Algorithm analysis

Foreword

We will discuss possibly random move sequences as potential algorithms. For simplicity we will use the terms "move sequence" and "algorithm" interchangeably, even though usually only very few move sequences with certain properties would be called an algorithm. We will try to come up with some criteria that differentiate them.

Some of the ideas explained here are already implemented in `tests/algorithm_analysis.py`

potential use cases

1. One goal is to create a new less resource expensive AI that mimics how humans solve twisty puzzles. More specifically the AI is supposed to develop it's own algorithms and solve the puzzle using those.

2. Coming up with new algorithms yourself can be quite timeconsuming and is prone to errors caused by miscounting or accidentally making the wrong move. This program could analys a move sequence automatically to determine whether or not it's worth remembering as an algorithm.

Information about a move sequence

We can classify move sequences using several criteria:

1. order of the algorithm
2. number of pieces changed by the algorithm
3. order of the piece permutation, disregarding piece rotations
4. order of the piece rotations, disregarding piece movements
5. whether or not pieces are...
    1. moved , ...
    2. rotated or ...
    3. moved and rotated
6. types of affected pieces (only if there are different classes of pieces in the puzzle)
   The information 2-5 can also be calculated for each piece type individually.

A desireable algorithm should only affect a small number of pieces (ideally 2 or 3, possibly up to 8). It should also either only rotate piece or only move them, not both at the same time.

How to calculate the different pieces of information

1. order of the algorithm *(implemented ✔)*

Requirements: ✔

- move sequence
- move permutations

This is probably the easiest part:

> 1. concatenate all move permutations in the move sequence
> 2. calculate the order of the resulting permutation

An implementation of this using `sympy` may be the fastest solution.

---

2. number of pieces changed by the algorithm *(implemented ✔)*

***Requirements:* ✔**

- move sequence
- move permutations
- piece sets

> 1. concatenate all move permutations in the move sequence
> 2. for every affected point, find the corresponding piece

Alternative:

Requirements:

- move sequence
- move piece permutations (the moves only describing piece movement without rotation)
- piece sets

> 1. concatenate all piece permutations in the move sequence
> 2. list all affected pieces

---

3. order of the piece permutation, disregarding piece rotations

Requirements:

- move sequence
- move piece permutations (the moves only describing piece movement without rotation)
- piece sets

> 1. determine the piece permutation as in the second approach to problem 2.
> 2. calculate the order of the resulting permutation

---

4. order of the piece rotations, disregarding piece movements

Requirements:

- move sequence
- move piece permutations (the moves only describing piece movement without rotation)
- piece lists

  - this requires sorted pieces!
    sorting of the points is not tested yet but is implemented in `state_validation_v2.py`

This may be impossible to calculate if the pieces are moved because different positions of the same piece type can be saved in a different order.

If the pieces are not moved but the puzzle is still changed, proceed with the following steps:

> 1. Determine the piece permutation as in the second approach to problem 2.
> 2. For each affected piece find how much the piece is rotated.
>   This can be done very efficiently by checking either
>   1. the position of the *image of the first element* of the piece list within the piece list.
>   2. the position of the first element of the piece list within the *image of the piece list*.

Option 2.1 should be slightly more efficient as it only needs to calculate one image.

Example:

Let the permutation of a piece be: `[1,3,7,5]` -> `[7,5,1,3]`. The image of `1` is `7`, `7` has index 2 in the orignal piece list, therfor the rotation has value 2.
In option 2.1 we would calculate the image of the whole piece and then search for the index of the `1`.

---

5. whether or not pieces are moved, rotated or both

> 1. determine the point permutation by concatenating the moves if that's a scrambled state:
> 2. determine the piece permutation as in the second approach to problem 2. Then there are a few different cases:
>   a) $A \neq id \land B \neq id \iff$ pieces moved and maybe rotated too,
>   b) $A \neq id\land \lnot B = id \iff$ pieces only rotated, not moved

For now the I am unable to determine the rotation of pieces if they are moved.

---

6. types of affected pieces

We can categorize the puzzle pieces by counting how many points they include. Pieces with different numbers of points cannot be interchanged.

Therefor an algorithm that only affects one type of piece is often more desireable than one that affects multiple types, because it allows for greater flexibility, solving each type of piece independently from the rest.