EuroVis Workshop on Visual Analytics (2019)
T. von Landesberger and C. Turkay (Editors)

Visualization of Rubik's Cube Solution Algorithms

C. A. Steinparz[1], A. P. Hinterreiter[1], H. Stitz[1] and M. Streit[1]

[1]Johannes Kepler University Linz, Institute of Computer Graphics, Austria

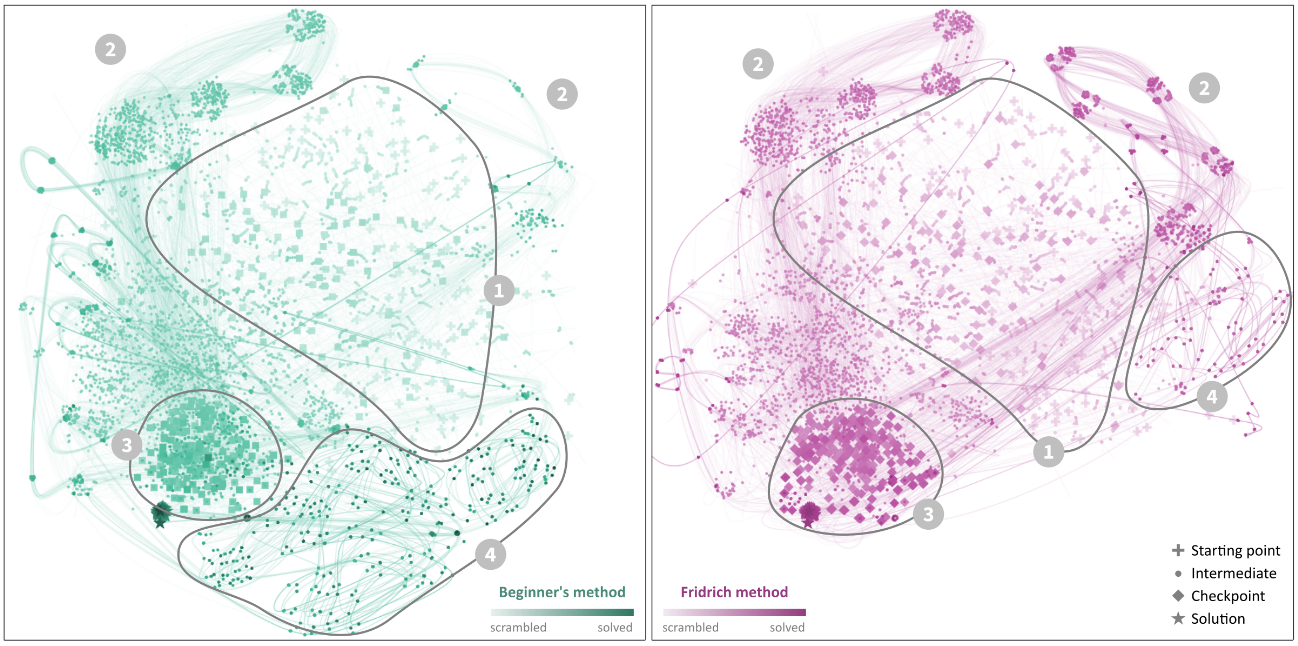


Figure 1: *Projected solution pathways for 100 random Rubik's Cubes solved with the beginner's method (left), and with Fridrich's method (right), respectively. Data for both algorithms was combined for the t-SNE calculation, but is shown in two individual visualizations for easier interpretation. The random initial states form a broad cluster near the center of the projected state space (1). Both solution algorithms take similar intermediate paths (2), and later checkpoints cluster densely near the final solution (3). Notably, Fridrich's algorithm avoids lengthy sequences of rotations right before the solution (4).*

Abstract

Rubik's Cube is among the world's most famous puzzle toys. Despite its relatively simple principle, it requires dedicated, carefully planned algorithms to be solved. In this paper, we present an approach to visualize how different solution algorithms navigate through the high-dimensional space of Rubik's Cube states. We use t-distributed stochastic neighbor embedding (t-SNE) to project feature vector representations of cube states to two dimensions. t-SNE preserves the similarity of cube states and leads to clusters of intermediate states and bundles of cube solution pathways in the projection. Our prototype implementation allows interactive exploration of differences between algorithms, showing detailed state information on demand.

1. Introduction

The increasing popularity of machine learning applications has been accompanied by a growing demand for tools to visualize and help understand complex algorithms [HKPC18]. In this paper, we chose Rubik's Cube as an example of a difficult problem solving task, and show how a visualization can help understand different solution algorithms acting in a high-dimensional space.

Rubik's Cube is a famous puzzle toy devised in 1974 by the Hungarian inventor and professor of architecture Ernő Rubik. With 350 million cubes sold around the world by January 2009, it is the

DOI: 10.2312/eurova.20191119

world's top-selling puzzle game [Ada09]. The classic version of Rubik's Cube has six faces, with each face being made up by a 3×3 grid of colored facets. These facets are the faces of smaller cubes, the so-called cubies. The cube consists of 26 such cubies: 8 corner cubies with three faces each, 12 edge cubies with two faces each, and 6 center cubies with one face each. The 27th cubie at the center of the cube is replaced by a mechanism that enables rotation of one 9-cubie slice of the cube at a time. For a standard Rubik's Cube, or any $n \times n \times n$ cube with odd n, the relative positions of the center cubies cannot change. The color of the center facet of one cube face is thus assigned to the whole face, and the cube is considered solved when each facet is correctly positioned on its color's face. Otherwise, the cube is said to be in a scrambled state.

Rubik's Cube is commonly known to be almost impossible to solve if no specific solution strategy or algorithm is applied. The closer a cube is to being solved and the more cubies are in the correct position, the more likely it is that any rotation with the intention of solving another part of the cube will scramble already correctly placed cubies. Therefore, precisely considered sequences of rotations need to be applied, which ensure that only specific cubies are moved to their intended destinations. Many solution strategies for Rubik's Cube have been developed. They differ in complexity and speed, depending on the amount of special patterns and conditions that are detected and utilized during the solving process. Generally, the faster a solution algorithm is and the fewer rotations are needed, the more sub-algorithms need to be learned and applied under the correct conditions. The classic beginners' method is highly inefficient but has only few sub-algorithms to be memorized. More advanced methods, such as Fridrich's CFOP method [Fri97] or the Petrus method, are harder to learn but usually require significantly fewer moves. Generally, solution algorithms often use checkpoints: special points in the state-space of the cube (e. g., having the yellow cross on the yellow side). This state-space, in which the solution algorithms act, is high-dimensional and encompasses more than 4.3×10^{19} unique states, which makes visualizing the solution pathways challenging.

In this paper, we present an approach for visualizing solution strategies for Rubik's Cube by showing how algorithms navigate differently through projections of the high-dimensional state-space. We focus on two solution strategies: a simple method that will be referred to as *beginner's method* and the more advanced CFOP method by Jessica Fridrich, hereafter referred to as *Fridrich method*. We further describe our implementation of a prototype for interactively exploring the projected solution pathways, and show how it can be used to answer the following questions:

Q1 Do checkpoints cluster for different initial states?

Q2 When exactly do algorithms start to diverge after starting from the same initial states?

Q3 Are the differences between algorithms immediately recognizable?

2. Related Work

Detailed information on a number of different Rubik's Cube solution algorithms has been compiled in several Wikis and Internet forums [Fer, Mak12]. AlgExplorer [Teo17], published on such an online forum in 2017, is a command-line tool for exploring the algorithm/solution space based on an extensive record of Rubik's Cube solutions [Jaf]. Scientific papers on Rubik's Cube have typically been concerned with finding optimal solutions. Kunkle et al., for instance, used out-of-core parallel computing and properties of the cube's permutation group to find a lower bound of 26 moves [KC07]. However, to our knowledge, no attempts at visualizing different solution strategies have been published.

The cube states along the solution pathways can be viewed essentially as a multivariate time series, with the cube face colors as categorical variables. Combining dimensionality reduction techniques with trajectory encoding is a common choice for visualizing multivariate time series. TimeSeriesPath [BWS*12], for instance, is based on principal component analysis (PCA). The Time Curves idiom by Bach et al. [BSH*16] visualizes patterns in temporal data by folding time series into trajectories according to a similarity measure. Schreck et al. used self organizing maps (SOMs) to visualize time-resolved financial data as trajectories [STKF07]. In a similar fashion, textual data has been visualized as projected n-gram series, e. g., by Mao et al. [MDL07]. A generalized version of n-gram trajectories for non-textual data was proposed by Ward and Guo [WG11]. Both of these approaches rely on PCA. Brown et al. used multidimensional scaling (MDS) to visualize system states as trajectories during user interaction [BYC*18]. In addition to PCA, SOMs, and MDS, also t-distributed stochastic neighbor embedding (t-SNE) [MH08] has proven a powerful tool for creating visualizations of high dimensional sequential data. Van den Elzen et al. applied t-SNE, among other projection techniques, to visualize high-dimensional states of dynamic networks [vdEHBvW16].

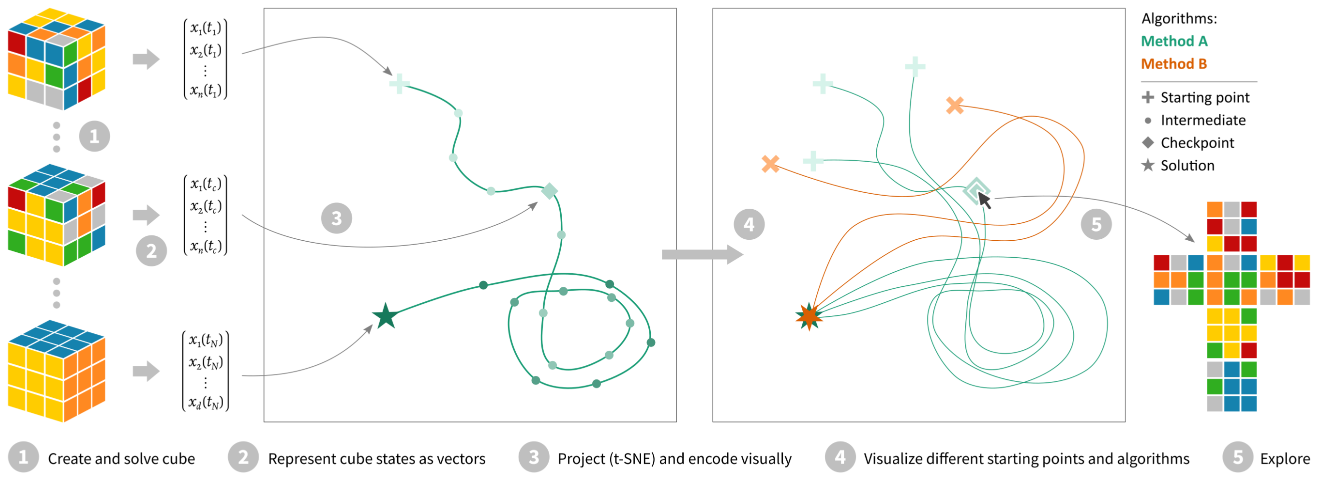
For our visualization of Rubik's Cube solution algorithms, we use a combination of dimensionality reduction, in our case t-SNE, and encoding state sequences as trajectories.

3. Method

Figure 2 shows the workflow for creating the Rubik's Cube solver visualization. This section describes steps 1 to 5 conceptually, including cube initialization, data abstraction, projection, visualization, and interaction. Technical details on the implementation are given in Section 4.

3.1. Cube States as Feature Vectors

We create randomly scrambled cubes and automatically apply different solution algorithms to them. The first step of visualizing the solution pathways is transforming the cube states into numerical representations for further processing. We based the encoding of the cube state on the data structure underlying the Rubik's Cube Solver Python API by Lucas Liberacki and Tom Brannan [LB15] (see Section 4 for implementation details). Each face of the cube is represented by a 3×3 matrix with one entry for each facet. For the entries representing the facet colors we chose a one-hot encoding, i. e., (0,0,0,0,0,1) for red, (0,0,0,0,1,0) for green etc. This encoding ensures meaningful distance metrics for the dimensionality reduction. The resulting $6 \times (3 \times 3) \times 6$ tensor was flattened, yielding a feature vector of length 324 for a single state.



$x_1(t_1)$, $x_2(t_1)$, ⋮, $x_n(t_1)$
$x_1(t_c)$, $x_2(t_c)$, ⋮, $x_n(t_c)$
$x_1(t_N)$, $x_2(t_N)$, ⋮, $x_d(t_N)$
Algorithms:
Method A
Method B
Starting point
Intermediate
Checkpoint
Solution
1 Create and solve cube
2 Represent cube states as vectors
3 Project (t-SNE) and encode visually
4 Visualize different starting points and algorithms
5 Explore

Figure 2: *Workflow for the Rubik's Cube solver visualization.*

3.2. Projection

The high-dimensional feature vectors of complete solution pathways are projected to a two-dimensional space using *t*-SNE. *t*-SNE ensures that similar states are close to each other in the projected space. We found that PCA, due to its linear nature, could not preserve high-dimensional clusters in the low-dimensional projected space. Note that *t*-SNE does not guarantee to map two equal states to the exact same point (as might be suggested by Figure 2), but the projections will usually be very close together.

The notion of similarity in the original feature space depends on an arbitrarily chosen distance metric. We tried different metrics and ultimately settled on Euclidean distance, since it yielded the most appealing results. Our way of projecting the cube states to two dimensions does not take into account the number of operations (i. e., rotations of cube slices) necessary to go from one state to the next. Instead, we argue that applying the Euclidean distance to our choice of feature vectors yields a representation that is more in line with the intuitive judgment of how scrambled the cube is. The reason for this is that, for a one-hot encoding, the Euclidean distance is equal to the square root of the Hamming distance. It thus corresponds to a "naive edit distance", that measures wrongly colored cube facets, but not the number of moves required to fix them.

3.3. Visualization and Interaction

For the visualization we create a separate path for each cube, going from its randomly scrambled origin to its fully solved endpoint. These paths pass through the projections of all intermediate states, and are implemented as Bezier curves as opposed to lines. This ensures better readability when many cubes are displayed. The paths for different solution algorithms are visually encoded by hue. Each state of a cube along its path is displayed by a marker. We visualize the random origins as crosses, checkpoints as squares, and the solved states as stars, respectively. The square markers for different algorithm are slightly rotated, to make them distinguishable from one another when two paths share points in the projected state space. Disk markers for intermediate states between checkpoints can be displayed on demand. The markers' brightness values encode the progression through the solution path, with bright colors corresponding to early states. For each marker, a mouse-over action reveals a detailed view of the cube state, showing the colored facets of the unfolded cube.

4. Implementation

We use Python to prepare the datasets for the visualization. First, we create randomly scrambled cubes by starting from a solved cube and performing random rotations. This ensures that only physically possible cube states are produced. The cubes are then solved with a modified version of the previously mentioned API by Liberacki and Brannan [LB15], to which we added a new solution algorithm as well as options for data export. *t*-SNE is applied to the generated cube states, and the projected states are written to CSV files along with detailed facet color information. We use the scikit-learn [PVG*11] *t*-SNE implementation with a learning rate of 100 and a perplexity of 50. The hyperparameters were chosen experimentally to yield satisfying results across many different datasets. For the actual visualization we use D3.js [OHB11] to render SVGs.

A deployed version of our prototype implementation can be accessed at `https://rubiks-cube-vis.netlify.com`.

5. Results and Discussion

The following section describes how our visualization of the solution algorithms helps answering questions **Q1** to **Q3** raised in the introduction.

5.1. Clustering

Figure 1 shows the projected solution paths for 100 randomly chosen initial cube states, both for the beginner's method and the more

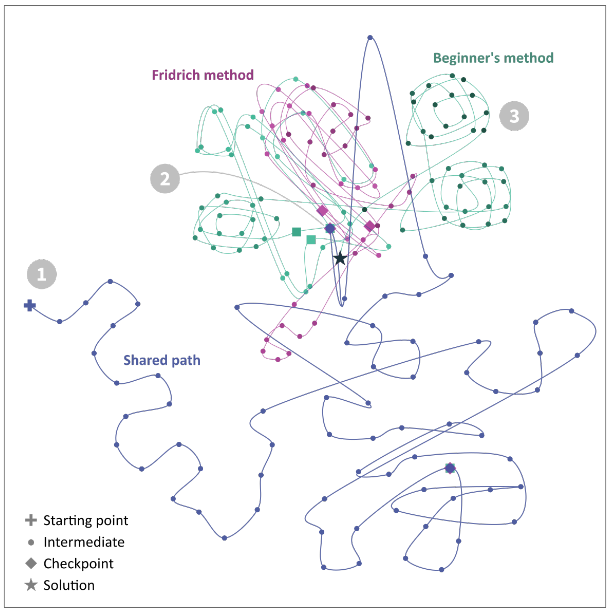


Figure 3: *Projected solution pathways of the same initial state (1) solved with beginner's method and Fridrich's method, respectively. The algorithms share the same path up to the second checkpoint (2), at which two layers of the cube are already fully solved. Near the end, the beginner's method requires additional rotations, while Fridrich's method approaches the solution much faster.*

advanced Fridrich's method. Clearly, projecting the cube states with *t*-SNE leads to the formation of different clusters. Most prominently, later checkpoints form dense clusters close to the final solution (see (3) in Figure 1), clearly answering question **Q1** in the affirmative. Earlier checkpoints with fewer correctly positioned cube facets are much more spread out. These checkpoints share a wide, sparse region with the randomly selected initial states (1). In later stages along the solution paths, intermediate states tend to form clusters as well, which leads to bundles of parallel paths between checkpoints (2).

5.2. Comparison of Algorithms

Figure 3 shows solution paths for the beginner's method and Fridrich's method applied to the same initial state (1). Our choice of color-mixing and the use of opacity allows us to answer question **Q2** positively: the solution paths overlap completely up to the second checkpoint (2). As the visualization, upon user demand, provides detailed views of unfolded cubes for all intermediate states, it can be seen that already two layers of the cube are fully solved at this point. We found this behavior to be general, i. e., independent of the initial state. Afterwards, the more optimized Fridrich method makes use of a large amount of different sub-algorithms. This avoids additional lengthy sequences of rotations close to the final solution. For the beginner's method, these rotations show up as characteristic coils (3).

When projecting more cube solutions, the differences between the strategies due to the use of sub-algorithms become even more apparent (see Figure 1). Our visualization shows large bundles of paths as a result of clusters of similar intermediate points. Up to a certain point, these bundles appear similar for the two different techniques (2). Further along the solution paths, the beginner's method's sub-optimal approach to correctly positioning the final cubies gives rise to a large cluster of many "avoidable" intermediate steps (see (4) in the left part of Figure 1).

In case of Fridrich' method, many fewer steps are required during the final stages of solving the cube, leading to a much smaller and less populated cluster of intermediate points (see (4) in the right part of Figure 1). Thus, the scalability of our visualization approach to hundreds of cubes allows us to reliably detect differences between the beginner's method and Fridrich's method, answering question **Q3** successfully.

5.3. Limitations

As already stated in Section 3.2, our choice of state representation and projection does not preserve the symmetries of the Rubik's Cube permutation group. Furthermore, the implementation of the solutions forces each solution pathway to reach the yellow cross checkpoint first, even when an equivalent checkpoint (i. e., having the cross on a different face) would be closer.

Finally, the visualization is currently rendered using SVGs. This causes long update times and potentially sluggish, unresponsive behavior when more than a couple of hundred cube solutions are shown. We will address this issue in a future version of our visualization tool, by switching from the SVG-based D3 implementation to VegaLite [SMWH17] with its more efficient canvas rendering. To further improve the interaction and reduce visual clutter when many paths are drawn, edge bundling may be used.

6. Conclusions & Outlook

In this work we showed how dimensionality reduction can be used to visualize the complex solution paths through the high-dimensional, vast space of Rubik's Cube states. Interactivity and display of detailed information on demand enable the user to comparatively explore different solution algorithms.

We plan to combine this visualization approach with a real Bluetooth connected Rubik's Cube. Rotations performed with this cube will then be directly projected in our visualization by means of an out-of-sample extension [GSH15].

Recently, McAller et al. showed that reinforcement learning enables artificially intelligent agents to autonomously learn how to solve Rubik's Cube [MASB18]. Our visualization approach is general enough to also be applied to such automatically found algorithms, and we plan to use it as a starting point for understanding how AI systems approach complex problem solving tasks.

7. Acknowledgments

This work was supported in part by the State of UpperAustria (FFG 851460, Human-Interpretable Machine Learning) and the Austrian Science Fund (FWF P27975-NBL).

References

[Ada09] Adams W. L.: The Rubik's cube: A puzzling success. *Time* (2009). URL: https://web.archive.org/web/20090201200141/http://www.time.com/time/magazine/article/0,9171,1874509,00.html. 2

[BSH*16] Bach B., Shi C., Heulot N., Madhyastha T., Grabowski T., Dragicevic P.: Time Curves: Folding time to visualize patterns of temporal evolution in data. *IEEE Transactions on Visualization & Computer Graphics 22*, 01 (2016), 559–568. doi:10.1109/TVCG.2015.2467851. 2

[BWS*12] Bernard J., Wilhelm N., Scherer M., May T., Schreck T.: TimeSeriesPaths: Projection-based explorative analysis of multivariate time series data. In *Journal of WSCG* (2012), pp. 97–106. URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-227012. 2

[BYC*18] Brown E. T., Yarlagadda S., Cook K. A., Chang R., Endert A.: Modelspace: Visualizing the trails of data models in visual analyticssystems. *IEEE VIS 2018 Workshop: Machine Learning from User Interaction for Visualization and Analytics* (2018). URL: https://learningfromusersworkshop.github.io/papers/ModelSpace.pdf. 2

[Fer] Ferenc D.: Different Rubik's cube solving methods. Accessed: 2019-02-28. URL: https://ruwix.com/the-rubiks-cube/different-rubiks-cube-solving-methods/. 2

[Fri97] Fridrich J.: My system for solving Rubik's cube, 1997. Accessed: 2019-02-25. URL: http://www.ws.binghamton.edu/fridrich/system.html. 2

[GSH15] Gisbrecht A., Schulz A., Hammer B.: Parametric nonlinear dimensionality reduction using kernel t-SNE. *Neurocomputing 147* (2015), 71–82. doi:https://doi.org/10.1016/j.neucom.2013.11.045. 4

[HKPC18] Hohman F., Kahng M., Pienta R., Chau D. H.: Visual analytics in deep learning: An interrogative survey for the next frontiers. *CoRR* (2018). arXiv:1801.06889. 1

[Jaf] Jaffray J.: Cubesolves. Accessed: 2019-02-25. URL: http://cubesolv.es/. 2

[KC07] Kunkle D., Cooperman G.: Twenty-six moves suffice for rubik's cube. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation* (New York, NY, USA, 2007), ISSAC '07, ACM, pp. 235–242. doi:10.1145/1277548.1277581. 2

[LB15] Liberacki L., Brannan T.: Rubik's cube solver coded in python, 2015. Accessed: 2019-02-28. URL: https://github.com/CubeLuke/Rubiks-Cube-Solver. 2, 3

[Mak12] Makisumi S.: Speedsolving.com wiki. category: 3x3x3 methods, 2012. Accessed: 2019-02-28. URL: https://www.speedsolving.com/wiki/index.php/Category:3x3x3_methods. 2

[MASB18] McAleer S., Agostinelli F., Shmakov A., Baldi P.: Solving the rubik's cube without human knowledge. *CoRR* (2018). arXiv:1805.07470. 4

[MDL07] Mao Y., Dillon J. V., Lebanon G.: Sequential document visualization. *IEEE Transactions on Visualization & Computer Graphics 13* (2007), 1208–1215. doi:10.1109/TVCG.2007.70592. 2

[MH08] Maaten L. v. d., Hinton G.: Visualizing data using t-SNE. *Journal of Machine Learning Research 9* (2008), 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html. 2

[OHB11] Ogievetsky V., Heer J., Bostock M.: D3: Data-Driven Documents. *IEEE Transactions on Visualization & Computer Graphics 17* (2011), 2301–2309. doi:10.1109/TVCG.2011.185. 3

[PVG*11] Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M., Duchesnay E.: Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830. URL: http://www.jmlr.org/papers/v12/pedregosa11a.html. 3

[SMWH17] Satyanarayan A., Moritz D., Wongsuphasawat K., Heer J.: Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization & Computer Graphics 23*, 1 (2017), 341–350. doi:10.1109/TVCG.2016.2599030. 4

[STKF07] Schreck T., Tekušová T., Kohlhammer J., Fellner D.: Trajectory-based visual analysis of large financial time series data. *SIGKDD Explor. Newsl. 9*, 2 (2007), 30–37. doi:10.1145/1345448.1345454. 2

[Teo17] Teoidus: AlgExplorer: command-line utility to assist in alg searching, 2017. Accessed: 2019-02-28. URL: https://www.speedsolving.com/forum/threads/65189/. 2

[vdEHBvW16] van den Elzen S., Holten D., Blaas J., van Wijk J. J.: Reducing snapshots to points: A visual analytics approach to dynamic network exploration. *IEEE Transactions on Visualization & Computer Graphics 22*, 1 (2016), 1–10. doi:10.1109/TVCG.2015.2468078. 2

[WG11] Ward M. O., Guo Z.: Visual exploration of time-series data with shape space projections. *Computer Graphics Forum 30*, 3 (2011), 701–710. doi:10.1111/j.1467-8659.2011.01919.x. 2