![JOSS - The Journal of Open Source Software]

maze-dataset: Maze Generation with Algorithmic Variety and Representational Flexibility

Michael Igorevich Ivanitskiy [1¶], **Aaron Sandoval** [4], **Alex F. Spies** [2], **Tilman Räuker** [3], **Brandon Knutson** [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 **3** UnSearch.org **4** Independent **¶** Corresponding author

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 user-friendly Python library for generating, processing, and visualizing datasets of mazes. This library supports a variety of maze generation algorithms providing mazes with or without loops, mazes that are connected or not, and many other variations. 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, tokenized text sequences, and various visualizations. As well as providing a simple interface for generating, storing, and loading these datasets, `maze-dataset` is extensively tested, type hinted, benchmarked, and documented.
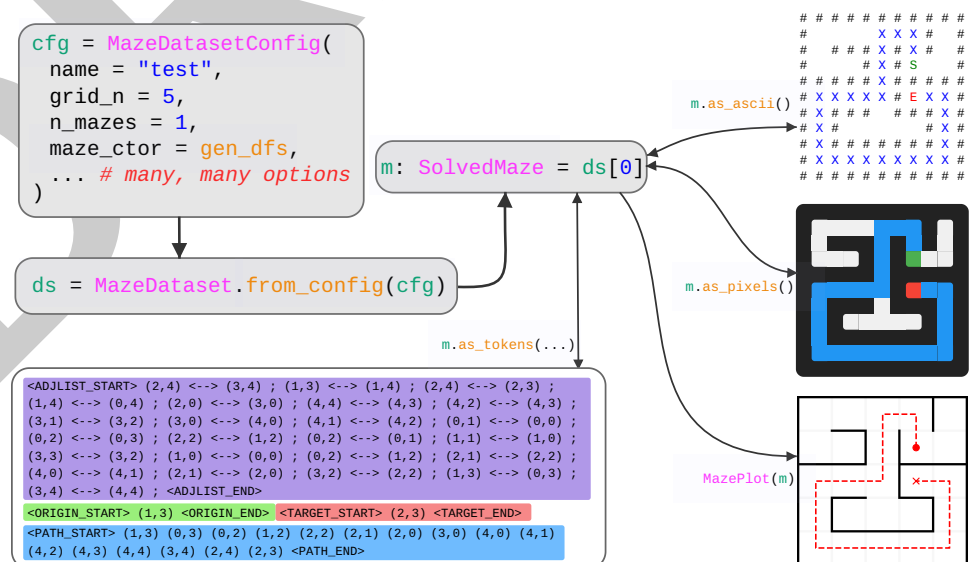
Figure 1: Usage of maze-dataset. We create a `MazeDataset` from a `MazeDatasetConfig`. This contains `SolvedMaze` objects which can be converted to and from a variety of formats. Code in the image contains clickable links to documentation. A variety of generated examples can be viewed here.

Statement of Need

While maze generation itself is straightforward, the architectural challenge comes from building a system supporting many algorithms with configurable parameters, property filtering, and representation transformation. 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 (LLMs) 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 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) lack support for transforming between multiple representations and provide limited control over the maze generation process.

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 produce mazes represented as rasterized images or other visual formats rather than the underlying graph structure, and this makes it difficult to work with these datasets.

- Most prior works provide mazes in visual or raster formats, and we provide a variety of similar output formats:

    - `RasterizedMazeDataset`, utilizing `as_pixels()`, 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)
    - `as_ascii()` provides a format similar to (Oppenheim, 2018; Singla, 2023)
    - `MazePlot` provides a feature-rich plotting utility with support for multiple paths, heatmaps over positions, and more. This is similar to the outputs of (Alance AB, 2019; Ehsan, 2022; Guo et al., 2011; Nag, 2020)

- 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, detailed in section: *Tokenized Output Formats*.

- 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.). 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. We use a highly efficient method detailed in section: *Implementation*.

- 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

We direct readers to our [examples](#), [docs](#), and [notebooks](#) for more information.

Generation and Basic 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.

```python
from maze_dataset import (
    MazeDataset, MazeDatasetConfig, LatticeMazeGenerators
)
# create a config
cfg: MazeDatasetConfig = MazeDatasetConfig(
    name="example", # names need not be unique
    grid_n=3,    # size of the maze
    n_mazes=32, # number of mazes in the dataset
    maze_ctor=LatticeMazeGenerators.gen_dfs, # many algorithms available
    # (optional) algorithm-specific parameters
    maze_ctor_kwargs={"do_forks": True, ...},
    # (optional) many options for restricting start/end points
    endpoint_kwargs={"deadend_start": True, ...},
)
# create a dataset
dataset: MazeDataset = MazeDataset.from_config(
    cfg, # pass the config
    ..., # other options for disk loading, parallelization, etc.
)
```

When initializing a dataset, options which do not affect the mazes themselves can be specified through the `from_config()` factory method as necessary. These options allow for saving/loading existing datasets instead of re-generating, parallelization options for generation, and more. Available maze generation algorithms are static methods of the `LatticeMazeGenerators` namespace class and include generation algorithms based on randomized depth-first search, Wilson's algorithm ([Wilson, 1996](#)), percolation ([Duminil-Copin, 2017](#); [Fisher & Essam, 2004](#)), Kruskal's algorithm ([Kruskal, 1956](#)), and others.

Furthermore, a dataset of mazes can be filtered to satisfy certain properties. Custom filters can be specified, and some filters are included in `MazeDatasetFilters`. For example, we can require a minimum path length of three steps from the origin to the target:

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

All implemented maze generation algorithms are stochastic by nature. For reproducibility, the seed parameter of `MazeDatasetConfig` may be set. In practice, using provided deduplication filters, we find that exact duplicate mazes are generated very infrequently, even when generating very large datasets.

For use cases where mazes of different sizes, generation algorithms, or other parameter variations are required, we provide the `MazeDatasetCollection` class, which allows for creating a single iterable dataset from multiple independent configurations.

Visual Output Formats

Internally, mazes are `SolvedMaze` objects, which have path information and a tensor optimized for storing sub-graphs of a lattice. These objects can be converted to and from several formats to maximize their utility in different contexts.

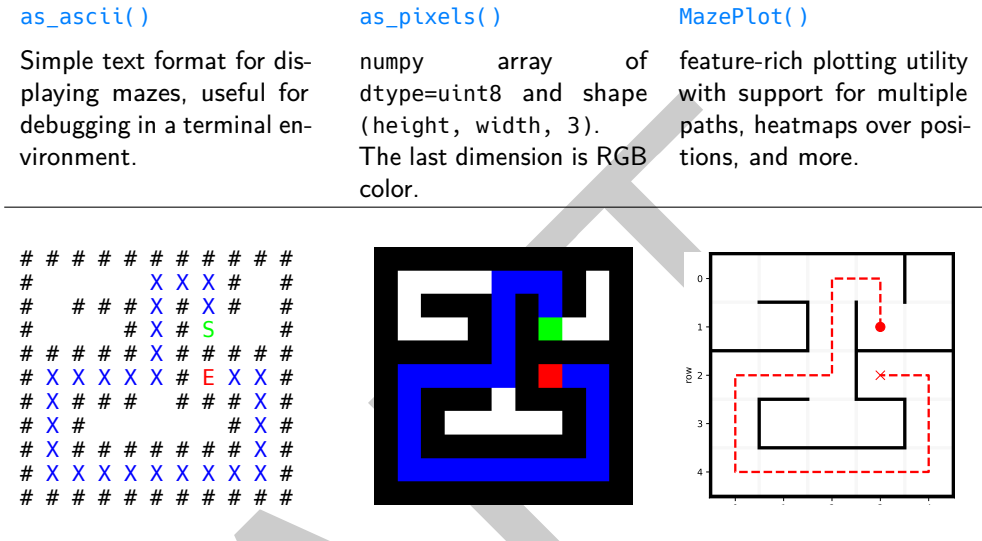| `as_ascii()`	`as_pixels()`	`MazePlot()`
Simple text format for displaying mazes, useful for debugging in a terminal environment.	numpy array of dtype=uint8 and shape (height, width, 3). The last dimension is RGB color.	feature-rich plotting utility with support for multiple paths, heatmaps over positions,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

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

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 Figure 3 below.
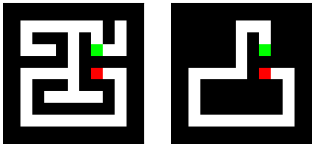

Figure 3: Input is the rasterized maze without the path marked (left), and provide as a target the maze with all but the correct path removed (right). Configuration options exist to adjust whether endpoints are included and if empty cells should be filled in.

Tokenized Output Formats

Autoregressive transformer models can be quite sensitive to the exact format of input data, and may even use delimiter tokens to perform reasoning steps (Pfau et al., 2024; Spies et al., 2024). To facilitate systematic investigation of the effects of different representations of data on text model performance, we provide a variety of tokenized text output formats.

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 50×50 are

supported when using a unique token for each position, and up to 128×128 are supported when positions in the maze are represented as a pair of coordinates.

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 allow exploration of 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 over 5.8 million unique tokenizers. There are 6 additional parameters available whose functionality is not verified via automated testing, 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; an example output sequence is shown in Figure 4.

```
<ADJLIST_START> (0,0) <--> (1,0) ; (2,0) <--> (3,0) ; (4,1) <--> (4,0) ; (2,0) <--> (2,1) ;
(1,0) <--> (1,1) ; (3,4) <--> (2,4) ; (4,2) <--> (4,3) ; (0,0) <--> (0,1) ; (0,3) <--> (0,2) ;
(4,4) <--> (3,4) ; (4,3) <--> (4,4) ; (4,1) <--> (4,2) ; (2,1) <--> (2,2) ; (1,4) <--> (0,4) ;
(1,2) <--> (0,2) ; (2,4) <--> (2,3) ; (4,0) <--> (3,0) ; (2,2) <--> (3,2) ; (1,2) <--> (2,2) ;
(1,3) <--> (0,3) ; (3,2) <--> (3,3) ; (0,2) <--> (0,1) ; (3,1) <--> (3,2) ; (1,3) <--> (1,4) ;
<ADJLIST_END>        <ORIGIN_START> (1,3) <ORIGIN_END>         <TARGET_START> (2,3) <TARGET_END>
<PATH_START> (1,3) (0,3) (0,2) (1,2) (2,2) (2,1) (2,0) (3,0) (4,0) (4,1) (4,2) (4,3) (4,4)
(3,4) (2,4) (2,3) <PATH_END>
```

Figure 4: Example text output format with token regions highlighted. Adjacency list : text representation of the graph, Origin : starting coordinate, Target : ending coordinate, Path : maze solution sequence

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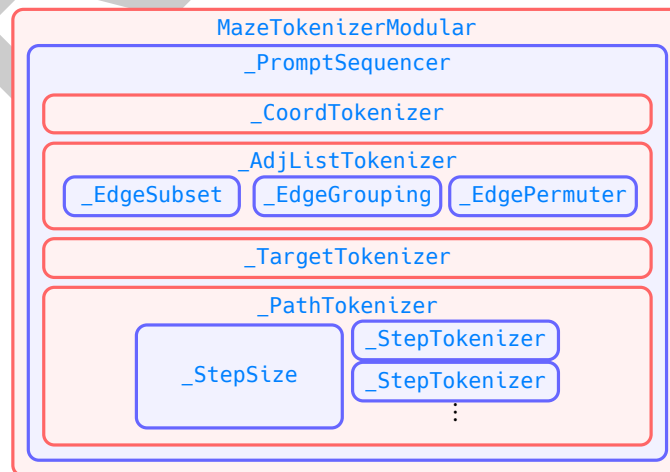
Figure 5: Nested internal structure of `_TokenizerElement` objects inside a typical `MazeTokenizerModular`.

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 variety of 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.

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
`@mark_as_unsupported` 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 in Table 1 and Figure 6. Experiments were
performed on a standard GitHub runner without parallelism.

maze_ctor	keyword args	all sizes	small $g \leq 10$	medium $g \in (10, 32]$	large $g > 32$
dfs		28.0	2.8	20.3	131.8
dfs	accessible_cells=20	2.3	2.2	2.4	2.2
dfs	do_forks=False	2.7	2.2	3.1	3.5
dfs	max_tree_depth=0.5	2.5	2.0	2.7	4.0
dfs_percolation	p=0.1	43.9	2.8	33.9	208.0
dfs_percolation	p=0.4	48.7	3.0	36.5	233.5
kruskal		12.8	1.9	10.3	55.8
percolation	p=1.0	50.2	2.6	37.2	242.5
recursive_div		10.2	1.7	8.9	42.1
wilson		676.5	7.8	188.6	3992.6
mean		559.9	13.0	223.5	3146.9
median		11.1	6.5	32.9	302.7

Table 1: Generation times for various algorithms and maze sizes. More information can be found on the
benchmarks page.
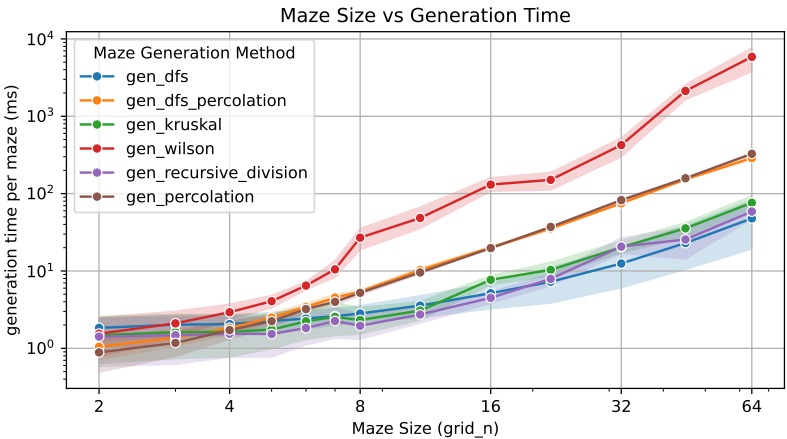
Figure 6: Plot of maze generation time. Generation time scales exponentially with maze size for all
algorithms. Generation time per maze does not depend on the number of mazes being generated, and
there is minimal overhead to initializing the generation process for a small dataset. Wilson's algorithm
is notably less efficient than others and has high variance. Note that values are averaged across all
parameter sets for that algorithm. More information can be found on the benchmarks page.

Success Rate Estimation

In order to replicate the exact dataset distribution of (Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021), the parameter `MazeDatasetConfig.endpoint_kwargs: EndpointKwargsType` allows for additional constraints such as enforcing that the start or end point be in a "dead end" with only one accessible neighbor cell. However, combining these constraints with cyclic mazes (such as those generated with percolation), as was required for the work in (Knutson et al., 2024), can lead to an absence of valid start and end points. Placing theoretical bounds on this success rate is difficult, as it depends on the exact maze generation algorithm and parameters used. To deal with this, our package provides a way to estimate the success rate of a given configuration using a symbolic regression model trained with PySR (Cranmer, 2023). More details on this can be found in `estimate_dataset_fractions.ipynb`. Using the estimation algorithm simply requires the user to call `cfg_new: MazeDatasetConfig = cfg.success_fraction_compensate()`, providing their initial `cfg` and then using the returned `cfg_new` in its place.

Success Rate Estimation Algorithm

The base function learned by symbolic regression provides limited insight and may be subject to change. It is defined as `cfg_success_predict_fn`, and takes a 5 dimensional float vector created by `MazeDatasetConfig._to_ps_array()` which represents the [percolation value, grid size, endpoint deadend configuration, endpoint uniqueness, categorical generation function index].

However, the outputs of this function are not directly usable due to minor divergences at the endpoints with respect to the percolation probability p. Since we know that maze success is either guaranteed or impossible for $p = 0$ and $p = 1$, we define the `soft_step` function to nudge the raw output of the symbolic regression. This function is defined with the following components:

shifted sigmoid σ_s, amplitude scaling A, and h function given by

$$\sigma_s(x) = (1 + e^{-10^3 \cdot (x - 0.5)})^{-1} \qquad A(q, a, w) = w \cdot (1 - |2q - 1|^a)$$

$$h(q, a) = q \cdot (1 - |2q - 1|^a) \cdot (1 - \sigma_s(q)) + (1 - (1 - q) \cdot (1 - |2(1 - q) - 1|^a)) \cdot \sigma_s(q)$$

We combine these to get the `soft_step` function, which is identity-like for $p \approx 0.5$, and pushes x to extremes otherwise.

$$\text{soft\_step}(x, p, \alpha, w) = h(x, A(p, \alpha, w))$$

Finally, we define

$$\text{cfg\_success\_predict\_fn}(\mathbf{x}) = \text{soft\_step}(\text{raw\_val}, x_0, 5, 10)$$

where `raw_val` is the output of the symbolic regression model. The parameter x_0 is the percolation probability, while all other parameters from `_to_ps_array()` only affect `raw_val`.
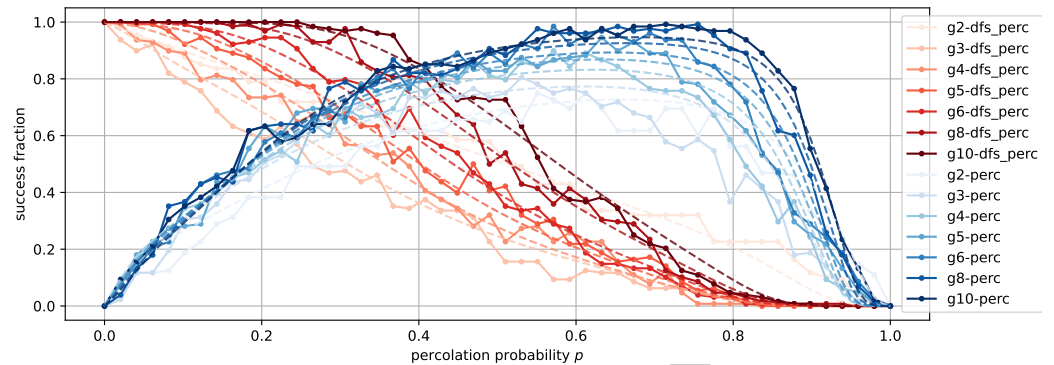
Figure 7: An example of both empirical and predicted success rates as a function of the percolation probability p for various maze sizes, percolation with and without depth first search, and `endpoint_kwargs` requiring that both the start and end be in unique dead ends. Empirical measures derived from a sample of 128 mazes. More information can be found on the benchmarks page.

Implementation

We refer to our repository and docs for documentation and up-to-date implementation details.

This package utilizes a simple, efficient representation of mazes as subgraphs of a finite lattice, which we call a `LatticeMaze`. Using an adjacency matrix for storing mazes would be memory inefficient by failing to exploit the highly sparse structure – for example, for a 2-dimensional maze, only 4 off-diagonal bands would be have nonzero values. On the other hand, using an adjacency list could lead to a poor lookup time for whether any given connection exists.

Instead, we describe mazes with the following representation: for a 2-dimensional lattice with r rows and c columns, we initialize a boolean array

$$A = \{0, 1\}^{2 \times r \times c}$$

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 *rightward* connection to $[i, j+1]$ exists. Thus, we avoid duplication of data about the existence of connections and facilitate fast lookup time, at the cost of requiring additional care with indexing. Note that this setup allows for a periodic lattice. Generation of mazes is detailed in `LatticeMazeGenerators`.

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. The endpoint selection can be controlled via `MazeDatasetConfig.endpoint_kwargs: EndpointKwargsType`, and complications caused by this are detailed in section: *Success Rate Estimation*. A maze with a solution is denoted a `SolvedMaze`, which inherits from `LatticeMaze`.

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

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 (Michael Igorevich Ivanitskiy, Spies, et al., 2023; Michael Igorevich Ivanitskiy, Shah, et al., 2023; Spies et al., 2024). It was extended 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), which required matching the pixel-padded and endpoint constrained output format of (Schwarzschild, Borgnia, Gupta, Bansal, et al., 2021). Ongoing work using `maze-dataset` aims to investigate the effects of varying the tokenization format on the performance of pretrained LLMs on spatial reasoning.

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

- By (Nolte et al., 2024) 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.

- By (Wang et al., 2024) and (Chen et al., 2024) to study the effectiveness of imperative learning.

- By (Zhang et al., 2025) to introduce a novel framework for reasoning diffusion models.

- By (Dao & Vu, 2025) to improve spatial reasoning in LLMs with GRPO.

Acknowledgements

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*: Imperative learning-based A* 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*.

Cranmer, M. (2023). Interpretable machine learning for science with PySR and SymbolicRegression. jl. *arXiv Preprint arXiv:2305.01582*.

Dao, A., & Vu, D. B. (2025). AlphaMaze: Enhancing large language models' spatial intelligence via GRPO. *arXiv Preprint arXiv:2502.14669*.

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.

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, Shah, R., Spies, A. F., Räuker, T., Valentine, D., Rager, C., Quirke, L., Mathwin, C., Corlouer, G., Behn, C. D., & others. (2023). A configurable library for generating and manipulating maze datasets. *arXiv Preprint arXiv:2309.10498*.

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*.

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.

276   *Advances in Neural Information Processing Systems*, *37*, 112329–112355.

277 Knutson, B., Rabeendran, A. C., Ivanitskiy, M., Pettyjohn, J., Diniz-Behn, C., Fung, S. W.,
278   & McKenzie, D. (2024). On logical extrapolation for mazes with recurrent and implicit
279   networks. *arXiv Preprint arXiv:2410.03020*.

280 Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman
281   problem. *Proceedings of the American Mathematical Society*, *7*(1), 48–50.

282 Liu, C., & Wu, B. (2023). Evaluating large language models on graphs: Performance insights
283   and comparative analysis. *arXiv Preprint arXiv:2308.11224*.

284 Nag, A. (2020). MDL suite: A language, generator and compiler for describing mazes. *Journal*
285   *of Open Source Software*, *5*(46), 1815.

286 Németh, F. (2019). *Maze-generation-algorithms*. https://github.com/ferenc-nemeth/
287   maze-generation-algorithms

288 Nolte, N., Kitouni, O., Williams, A., Rabbat, M., & Ibrahim, M. (2024). Transformers can
289   navigate mazes with multi-step prediction. *arXiv Preprint arXiv:2412.05117*.

290 Oppenheim, J. (2018). *Maze-generator: Generate a random maze represented as a 2D array*
291   *using depth-first search*. https://github.com/oppenheimj/maze-generator/; GitHub.

292 Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z.,
293   Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M.,
294   Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An
295   Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A.
296   Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information*
297   *processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/
298   paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

299 Pfau, J., Merrill, W., & Bowman, S. R. (2024). Let's think dot by dot: Hidden computation
300   in transformer language models. *arXiv Preprint arXiv:2404.15758*.

301 Räuker, T., Ho, A., Casper, S., & Hadfield-Menell, D. (2023). Toward transparent ai: A survey
302   on interpreting the inner structures of deep neural networks. *2023 IEEE Conference on*
303   *Secure and Trustworthy Machine Learning (SaTML)*, 464–483.

304 Schwarzschild, A., Borgnia, E., Gupta, A., Bansal, A., Emam, Z., Huang, F., Goldblum, M., &
305   Goldstein, T. (2021). *Datasets for Studying Generalization from Easy to Hard Examples*
306   (No. arXiv:2108.06011). arXiv. https://doi.org/10.48550/arXiv.2108.06011

307 Schwarzschild, A., Borgnia, E., Gupta, A., Huang, F., Vishkin, U., Goldblum, M., & Goldstein,
308   T. (2021). Can you learn an algorithm? Generalizing from easy to hard problems with
309   recurrent networks. *Advances in Neural Information Processing Systems*, *34*, 6695–6706.

310 Singla, A. (2023). Evaluating ChatGPT and GPT-4 for visual programming. *arXiv Preprint*
311   *arXiv:2308.02522*.

312 Spies, A. F., Edwards, W., Ivanitskiy, M. I., Skapars, A., Räuker, T., Inoue, K., Russo, A., &
313   Shanahan, M. (2024). Transformers use causal world models in maze-solving tasks. *arXiv*
314   *Preprint arXiv:2412.11867*.

315 Wang, C., Ji, K., Geng, J., Ren, Z., Fu, T., Yang, F., Guo, Y., He, H., Chen, X., Zhan, Z., &
316   others. (2024). Imperative learning: A self-supervised neural-symbolic learning framework
317   for robot autonomy. *arXiv Preprint arXiv:2406.16087*.

318 Wilson, D. B. (1996). Generating random spanning trees more quickly than the cover time.
319   *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing -*
320   *STOC '96*, 296–303. https://doi.org/10.1145/237814.237880

321 Zhang, T., Pan, J.-S., Feng, R., & Wu, T. (2025). T-SCEND: Test-time scalable MCTS-

322   enhanced diffusion model. *arXiv Preprint arXiv:2502.01989*.