the level having fallen from the table.

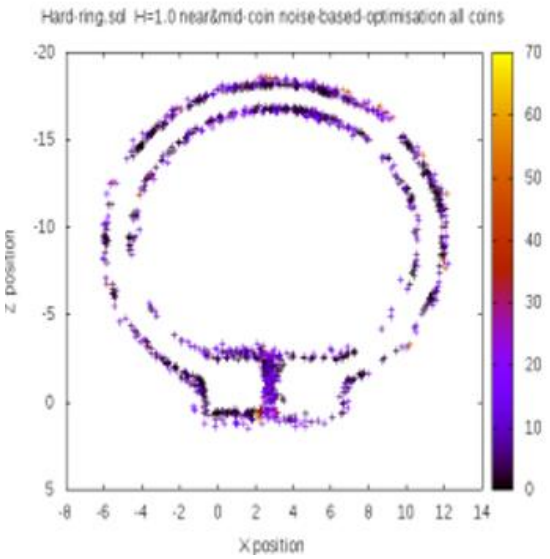
Step 1/3: Tilt input

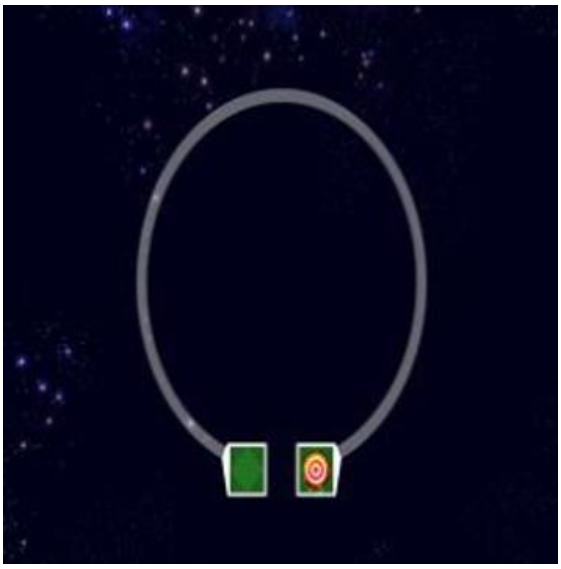
I edited the code to tilt the table roughly four times a second, in eight cardinal directions, or none, so nine options exist for each time step.



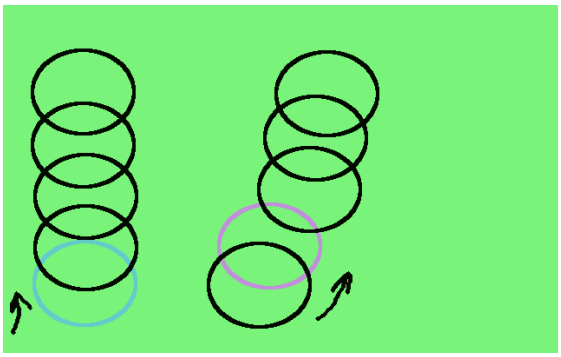
Each direction was explored for each ¼ second step [breadth first]. The reward was random, unless a coin was collected in that step or a new location was entered. The most rewarded tilt was saved, and the next step was tried.

Step 2/3: Fallout Correction





Fallout was only detected when the ball had fallen through the bottom of the level and the game had been lost. A tilt several steps before that, before the ball had left the table, was chosen to be responsible.



That tilt was excluded from the path and new tilts were explored from that point on following the method of step one. This could result in two things.

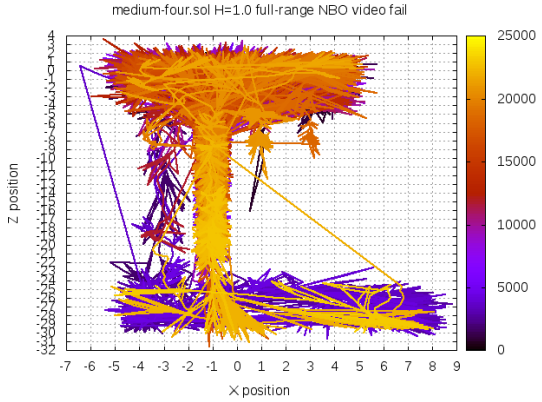
① the ball continues to fall out and the same timestep was blamed until all nine tilts at that step were blocked, if this happens then the step previous to it is blamed and its tilt at that time was blocked in the same way.

② The ball does not fall out, or falls out at a later time, but then a later tilt is blamed for the new fallout and the path has been changed to one that falls out later and this can be repeated until the fallout does not occur.

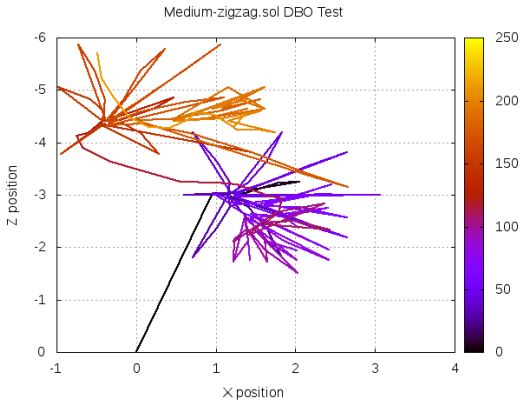
The idea of this is to alter a tilt sufficiently back in time that the fallout can be prevented. Too late a step and this is impossible, too early and the algorithm is slowed and progress is wasted. For faster fallouts more space is needed to decelerate so an earlier step is blamed.

Step 3/3: Coins

To make sure coins are collected exploration with semi-random tilts was repeated, goals were set, like “two coins in two seconds”, and this would run e.g. 7 times and if successful would reset and then try to get two in the next two seconds.



If it failed, then less ambitious goals would be tried a few times until they timed out or succeeded and then the next few steps would get the same treatment. Because of the lack of world knowledge this was necessary to explore.



Direction-based exploration was also used. For the two seconds each direction was held to see if the goals would be met. This fanned the paths in nine directions and explored more area whilst the previous noise-based method could produce more unusual paths. Both were used.

First Instance of fallout-correction success(x2 speed)
https://www.youtube.com/watch?v=Cw8WDSlvWxo

Optimisation/Coin-Collection on a simple level:
https://youtu.be/GRioL03pXKs

Optimisation on a harder level:
https://youtu.be/SWS_KyA7hMg

Montage Of several levels with music:
https://youtu.be/aleKS4S4U9E

Failings

The algorithm is largely brute-force and computationally expensive. Raised bumps on the table surface cannot be reached with this, some reward system based on elevation could work. Speed is not specified but a certain speed is appropriate for certain levels. Some planning could be input, such as a good area to get to. Odd things are possible, like change of prevalent direction or curl, and could be rewarded. This program could work for other systems by replacing the nine options with, for example, combinations of buttons or moves.