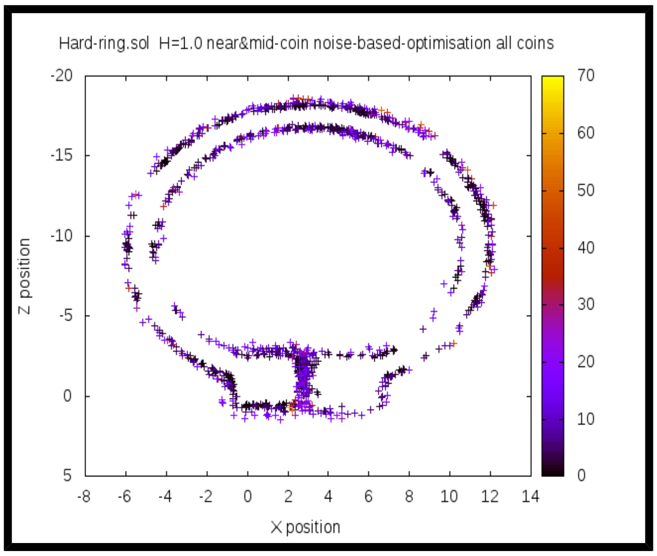
NEVERCRUNCHER DEVELOPER GUIDE

Firstly I would advise that not only does this program take up to and sometimes well above 14 hours to complete depending on the optimisation function, it is also complicated and hacky and unfinished. Currently (Oct 2017) it does not work for levels that have raised platforms because it assumes all balls fall from the same height. Neither does it find or look for or react to the level goal. Results will vary unless the seed is set to a constant value and the optimisation function is currently set to a very high amount of direction based optimisation which may not suit. There are no graphics during crunching and often the program will have to be restarted once crunching succeeds. This is a proof of concept that uses no world knowledge, other than the position and speed of the ball, amount of coins and whether it has passed the lowest part of the level (Fallout), after falling off the table. In developing NeverCruncher I would encourage you to not introduce world knowledge to the program.

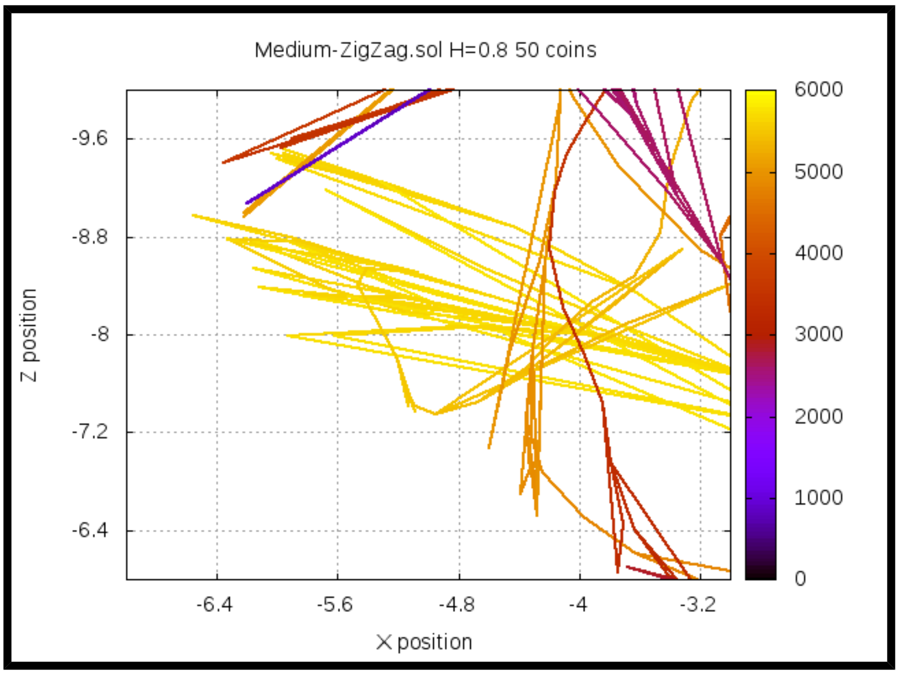
Here I attempt to explain the operation of NeverCruncher and its processes with a view to guiding further development and making improvement as pedestrian as is possible. The current anatomy of the program is shown at the end of this document in full. My comments to the code are copious and are prefixed with JK unless it is obvious whose they are. The bulk of my code is in st_play.c excepting a few calls to get data from the game and update the function. These can be found by searching for my prefix.

The program runs till the state being explored is reached then one option is tried and evaluated and then the program exits. Once nine options have been evaluated the best is taken and recorded and the state being explored is incremented.

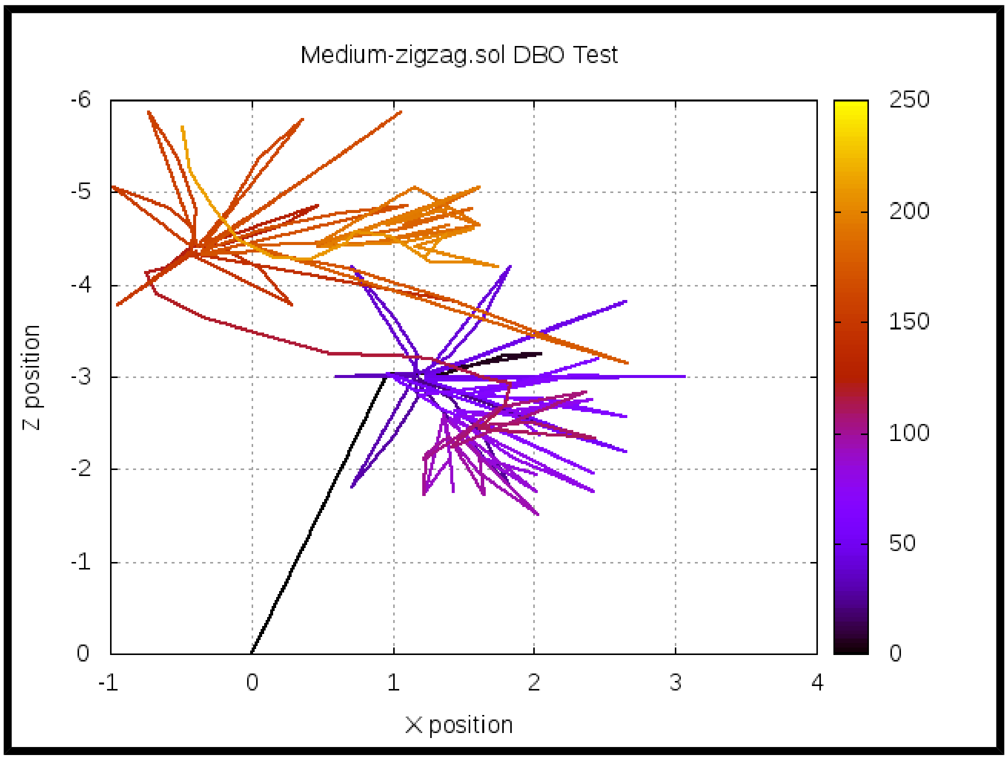
There are three main processes. The first is the reward signal which assigns a float to each tilt option on each step, this is composed of the number of coins plus a noise factor minus a heuristic penalty. The heuristic takes a value from the reward if the ball is in the same square for more than once step. The second is fallout correction which detects fallout and rewinds a number of steps and then finds which option was tried that step and blocks that option (hatestate) and re-explores the state. The third is optimisation/correction, in which should an optimisation condition not be met the state being explored is rewound and either explored normally (noise based exploration), or explored with directions fixed (direction based exploration) until the condition is met or until the optimisation counter runs out.



Fallouts for ring.sol coloured by velocity.



Noise-based-optimisation creates a fanning effect on the path.



Direction-based-optimisation creates these stars.

TODO – A more effective fallout-correction mechanism is required that will track the Y coordinate and detect fallouts from raised areas. The heuristic could be extended to punish the agent for repeated visits to the same square, square size could be changed. Noise-based-optimisation and direction based optimisation have their own merits but both are used. Optimisation can be based on any factor but currently it is based on coins collected. For example another layer of optimisation could be applied to ensure exploration is optimum. One of the obstacles you will face is that it takes 14+ hours to test and testing is paramount.

FINAL ALGORITHM DESIGN

<Here "I" increases from zero to the step being explored exploring one option of one step in a run and exiting. Steps less than the state being explored are read from an optimised array. The step being explored is incremented once 9 options have been tried>

I = zero

If first run of game, folders created

If Fallout occurs

Rewinds N states proportional to Velocity and Height of fall.

Writes HateState for state being rewound to.

Then the tilt and rewards for each step are re-evaluated.

I < Step beingExplored

Read Tilt from array.

Record old options selected for use by Fallout-

Correction-Mechanisim

I = Step Being Explored

Read Hatestates and ignore option if is a

hatestate.

Set option from NSEW list

I = Step Being Explored plus one

Get reward from options set in previous step

choose best of nine option and write to tilt array.

At this point the stats for the agent are recorded.

These stats trigger the optimisation methods.

Optimisation Rewind

The state are rewound a number of steps according to the property that is looked for.

An iterator is increased each time.

This recurs till either the property is found or the iterator reaches its limit.

<END OF METHOD>

The program iterator "I"starts at Zero and runs on till it gets the step being explored. If the state being explored is tried enough times it moves to the next state and so on till the timespace has been filled.

Narrative

---The first run creates the folders that will hold the log files.

----Fallout is when the ball is detected below the baseline. It corresponds to a poor option chosen several steps back. So the step being explored is rewound to that state and the option responsible is added to the list of hatestates for that time. And the steps are repeated with that option blocked.

-- The state's tilt is read from a logfile and the options taken are held in memory so they can be flagged as hatestates if they lead to a fallout.

-- 9 options in total are examined for each state N,S,E,W, NE,NW,SE,SW and no tilt. Hatestates are skipped as they lead to fallout.

---- The reward gained by a given exploration direction is found a step later.

-- The optimisation property can be anything but is generally a number of coins gotten by a number of steps.

-- Once the program completes exploration stops and the log files are written.

<Should you have any problems private message computron59 on neverforum.com>