The Journal of Open Source Software

maze-dataset

Michael Igorevich Ivanitskiy [1¶], **Aaron Sandoval** [1], **Alex F. Spies** [2], **Rusheb Shah** [1], **Brandon Knutson** [1], [1], **Cecilia Diniz Behn** [1], and **Samy Wu Fung** [1]

1 Colorado School of Mines, Department of Applied Mathematics and Statistics **2** Imperial College London ¶ Corresponding author

DOI: N/A

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: 01 January 1970

Summary

Solving mazes is a classic problem in computer science and artificial intelligence, and humans have been constructing mazes for thousands of years. Although finding the shortest path through a maze is a solved problem, this very fact makes it an excellent testbed for studying how machine learning algorithms solve problems and represent spatial information. We introduce `maze-dataset`, a Python library for generating, processing, and visualizing datasets of mazes. This library supports a variety of maze generation algorithms providing both mazes with loops and "perfect" mazes without them. These generation algorithms can be configured with various parameters, and the resulting mazes can be filtered to satisfy desired properties. Also provided are tools for converting mazes to and from various formats suitable for a variety of neural network architectures, such as rasterized images and tokenized text sequences, as well as various visualization tools. As well as providing a simple interface for generating, storing, and loading these datasets, `maze-dataset` is extensively tested, type hinted, benchmarked, and documented.

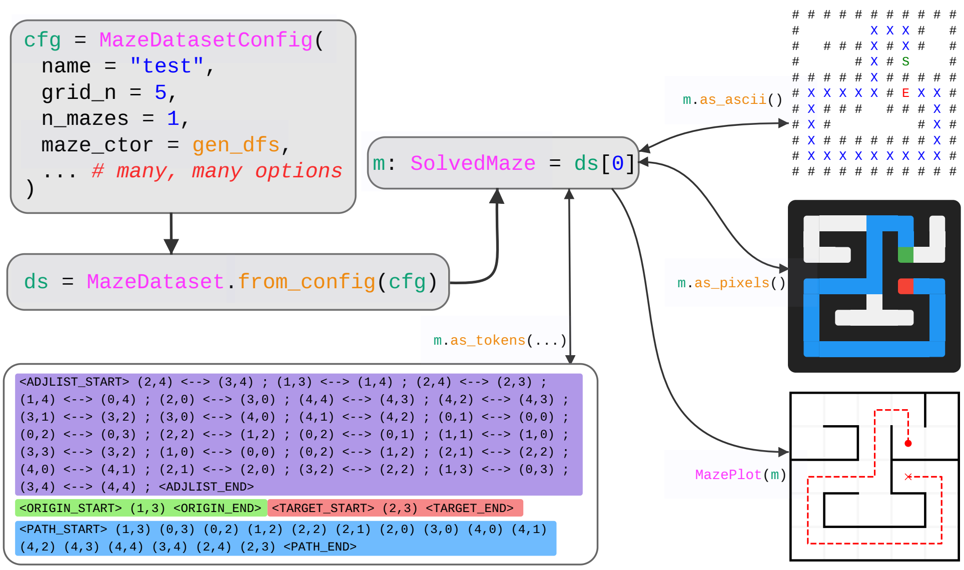

Figure 1: Usage of maze-dataset. We create a 'MazeDataset' from a 'MazeDatasetConfig'. This contains 'SolvedMaze' objects which can be converted to and from a variety of formats. Code in the image contains clickable links to documentation. A variety of generated examples can be viewed here.

Statement of Need

The generation of mazes with a given algorithm is not inherently a complex task, but the ability to seamlessly switch out algorithms, modify algorithm parameters, or filter by desired properties all while preserving the ability to convert between different representations of the maze is not trivial. This library aims to greatly streamline the process of generating and working with datasets of mazes that can be described as subgraphs of an $n \times n$ lattice with boolean connections and, optionally, start and end points that are nodes in the graph. Furthermore, we place emphasis on a wide variety of possible text output formats aimed at evaluating the spatial reasoning capabilities of Large Language Models and other text-based transformer models.

For interpretability and behavioral research, algorithmic tasks offer benefits by allowing systematic data generation and task decomposition, as well as simplifying the process of circuit discovery (Räuker et al., 2023). Although mazes are well suited for these investigations, we have found that existing maze generation packages (Cobbe et al., 2019; Ehsan, 2022; Harries et al., n.d.; Németh, 2019; Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021) do not support flexible maze generation algorithms that provide fine-grained control of generation parameters and the ability to easily transform between multiple representations of the mazes (Images, Textual, Tokenized) for training and testing models.

Related Works

A multitude of public and open-source software packages exist for generating mazes (Ehsan, 2022; Németh, 2019; Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021). However, nearly all of these packages generate and store mazes in a form that is not optimized for storage space or, more importantly, computer readability. The mazes produces by other packages are usually rasterized or in some form of image, rather than the underlying graph structure, and this makes it difficult to work with these datasets.

- Most prior works provide mazes in some kind of image or raster format, which is not suitable for training autoregressive text-based transformer models – a key usage case this work seeks to enable. However, we still provide a variety of similar output formats:
  - we also include the `RasterizedMazeDataset` class, utilizing `as_pixels()`, in our codebase, which can exactly mimic the outputs provided in `easy-to-hard-data`(Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021) and can be configured to be similar to the outputs of (Németh, 2019).
  - Our `as_ascii()` method provides a format similar to that used in Oppenheim (2018).
  - Our `MazePlot` class provides a feature-rich plotting utility with support for multiple paths, heatmaps over positions, and more. This is similar to the outputs of Ehsan (2022)
- The text format provided by `SolvedMaze(...).as_tokens()` is similar to that of (Liu & Wu, 2023), but provides over 5.8 million unique formats for converting mazes to a text stream. We maintain a single underlying format, meaning that the same maze can be turned into a variety of text streams to assess how the precise format of the text stream affects the model.
- For rigorous investigations of the response of a model to various distributional shifts, preserving metadata about the generation algorithm with the dataset itself is essential. To this end, our package efficiently stores the dataset along with its metadata in a single human-readable file (M. Ivanitskiy, n.d.). This metadata is loaded when the dataset is retrieved from disk and makes it simple to understand how exactly each maze was generated. As far as we are aware, no existing packages do this reliably.
- Storing mazes as images is not only difficult to work with, but also inefficient. Directly storing adjacency matrices is also inefficient as subgraphs of the lattice are sparse. Storing

adjacency lists can be efficient, but comes with a higher lookup cost and possible high comparison cost. We use a simple, efficient representation of mazes that is optimized for subgraphs of a d-dimensional finite lattice that we do not believe is used in any existing maze generation package.

- Our package is easily installable with source code freely available. It is extensively tested, type hinted, benchmarked, and documented. Many other maze generation packages lack this level of rigor and scope, and some (Ayaz et al., 2008) appear to simply no longer be accessible.

Features

Generation and Usage

Our package can be installed from PyPi via `pip install maze-dataset`, or directly from the git repository (Michael I. Ivanitskiy et al., 2023a).

To create a dataset, we first create a `MazeDatasetConfig` configuration object, which specifies the seed, number, and size of mazes, as well as the generation algorithm and its corresponding parameters. This object is passed to a `MazeDataset` class to create a dataset. Crucially, this `MazeDataset` mimics the interface of a PyTorch (Paszke et al., 2019) `Dataset`, and can thus be easily incorporated into existing data pre-processing and training pipelines, e.g., through the use of a `Dataloader` class.

```
from maze_dataset import MazeDataset, MazeDatasetConfig, LatticeMazeGenerators
cfg: MazeDatasetConfig = MazeDatasetConfig(
    name="example",
    grid_n=3,
    n_mazes=32,
    maze_ctor=LatticeMazeGenerators.gen_dfs,
)
dataset: MazeDataset = MazeDataset.from_config(cfg)
```

When initializing mazes, further configuration options can be specified through the `from_config()` factory method as necessary. Options allow for saving/loading existing datasets instead of regenerating, and parallelization options for generation. Available maze generation algorithms are static methods of the `LatticeMazeGenerators` class and include the following:

- `gen_dfs` **(randomized depth-first search)**: Parameters can be passed to constrain the number of accessible cells, the number of forks in the maze, and the maximum tree depth. Creates a spanning tree by default or a partially spanning tree if constrained.
- `gen_wilson` **(Wilson's algorithm)**: Generates a random spanning tree via loop-erased random walk (Wilson, 1996).
- `gen_percolation` **(percolation)**: Starting with no connections, every possible lattice connection is set to either true or false with some probability p, independently of all other connections. For the kinds of graphs that this process generates, we refer to existing work (Duminil-Copin, 2017; Fisher & Essam, 2004).
- `gen_dfs_percolation` **(randomized depth-first search with percolation)**: A connection exists if it exists in a maze generated via `gen_dfs OR gen_percolation`. Useful for generating mazes that are not acyclic graphs.

Furthermore, a dataset of mazes can be filtered to satisfy certain properties:

```
dataset_filtered: MazeDataset = dataset.filter_by.path_length(min_length=3)
```

Custom filters can be specified, and several filters are included:

- `path_length(min_length: int)`: shortest length from the origin to target should be at least `min_length`.
- `start_end_distance(min_distance: int)`: Manhattan distance between start and end should be at least `min_distance`, ignoring walls.
- `remove_duplicates(...)`: remove mazes which are similar to others in the dataset, measured via Hamming distance.
- `remove_duplicates_fast()`: remove mazes which are exactly identical to others in the dataset.

All implemented maze generation algorithms are stochastic by nature. For reproducibility, the `seed` parameter of `MazeDatasetConfig` may be set. In practice, we do not find that exact duplicates of mazes are generated with any meaningful frequency, even when generating large datasets.

Visual Output Formats

Internally, mazes are `SolvedMaze` objects, which have path information, and a connection list optimized for storing sub-graphs of a lattice. These objects can be converted to and from several formats.

`as_ascii()`	`as_pixels()`	`MazePlot()`
Simple text format for displaying mazes, useful for debugging in a terminal environment.	`numpy` array of `dtype=uint8` and shape `(height, width, 3)`. The last dimension is RGB color.	feature-rich plotting utility with support for multiple paths, heatmaps over positions, and more.

docs/paper/figures/output-fmts.pdf

Figure 2: Various output formats. Top row (left to right): ASCII diagram, rasterized pixel grid, and advanced display.

Visual Outputs for Training and Evaluation

In previous work, maze tasks have been used with Recurrent Convolutional Neural Network (RCNN) derived architectures (Schwarzschild, Borgnia, Gupta, Huang, et al., 2021). To facilitate the use of our package in this context, we replicate the format of (Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021) and provide the `RasterizedMazeDataset` class which returns rasterized pairs of (input, target) mazes as shown in **??** below.

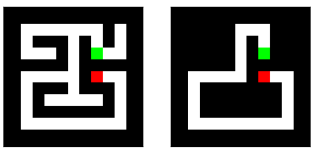

Figure 3: Input is the rasterized maze without the path marked (left), and provide as a target the maze with all but the correct path removed. Configuration options exist to adjust whether endpoints are included and if empty cells should be filled in.

The Journal of Open Source Software

Tokenized Output Formats

To train autoregressive text models such as transformers, we convert mazes to token sequences in two steps. First, the maze is stringified using `as_tokens()`. The `MazeTokenizerModular` class provides a powerful interface for configuring maze stringification behavior. Second, the sequence of strings is tokenized into integers using `encode()`. Tokenization uses a fixed vocabulary for simplicity. Mazes up to 50x50 are supported using unique tokens, and up to 128x128 when using coordinate tuple tokens.

Stringification Options and `MazeTokenizerModular`

There are many algorithms by which one might tokenize a 2D maze into a 1D format usable by autoregressive text models. Training multiple models on the encodings output from each of these algorithms may produce very different internal representations, learned solution algorithms, and levels of performance. To explore how different maze tokenization algorithms affect these models, the `MazeTokenizerModular` class contains a rich set of options to customize how mazes are stringified. This class contains 19 discrete parameters, resulting in 5.9 million unique tokenizers. But wait, there's more! There are 6 additional parameters available in the library which are untested but further expand the the number of tokenizers by a factor of $44/3$ to 86 million.

All output sequences consist of four token regions representing different features of the maze. These regions are distinguished by color in Figure below.

- Adjacency list: A text representation of the lattice graph
- Origin: Starting coordinate
- Target: Ending coordinate
- Path: Maze solution sequence from the start to the end

Figure 4: Example text output format with token regions highlighted.

Each `MazeTokenizerModular` is constructed from a set of several `_TokenizerElement` objects, each of which specifies how different token regions or other elements of the stringification are produced.

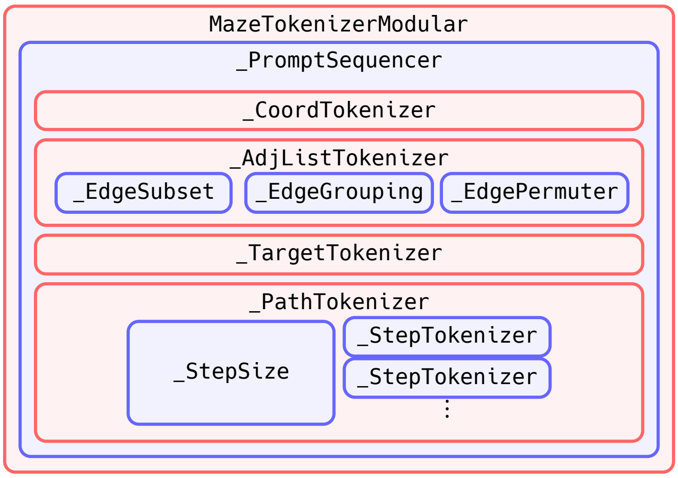

Figure 5: Nested internal structure of `_TokenizerElement` objects inside a typical `MazeTokenizerModular` object.

Optional delimiter tokens may be added in many places in the output. Delimiter options are all configured using the parameters named `pre`, `intra`, and `post` in various `_TokenizerElement` classes. Each option controls a unique delimiter token. Here we describe each `_TokenizerElement` and the behaviors they support. We also discuss some of the model behaviors and properties that may be investigated using these options.

Coordinates

The `_CoordTokenizer` object controls how coordinates in the lattice are represented in across all token regions. Options include:

- **Unique tokens**: Each coordinate is represented as a single unique token `"(i,j)"`
- **Coordinate tuple tokens**: Each coordinate is represented as a sequence of 2 tokens, respectively encoding the row and column positions: `["i", ",", "j"]`

Adjacency List

The `_AdjListTokenizer` object controls this token region. All tokenizations represent the maze connectivity as a sequence of connections or walls between pairs of adjacent coordinates in the lattice.

- `_EdgeSubset`: Specifies the subset of lattice edges to be tokenized
  - **All edges**: Every edge in the lattice
  - **Connections**: Only edges which contain a connection
  - **Walls**: Only edges which contain a wall
- `_EdgePermuter`: Specifies how to sequence the two coordinates in each lattice edge
  - **Random**
  - **Sorted**: The smaller coordinate always comes first
  - **Both permutations**: Each edge is represented twice, once with each permutation. This option attempts to represent connections in a more directionally symmetric manner. Including only one permutation of each edge may affect models' internal representations of edges, treating a path traversing the edge differently depending on if the coordinate sequence in the path matches the sequence in the adjacency list.
- `shuffle_d0`: Whether to shuffle the edges randomly or sort them in the output by their first coordinate
- `connection_token_ordinal`: Location in the sequence of the token representing whether the edge is a connection or a wall

Path

The `_PathTokenizer` object controls this token region. Paths are all represented as a sequence of steps moving from the start to the end position.

- `_StepSize`: Specifies the size of each step
  - **Singles**: Every coordinate traversed between start and end is directly represented
  - **Forks**: Only coordinates at forking points in the maze are represented. The paths between forking points are implicit. Using this option might train models more directly to represent forking points differently from coordinates where the maze connectivity implies an obvious next step in the path.
- `_StepTokenizer`: Specifies how an individual step is represented
  - **Coordinate**: The coordinates of each step are directly tokenized using a `_CoordTokenizer`
  - **Cardinal direction**: A single token corresponding to the cardinal direction taken at the starting position of that step. E.g., `NORTH`, `SOUTH`. If using a `_StepSize` other than **Singles**, this direction may not correspond to the final direction traveled to arrive at the end position of the step.

- **Relative direction**: A single token corresponding to the first-person perspective relative direction taken at the starting position of that step. E.g., `RIGHT`, `LEFT`.
- **Distance**: A single token corresponding to the number of coordinate positions traversed in that step. E.g., using a `_StepSize` of **Singles**, the **Distance** token would be the same for each step, corresponding to a distance of 1 coordinate. This option is only of interest in combination with a `_StepSize` other than **Singles**.

A `_PathTokenizer` contains a sequence of one or more unique `_StepTokenizer` objects. Different step representations may be mixed and permuted, allowing for investigation of model representations of multiple aspects of a maze solution at once.

Tokenized Outputs for Training and Evaluation

During deployment we provide only the prompt up to the `<PATH_START>` token.

Examples of usage of this dataset to train autoregressive transformers can be found in our `maze-transformer` library (Michael I. Ivanitskiy et al., 2023b). Other tokenization and vocabulary schemes are also included, such as representing each coordinate as a pair of i, j index tokens.

Extensibility

The tokenizer architecture is purposefully designed such that adding and testing a wide variety of new tokenization algorithms is fast and minimizes disturbances to functioning code. This is enabled by the modular architecture and the automatic inclusion of any new tokenizers in integration tests. To create a new tokenizer, developers forking the library may simply create their own `_TokenizerElement` subclass and implement the abstract methods. If the behavior change is sufficiently small, simply adding a parameter to an existing `_TokenizerElement` subclass and updating its implementation will suffice. For small additions, simply adding new cases to existing unit tests will suffice.

The breadth of tokenizers is also easily scaled in the opposite direction. Due to the exponential scaling of parameter combinations, adding a small number of new features can significantly slow certain procedures which rely on constructing all possible tokenizers, such as integration tests. If any existing subclass contains features which aren't needed, a developer tool decorator is provided which can be applied to the unneeded `_TokenizerElement` subclasses to prune those features and compact the available space of tokenizers.

Benchmarks of Generation Speed

We provide approximate benchmarks for relative generation time across various algorithms, parameter choices, maze sizes, and dataset sizes.

Method & Parameters	Average time per maze (ms)				
Generation algorithm	Generation parameters	all sizes	small ($g \leq 10$)	medium ($10 < g \leq 32$)	large ($g > 32$)
gen_dfs	accessible_cells=20	2.4	2.4	2.6	2.4
gen_dfs	do_forks=False	3.0	2.4	3.7	3.8
gen_dfs	max_tree_depth=0.5	4.5	2.2	4.9	11.6
gen_dfs	–	31.1	2.8	28.0	136.5
gen_dfs_percolation	p=0.1	53.9	3.6	42.5	252.9

The Journal of Open Source Software

Method & Parameters	Average time per maze (ms)				
gen_dfs_percolation	p=0.4	58.8	3.7	44.7	280.2
gen_percolation	–	59.1	3.3	43.6	285.2
gen_wilson	–	767.9	10.1	212.9	4530.4
:========	=========	=======	=======	==========	======================
median (all runs)		10.8	6.0	44.4	367.7
mean (all runs)		490.0	11.7	187.2	2769.6

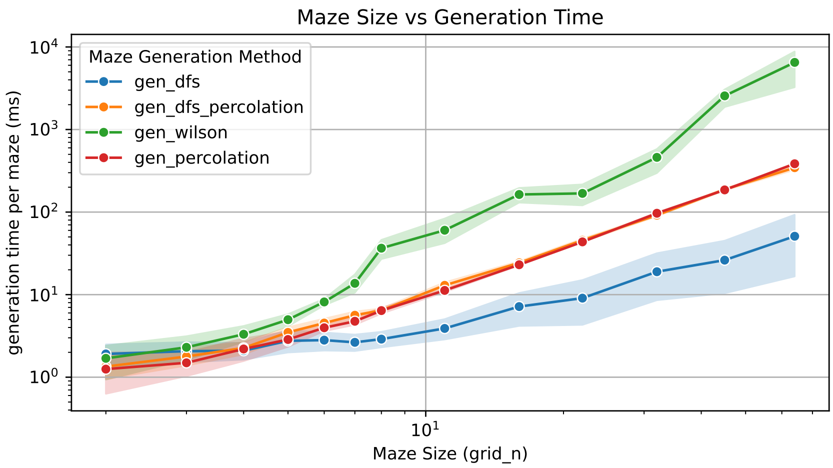

Figure 6: Plots of maze generation time. Generation time scales exponentially with maze size for all algorithms (left). Generation time does not depend on the number of mazes being generated, and there is minimal overhead to initializing the generation process for a small dataset (right). Wilson's algorithm is notably less efficient than others and has high variance. Note that for both plots, values are averaged across all parameter sets for that algorithm, and parallelization is disabled.

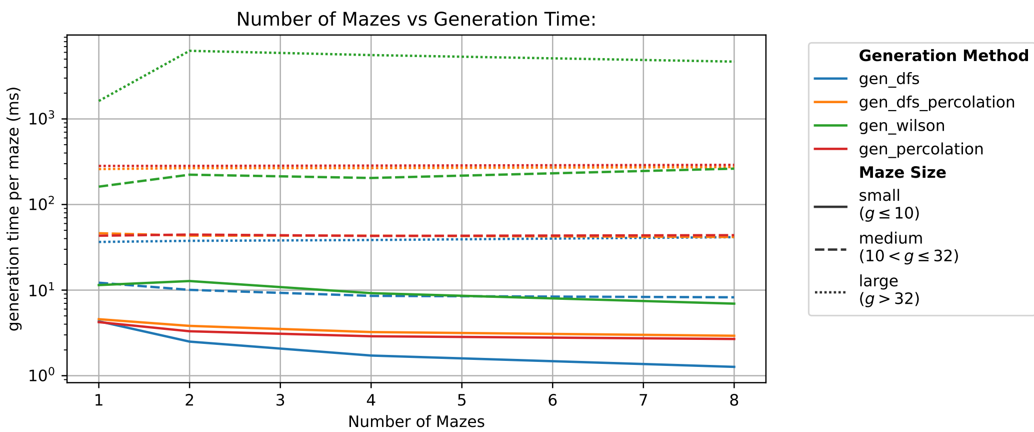

Figure 7: Maze size vs generation time

Implementation

We refer to our GitHub repository (Michael I. Ivanitskiy et al., 2023a) for documentation and up-to-date implementation details.

This package utilizes a simple, efficient representation of mazes. Using an adjacency list to represent mazes would lead to a poor lookup time of whether any given connection exists, whilst using a dense adjacency matrix would waste memory by failing to exploit the structure (e.g., only 4 of the diagonals would be filled in). Instead, we describe mazes with the following simple representation: for a d-dimensional lattice with r rows and c columns, we initialize a boolean array $A = \{0, 1\}^{d \times r \times c}$, which we refer to in the code as a `connection_list`. The value at $A[0, i, j]$ determines whether a downward connection exists from node $[i, j]$ to $[i+1, j]$. Likewise, the value at $A[1, i, j]$ determines whether a rightwards connection to $[i, j+1]$ exists. Thus, we avoid duplication of data about the existence of connections, at the cost of requiring additional care with indexing when looking for a connection upwards or to the left. Note that this setup allows for a periodic lattice.

To produce solutions to mazes, two points are selected uniformly at random without replacement from the connected component of the maze, and the A^* algorithm (Hart et al., 1968) is applied to find the shortest path between them.

Parallelization is implemented via the `multiprocessing` module in the Python standard library, and parallel generation can be controlled via keyword arguments to the `MazeDataset.from_config()` function.

Limitations of `maze-dataset`

For simplicity, the package primarily supports mazes that are sub-graphs of a 2-dimensional rectangular lattice. Some support for higher-dimensional lattices is present, but not all output formats are adapted for higher dimensional mazes. Implementation Implementation

Usage in Research

This package was originally built for the needs of the (Michael I. Ivanitskiy et al., 2023b) project, which aims to investigate spatial planning and world models in autoregressive transformer models trained on mazes Spies et al. (2024). This project has also adapted itself to be useful for work on understanding the mechanisms by which recurrent convolutional and implicit networks (Fung et al., 2022) solve mazes given a rasterized view (Knutson et al., 2024), and for this we match the output format of (Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021).

This package has also been utilized in work by other groups:

- (Nolte et al., 2024) use `maze-dataset` to compare the effectiveness of transformers trained with the MLM-$\mathcal{U}$ (Kitouni et al., 2024) multistep prediction objective against standard autoregressive training for multi-step planning on our maze task.
- (Wang et al., 2024) and (Chen et al., 2024) use `maze-dataset` to study the effectiveness of imperative learning

Conclusion

The `maze-dataset` library (Michael I. Ivanitskiy et al., 2023a) introduced in this paper provides a flexible and extensible toolkit for generating, processing, and analyzing maze datasets. By supporting various procedural generation algorithms and conversion utilities, it enables the creation of mazes with customizable properties to suit diverse research needs. Planned improvements to the `maze-dataset` include adding more generation algorithms (such as Prim's algorithm (Dijkstra, 1959; Jarnık, 1930; Prim, 1957) and Kruskal's algorithm (Kruskal, 1956), among others (Gabrovšek, 2019)), adding the ability to augment a maze with an adjacency list to add "shortcuts" to the maze, and resolving certain limitations detailed in Section Limitations.

Acknowledgements

This work was partially supported by and many of the authors were brought together by AI Safety Camp and AI Safety Support. This work was partially funded by National Science Foundation awards DMS-2110745 and DMS-2309810. We are also grateful to LTFF and FAR Labs for hosting three of the authors for a Residency Visit, and to various members of FAR's technical staff for their advice. We thank the Mines Optimization and Deep Learning group (MODL) for fruitful discussions. We also thank Michael Rosenberg for recommending the usage of Finite State Transducers for storing tokenizer validation information.

References

Alance AB. (2019). *Maze generator*. http://www.mazegenerator.net. Ayaz, H., Allen, S. L., Platek, S. M., & Onaral, B. (2008). Maze suite 1.0: A complete set of tools to prepare, present, and analyze navigational and spatial cognitive neuroscience experiments. *Behavior Research Methods*, *40*, 353–359. Chen, X., Yang, F., & Wang, C. (2024). iA $\hat{}*$: Imperative learning-based a $\hat{}*$ search for pathfinding. *arXiv Preprint arXiv:2403.15870*. Cobbe, K., Hesse, C., Hilton, J., & Schulman, J. (2019). Leveraging procedural generation to benchmark reinforcement learning. *arXiv Preprint arXiv:1912.01588*. Dijkstra, E. W. (1959). *A note on two problems in connexion with graphs:(Numerische Mathematik, 1 (1959), p 269-271)*. Duminil-Copin, H. (2017). *Sixty years of percolation* (No. arXiv:1712.04651). arXiv. http://arxiv.org/abs/1712.04651 Ehsan, E. (2022). *Maze*. https://github.com/emadehsan/maze Fisher, M. E., & Essam, J. W. (2004). Some Cluster Size and Percolation Problems. *Journal of Mathematical Physics*, *2*(4), 609–619. https://doi.org/10.1063/1.1703745 Fung, S. W., Heaton, H., Li, Q., McKenzie, D., Osher, S., & Yin, W. (2022). Jfb: Jacobian-free backpropagation for implicit networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, *36*, 6648–6656. Gabrovšek. (2019). Analysis of maze generating algorithms. *IPSI Transactions on Internet Research*, *15.1*, 23–30. http://www.ipsitransactions.org/journals/papers/tir/2019jan/p5.pdf Guo, C., Barthelet, L., & Morris, R. (2011). *Maze generator and solver*. Wolfram Demonstrations Project, https://demonstrations.wolfram.com/MazeGeneratorAndSolver/. Harries, L., Lee, S., Rzepecki, J., Hofmann, K., & Devlin, S. (n.d.). MazeExplorer: A Customisable 3D Benchmark for Assessing Generalisation in Reinforcement Learning. *2019 IEEE Conf. Games CoG*, 1–4. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136 Ivanitskiy, M. (n.d.). *ZANJ*. https://github.com/mivanit/ZANJ Ivanitskiy, Michael I., Shah, R., Spies, A. F., Räuker, T., Valentine, D., Rager, C., Quirke, L., Corlouer, G., & Mathwin, C. (2023a). *Maze dataset*. https://github.com/understanding-search/maze-dataset Ivanitskiy, Michael I., Shah, R., Spies, A. F., Räuker, T., Valentine, D., Rager, C., Quirke, L., Corlouer, G., & Mathwin, C. (2023b). *Maze transformer interpretability*. https://github.com/understanding-search/maze-transformer Ivanitskiy, Michael Igorevich, Spies, A. F., Räuker, T., Corlouer, G., Mathwin, C., Quirke, L., Rager, C., Shah, R., Valentine, D., Behn, C. D., & others. (2023). Structured world representations in maze-solving transformers. *arXiv Preprint arXiv:2312.02566*. Jarnık, V. (1930). About a certain minimal problem. *Práce Moravské Prırodovedecké Spolecnosti*, *6*, 57–63. Kitouni, O., Nolte, N. S., Williams, A., Rabbat, M., Bouchacourt, D., & Ibrahim, M. (2024). The factorization curse: Which tokens you predict underlie the reversal curse and more. *Advances in Neural Information Processing Systems*, *37*, 112329–112355. Knutson, B., Rabeendran, A. C., Ivanitskiy, M., Pettyjohn,

J., Diniz-Behn, C., Fung, S. W., & McKenzie, D. (2024). On logical extrapolation for mazes with recurrent and implicit networks. *arXiv Preprint arXiv:2410.03020*. Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, *7*(1), 48–50. https://doi.org/10.1090/S0002-9939-1956-0078686-7 Liu, C., & Wu, B. (2023). Evaluating large language models on graphs: Performance insights and comparative analysis. *arXiv Preprint arXiv:2308.11224*. Nag, A. (2020). MDL suite: A language, generator and compiler for describing mazes. *Journal of Open Source Software*, *5*(46), 1815. Németh, F. (2019). *Maze-generation-algorithms*. https://github.com/ferenc-nemeth/maze-generation-algorithms Nolte, N., Kitouni, O., Williams, A., Rabbat, M., & Ibrahim, M. (2024). Transformers can navigate mazes with multi-step prediction. *arXiv Preprint arXiv:2412.05117*. Oppenheim, J. (2018). *Maze-generator: Generate a random maze represented as a 2D array using depth-first search*. https://github.com/oppenheimj/maze-generator/; GitHub. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, *36*(6), 1389–1401. Räuker, T., Ho, A., Casper, S., & Hadfield-Menell, D. (2023). Toward transparent ai: A survey on interpreting the inner structures of deep neural networks. *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 464–483. Schwarzschild, A., Borgnia, E., Gupta, A., Bansal, A., Emam, Z., Huang, F., Goldblum, M., & Goldstein, T. (2021). *Datasets for Studying Generalization from Easy to Hard Examples* (No. arXiv:2108.06011). arXiv. https://doi.org/10.48550/arXiv.2108.06011 Schwarzschild, A., Borgnia, E., Gupta, A., Huang, F., Vishkin, U., Goldblum, M., & Goldstein, T. (2021). Can you learn an algorithm? Generalizing from easy to hard problems with recurrent networks. *Advances in Neural Information Processing Systems*, *34*, 6695–6706. Singla, A. (2023). Evaluating ChatGPT and GPT-4 for visual programming. *arXiv Preprint arXiv:2308.02522*. Spies, A. F., Edwards, W., Ivanitskiy, M. I., Skapars, A., Räuker, T., Inoue, K., Russo, A., & Shanahan, M. (2024). Transformers use causal world models in maze-solving tasks. *arXiv Preprint arXiv:2412.11867*. Wang, C., Ji, K., Geng, J., Ren, Z., Fu, T., Yang, F., Guo, Y., He, H., Chen, X., Zhan, Z., & others. (2024). Imperative learning: A self-supervised neural-symbolic learning framework for robot autonomy. *arXiv Preprint arXiv:2406.16087*. Wilson, D. B. (1996). Generating random spanning trees more quickly than the cover time. *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing - STOC '96*, 296–303. https://doi.org/10.1145/237814.237880