Algorithm Generation

One of the core contributions of this work is the automatic generation of algorithms for twisty puzzles. We define algorithms as sequences of base moves, that fulfill certain criteria:

- they affect only *few* puzzle pieces (meaning of *few* depends on the puzzle, more on that later)
- they have low order (usually < 6)
- they are not too long (< ~200 moves)
- they are reasonably easy to memorize for humans (< ~20 moves before repeating a pattern)

How to generate algorithms

1. generate move sequences with a few base moves
2. find useful numbers of repetitions for these base move sequences, exploiting the permutation group structure of the puzzle
3. filter out move sequences that are too long or affect too many pieces
   What remains are potentially useful algorithms.
4. filter out overlap with previously generated algorithms or refine the existing ones
   We don't want to generate the same algorithm twice, as it would be a waste of resources to train an RL agent to use the same algorithm in two different ways.
5. reapeat 1-4 or stop generating new algorithms

1. Generate move sequences

To generate the base sequences, we utilize the `smart_scramble` function: This avoids

- sequences passing through the same state twice
- reduces number of moves by replacing moves with their inverses when beneficial

2. Find useful numbers of repetitions

Given a base sequence s with cycles c_i, we are looking for numbers of repetitions $k \in \mathbb{N}$ such that s^k affects fewer or the same number of pieces as s. To achieve this, k must be a multiple of a cylce's order $ord(c_i)$ for at least one cycle. Then the order of s^k is $ord(s)/k$ and any points that were affected by c_i will remain unaffected by s^k.

We also know that $k\leq ord(s)/2$ must hold, since any larger k would not divide $ord(s)$, leading to unnecessarily long algorithms. Due to the finite order of s, it holds that $s^k = s^{ord(s)-k} ,, \forall k \in \mathbb{N}$. Therefore any $k > ord(s)/2$ can be replaced by $ord(s)-k$ without changing the resulting permutation.

This way, we can efficiently find promising algorithm candidates and predict their effect on the puzzle. We could even specifically search for algorithms affecting given pieces, by considering repetitions of the base sequence that keep other pieces fixed.

Given a list of points that we want to affect with an algorithm and a base sequence s, we could find all cycles c_j of s that affect other points (let I be the set of these indices). Repeating s $lcm({ord(c_j) | j \in I})$ times, ensures that only the desired points are affected by the resulting algorithm. When searching for

algorithms affecting specific points, we can also limit the search space for base sequences to include at least one move m affecting each of the desired points, and at least one other move m' that intersects with m. Without these conditions, the resulting algorithms would never affect the desired points in a meaningful way.

3. Filter out long or too complex algorithms

We want to avoid generating algorithms that are too long as they would be difficult for humans to execute reliably on physical puzzles.

We also want to limit the number of pieces affected by an algorithm, to make it easier for humans to predict their effect and for RL agents to get dense rewards. Algorithms that affect many pieces can cause large jumps in the agent's reward, which can lead to deep local optima, which inhibit learning (demonstrated by greedy solver getting stuck).

After this step, we consider any remaining repeated move sequences as useful algorithms. To build up a diverse, useful set of algorithms with which to solve the puzzle, we

4. Filter out overlap with existing algorithms

Finally, we want to avoid generating the same or very similar algorithms multiple times. This would be confusing for humans and slow down learning for RL agents by bloating the action set.

Given a new algorithm a, a set A of existing ones and a set R of spatial rotations of the entire puzzle, we consider a few cases:

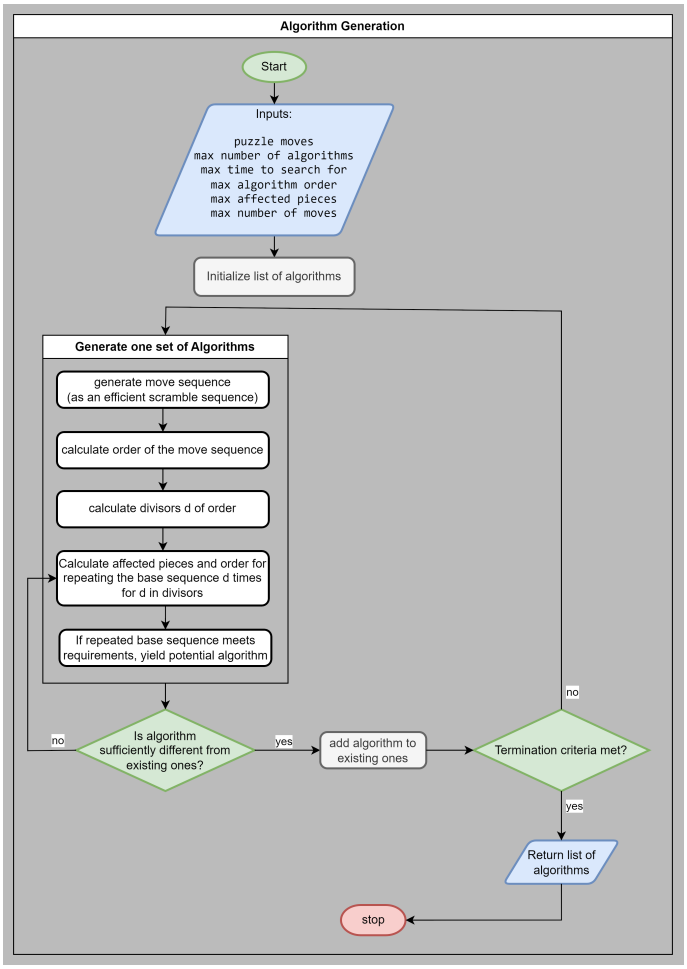
1. Does a have the same permutation as any $a' \in A$ or any spatial rotation $r \circ a' \circ r'$ with $a' \in A, r, r' \in R$?
    - If yes, compare the number of moves required by both algorithms. Keep the shorter one, discard the longer one.
      If they are the same length, we could decide based on which base moves are used, or the length of the base sequence (analogous to the number of repetitions).
      Afterwards, we can move on to generating new algorithms or stop generation.
2. if a or any rotation of it are not in A, we can check if algorithms in A are similar to a, by investigating the order as well as number and types of affected pieces. For this purpose, we propose an algorithm signature, that can be used quickly tell algorithms apart or detect similarity. (see `src/algorithm_generation/algorithm_analysis.py` and `algorithm_signature_planning.pdf`. For an outdated earlier version, see `algorithm_analysis.md`).

See also: `algorithm_filtering.md`

5. Repeat or stop generating new algorithms

Deciding when to stop generating new algorithms is a difficult problem. Experiments with the greedy solver have shown that having only some algorithms available

Algorithm Generation Flowchart



Idea for advanced algorithm generation

Part 2 of this method describes how to find algorithms targetting specific points. This could be used to dynamically find algorithms solving the puzzle in steps.

At first, find any algorithm. Then, over many scrambles and greedy solves optimizing the reward function, we can differentiate pieces that can be solved with the exisitng algorihtms from those that can't. We can then generate new algorithms targetting only the unsolved pieces. This prevents making the action set for RL agents unnecessarily large.

Positive effects of this approach

- This allows enables solutions to follow a common strategy used by humans: first solve a few pieces while ignoring the rest, then solve the rest in similar steps, ignoring some others. This can be useful, as it simplifies the algorithms needed to solve the puzzle. Example: When we start our solution by solving the corners of a Rubik's cube, we don't have to avoid affecting the edges. We only try to avoid undoing existing progress.

(image taken from cubelelo.com, *cropped to remove text*)

- This would solve the problem of having multiple algorithms of different orders affecting the same or similar pieces.
- This technique could provide a good indication when to stop searching for new algorithms: If the greedy solver can solve a high proportion of states, we can assume that the action set is sufficient for the RL agent to efficiently learn a good policy.