Learning Constrained Graph Layout for Content Generation

Seth Cooper[1], Eden Balema[2]

[1]Northeastern University, USA
[2]Dickinson College, USA

Abstract
Graphs are a common format for procedurally generated content in games. However, many graph-based approaches require grammars to be manually authored or additional processing to add spatial layout information. In this work, we extend an existing system for constrained graph generation that learns from examples. The main extension is adding graph layout information directly into the graph learning and generation constraint problem by incorporating spatial transforms. We demonstrate the approach in several level generation applications and potential use in citizen science games.

Keywords
graphs, procedural content generation, constraints

1. Background

Graphs are one of many representations for procedurally generated content [1] in games. When using graphs, grammars are a popular technique [2, 3, 4, 5, 6]. However, many graph-based approaches require these grammars to be manually authored or additional processing to add spatial layout information. Dormans's work [7], for example, uses a two-pass system of grammars, one that first generates an abstract "mission" graph, which is then used by a shape grammar to create the "space" for the level.

A few approaches have learned graph generation from examples, which can be considered a form of Procedural Content Generation via Machine Learning (PCGML) [8]. For example, Londoño and Missura's [9] work used example Super Mario Bros. levels to learn graph patterns and grammars. The approach of Hauck and Aranha [10] learns a grammar for generating Super Mario Bros. levels, but is highly tailored to that game's tile grid structure. Merrell and Manocha's [11] work on continuous model synthesis can also be considered learning to generate graphs with layout from examples, though relies on constructing arrays of intersecting lines or planes and using these for node and edge placement. Merrell's [12] more recent work can also be considered learning graph layout from examples, and uses grammars.

In terms of graph layout, a wide variety of algorithms have been developed, many incorporated into packages such as GraphViz [13] and Tulip [14]. These generally take an existing graph as input, rather than generating a graph and layout simultaneously.

AIIDE Workshop on Experimental Artificial Intelligence in Games, October 08, 2023, University of Utah, Utah, USA
se.cooper@northeastern.edu (S. Cooper);
balemae@dickinson.edu (E. Balema)
0000-0003-4504-0877 (S. Cooper)

In this work, we extend the existing Sturgeon-GRAPH system [15] for constrained graph generation. Sturgeon-GRAPH learns local patterns from example graphs and uses them to generate new graphs; however, prior to this work, it did not use spatial information, only learning connectivity. Here, we incorporate relative spatial relationships between nodes in the patterns learned by Sturgeon-GRAPH. These spatial relationships are then incorporated into the graph generation constraint problem. Thus, the solution to the problem generates both the graph connectivity structure and the spatial layout of the nodes. We also use some other extensions in support of this goal, including an alternate graph connectivity constraint and pattern definitions. We demonstrate the approach in level generation and possible use as a tool in molecular citizen science games.

2. Overview

This work extends the existing Sturgeon-GRAPH system [15] for constrained graph generation from examples. That system, built on the Sturgeon constraint-based PCG system [16], first extracts a *graph description*, consisting of distinct local patterns of labelled nodes and edges, from example graphs. This graph description is then used to set up a system of Boolean constraints, the solution of which is a graph in which all patterns exist in the example graphs. For efficiency, the system only considers a subset of edges as potential edges for the solution, rather than all possible edges between every pair of nodes (e.g. *band-n* edges only consider edges between nodes from id i up to id $i + n$).

Previously, Sturgeon-GRAPH used a constraint problem that handled graph generation, where layout was a separate post-processing step. In this work, we extend the Sturgeon-GRAPH system to incorporate layout information into the same constraint problem along with graph generation. This way, graph generation and layout are solved in a single problem. This is done by including constraints on relative spatial transforms, with support for two types of 2D transformations (discussed below). For the use of relative transformations, the proposed approach works only with directed *dtree* or *dag* graph types.

2.1. Graph Description Learning

When learning the graph description, before extracting local patterns from the graph, relative spatial transforms are added to the (directed) edges of the graph—the relative transform from the edge source to destination. This is done by starting at the root of the graph and proceeding breadth-first through the graph, adding the relative transform from the source to the destination to each edge.

The two types of 2D transforms currently supported are translation-only ($\mathcal{T}$) and translation-rotation ($\mathcal{TR}$). For $\mathcal{T}$ transforms, the relative transform on an edge is just the relative x and y delta of the nodes. For $\mathcal{TR}$ transforms, relative transforms are stored in the graph description as polar coordinates: a magnitude, and relative angle from the incoming *primary* edge. As *dag*s can have nodes with more than one incoming edge, when using $\mathcal{TR}$ one of the incoming edges (from the node with lowest ID) is set as the primary incoming edge and used for relative transforms.

After adding transform information to edges, pattern learning from the example graphs proceeds much as before, except that the transform information is also considered when finding unique patterns.

2.2. Constrained Graph Generation

The *relative* transform information along edges is incorporated into the constraint problem to solve for *global* transforms for each of the nodes. In order to incorporate spatial transforms into the constraint problem, Sturgeon's solver API needed to be extended to support real-valued variables and some constraints on them. The existing API only supported Boolean variables; this allowed for a wide variety of low-level solvers to be used, including those that do not support real-valued variables. Thus, the following extensions to the API were only implemented in the low-level Z3 solver [17]. Also here for simplicity, we use the L_∞ distance (e.g. square). The following functions were added to the solver API:

MakeVarXform($xtype$) — Create a new transform variable of type $xtype$ ($\mathcal{T}$ or $\mathcal{TR}$). This may correspond to more than one variable in the underlying low-level solver. In our implementation, type $\mathcal{T}$ uses two real variables, and type $\mathcal{TR}$ uses six real variables representing the top two rows of a 3×3 homogeneous transform matrix, with bottom row implicitly $(0, 0, 1)$. This function is used to allocate a transform variable for each node.

GetVarPosXform(v) — Get the position associated with the given transform variable v. In our implementation, $\mathcal{T}$ is just the two variables, while $\mathcal{TR}$ is the translation part of the matrix. This function is used to get the positions of the nodes out of the solution.

CnstrImpliesXform($v, x0, x1, dx, primary$) — Add a constraint that if the Boolean variable v is true, then transform variable $x1$ is constrained to be the result of the constant dx applied to transform variable $x0$ (in the case of $\mathcal{T}$, this is addition, in the case of $\mathcal{TR}$, matrix multiplication). If the constant $primary$ is false, then only the positional part of $x1$ is constrained to be a small distance around the transformed location. This function is used to set up constraints such that if an edge is present in the solution, its relative transform is applied from its source to its destination.

CnstrIdentityDistXform($mvs, xs, mindist$) — Add constraints that the positional parts of the transform variables xs, corresponding to the Boolean variables mvs (which are variables for missing nodes) that are false, are at least $mindist$ distance apart. Also, add a constraint that the first transform variable in xs is the identity transform and the first variable in mvs is false. This function is used to initialize basic positional constraints on the transforms for nodes that are present in the solution.

2.3. Additional Extensions

In support of this work, we added a few other extensions to Sturgeon-GRAPH.

First, we added support for different pattern definitions. In the original Sturgeon-GRAPH, a pattern consists of a *key* node, the edges connected to that node, and the nodes connected to those edges, i.e. immediate neighbors (edge-node patterns). In this work we added a pattern definition that only includes the key node and its connected edges, ignoring the labels of the connected nodes (edge-only patterns).

We also added a new type of edges to consider, *stripe-*$n_1, n_2, ..., n_m$, which, for a node with id i, considers only edges to nodes with ids $i + n_1$, $i + n_2$, ..., $i + n_m$.

We also added an alternative method for constraining graphs to be connected (i.e., not split into multiple graphs). The previous method added a single "reachability" variable for each node, started from a single reachable root node, and followed directed edges out to make sure all nodes in the graph were reachable from the root node. The use of directed edges prevented independent cycles from being found as connected but also limited generated graphs to a single root node. In this work, we added an approach where connectivity is computed using "reachability layers". There are N layers, and each node has

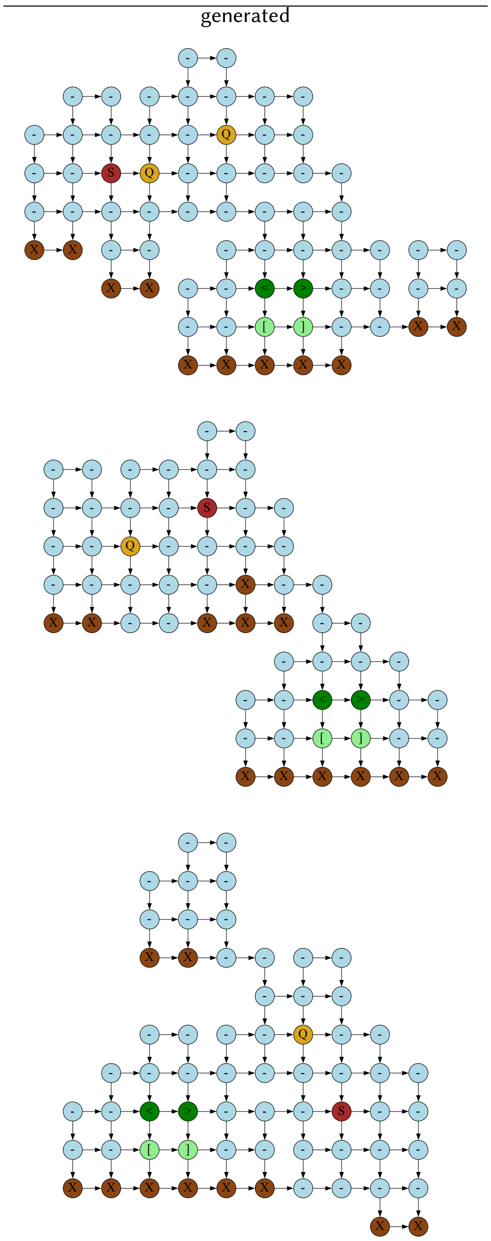

Figure 1: Sample generated `mario` graphs. These represent Super Mario Bros. levels, and node labels are tile text.

a Boolean reachability variable in each layer. In layer 0, exactly 1 node is reachable. If a node is not reachable in layer $n-1$, and not connected (either as source or destination) to a reachable node in layer $n-1$, it is not reachable in layer n. All nodes must either be reachable by the final layer, or missing from the generated graph.

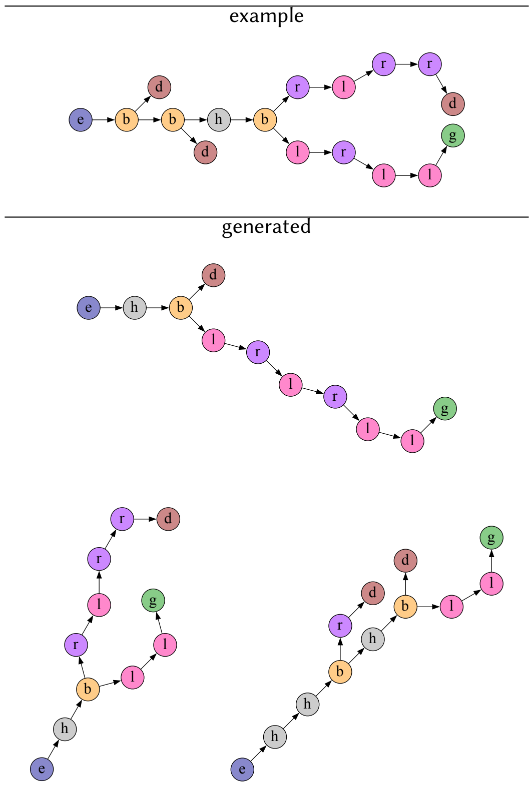

Figure 2: Example graph and sample generated `dungeon` graphs. These represent a dungeon layout, and node labels are type of room.

Finally, we added support for unique undirected cycle labeling. The aim of this is for cycles (or, what would be cycles in the undirected version of the graphs) to be learned as complete units. To accomplish this, we find the cycle basis of the undirected graphs and augment the label of each edge in a basis with the index of its basis and index within its basis.

3. Applications

Here we describe a few applications of the system. For each application, we generated ten graphs, to get timing information and select samples for figures.

`mario`: This application is based on a version of Super Mario Bros. 1-1 [18] from the VGLC [19], converted into a grid graph. Node labels are the tile text from the level. This is similar to the example application in previous Sturgeon-GRAPH work [15], except it considers *stripe-1,8* edges and learns the relative node positions from the example, using them in the graph generation process. Thus the grid layout can be handled without the

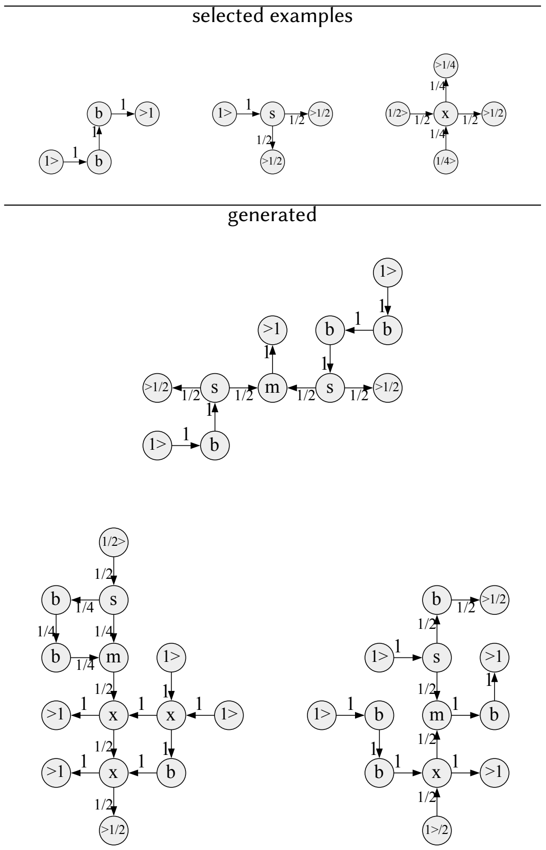

Figure 3: A subset of the example graphs, and generated fract graphs. These represent levels in an educational game about fractions. Node labels are type of piece or space, and edge labels are amount of laser.

specialized grid approach used previously.

This application used $\mathcal{T}$ transforms with edge-node patterns, and generated graphs between 60–80 nodes with *stripe-1,8* edges and at least one of each node label. Sample generated graphs are shown in Figure 1. Average generation time was 52.0s (SD=17.3s).

Notably, the graphs are not required to be rectangular. This could present interesting challenges or opportunities for gameplay.

`dungeon`: This application is a graph with branches and turns, representing a simple dungeon game. It is based on a manually-constructed example level, with nodes labeled for the type of room represented: entrance (`e`), goal (`g`), branch (`b`), right turn (`r`), left turn (`l`), straight hallway (`h`), and dead end (`d`).

This application used $\mathcal{TR}$ transforms with edge-only patterns, and generated graphs between 10–15 nodes with *band-3* edges and at least one of each node label,

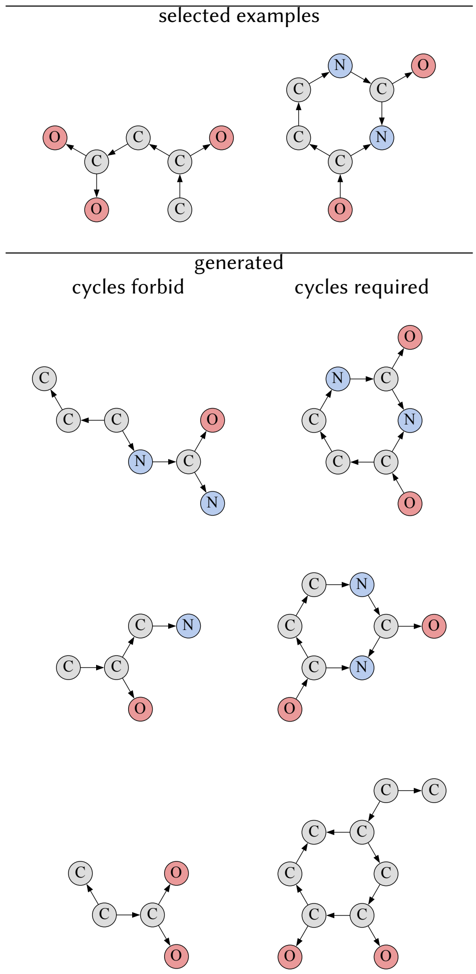

Figure 4: A subset of the example graphs, and generated mol graphs. These represent small molecule structures, and node labels are atom types.

and at most one `e` and `g` node label. The example graph and some sample generated graphs are shown in Figure 2. Average generation time was 35.7s (SD=40.2s).

`fract`: Based on the math educational game, Refraction. In the game, lasers with numerical values come out of sources. Lasers can be manipulated by various pieces such as splitters, which split lasers into fractional parts, or benders, which redirect lasers. The goal is to get lasers with specific values to spaceships to power them up. The game has been used for constraint-based level generation before [20].

In this application, we manually created several small example graphs demonstrating how pieces work. These included labeled nodes for sources (□>), spaceships (>□), splitters (`s`), benders (`b`), and mergers (`m`); we also included nodes for spaces with lasers repeating (r) and crossing (x) so that other pieces would not be placed in their way. Edge labels represent the fractional value of the laser.

This application used $\mathcal{T}$ transforms with edge-only patterns. To increase variety, the example graphs were rotated 90, 180, and 270 degrees. Generated graphs between 10–15 nodes with *band-4* edges and at least one `m` and `s` node labels and two `1>` node labels. Sample example and generated graphs are shown in Figure 3. Average generation time was 15.2s (SD=1.1s).

`mol`: Based on small molecule structures. Recently deep learning methods have been proposed for generating small molecules, for example using SMILES string representations [21] or directly on graphs [22, 23]; however, these learning approaches often use large training datasets. In this application, seven 2D small molecule PDBs from SMPDB [24] were converted to graph format. Node labels were atomic types. Other information, such as bond order (i.e. single/double/triple), was not used, but could potentially be incorporated into labels in the future. This application shows the flexibility of the approach, and we imagine it might be useful as a player tool for molecular design in citizen science games such as Foldit [25].

This application used $\mathcal{TR}$ transforms with edge-node patterns. Undirected cycle labeling was used. Generated graphs were between 5–10 nodes with *band-5* edges. We generated graphs that both forbid and required undirected cycles. Sample examples and generated graphs are shown in Figure 4. Average generation time was 5.3s (SD=3.7s) with cycles forbidden and 211.8s (SD=149.5s) with cycles required.

Given the small size and small number of undirected cycle examples, there was little variety in the graphs requiring undirected cycles, often reproducing ringed examples or very similar. They also took substantially longer to generate.

4. Conclusion

In this work, we present an approach to learning constrained graph layouts from example graphs. This was approached by adding transform information to the graph description and constraints using the Sturgeon-GRAPH system [15].

In the future, we are interested in exploring other constraints on spatial positions, such as nodes being at certain locations, as well as expanding scalability to larger graphs and 3D transformations. We would also like to more thoroughly evaluate the generator, such as with an expressive range [26] or playability analysis.

Acnowledgements

Part of this material is based upon work supported by the National Science Foundation under under Grant No. 1950697, and the Rosetta Commons REU Program.

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, Springer, 2016.

[2] J. M. Font, R. Izquierdo, D. Manrique, J. Togelius, Constrained level generation through grammar-based evolutionary algorithms, in: G. Squillero, P. Burelli (Eds.), Applications of Evolutionary Computation, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2016, pp. 558–573.

[3] J. Valls-Vargas, J. Zhu, S. Ontañón, Graph grammar-based controllable generation of puzzles for a learning game about parallel programming, in: Proceedings of the 12th International Conference on the Foundations of Digital Games, 2017, pp. 7:1–7:10.

[4] R. van Rozen, Q. Heijn, Measuring quality of grammars for procedural level generation, in: Proceedings of the 13th International Conference on the Foundations of Digital Games, 2018, pp. 1–8.

[5] C. Jemmali, C. Ithier, S. Cooper, M. S. El-Nasr, Grammar based modular level generator for a programming puzzle game, in: Proceedings of the Experimental AI in Games Workshop, 2020, p. 7.

[6] A. Madkour, S. Marsella, C. Harteveld, M. Seif El-Nasr, J.-W. van de Meent, Guiding generative graph grammars of dungeon mission graphs via examples, in: Experimental AI in Games Workshop, 2021.

[7] J. Dormans, Adventures in level design: Generating missions and spaces for action adventure games, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010, pp. 1–8.

[8] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, J. Togelius, Procedural Content Generation via Machine Learning (PCGML), IEEE Transactions on Games 10 (2018) 257–270.

[9] S. Londoño, O. Missura, Graph grammars for Super Mario Bros levels, in: Sixth FDG Workshop on Procedural Content Generation, 2015.

[10] E. Hauck, C. Aranha, Automatic generation of Super Mario levels via graph grammars, in: 2020 IEEE Conference on Games (CoG), 2020, pp. 297–304.

[11] P. Merrell, D. Manocha, Continuous model synthesis, ACM Transactions on Graphics 27 (2008) 158:1–158:7.

[12] P. Merrell, Example-based procedural modeling us-

ing graph grammars, ACM Transactions on Graphics 42 (2023) 60:1–60:16.

[13] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, G. Woodhull, Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools, in: M. Jünger, P. Mutzel (Eds.), Graph Drawing Software, Mathematics and Visualization, 2004, pp. 127–148.

[14] D. Auber, Tulip — A Huge Graph Visualization Framework, in: M. Jünger, P. Mutzel (Eds.), Graph Drawing Software, Mathematics and Visualization, Springer, Berlin, Heidelberg, 2004, pp. 105–126.

[15] S. Cooper, Sturgeon-GRAPH: Constrained Graph Generation from Examples, in: Proceedings of the 18th International Conference on the Foundations of Digital Games, 2023, pp. 1–9.

[16] S. Cooper, Sturgeon: Tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 26–36.

[17] L. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, 2008, pp. 337–340.

[18] Nintendo, Super Mario Bros., 1985. Game [NES].

[19] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontañón, The VGLC: The Video Game Level Corpus, arXiv:1606.07487 [cs] (2016).

[20] A. M. Smith, E. Andersen, M. Mateas, Z. Popović, A case study of expressively constrainable level design automation tools for a puzzle game, in: Proceedings of the International Conference on the Foundations of Digital Games, 2012, pp. 156–163.

[21] E. Mazuz, G. Shtar, B. Shapira, L. Rokach, Molecule generation using transformers and policy gradient reinforcement learning, Scientific Reports 13 (2023) 8799.

[22] Q. Liu, M. Allamanis, M. Brockschmidt, A. L. Gaunt, Constrained graph variational autoencoders for molecule design, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 7806–7815.

[23] O. Mahmood, E. Mansimov, R. Bonneau, K. Cho, Masked graph modeling for molecule generation, Nature Communications 12 (2021) 3156.

[24] A. Frolkis, C. Knox, E. Lim, T. Jewison, V. Law, D. D. Hau, P. Liu, B. Gautam, S. Ly, A. C. Guo, J. Xia, Y. Liang, S. Shrivastava, D. S. Wishart, SMPDB: The Small Molecule Pathway Database, Nucleic Acids Research 38 (2010) D480–487.

[25] B. Koepnick, J. Flatten, T. Husain, A. Ford, D.-A. Silva, M. J. Bick, A. Bauer, G. Liu, Y. Ishida, A. Boykov, R. D. Estep, S. Kleinfelter, T. Nørgård-Solano, L. Wei, F. Players, G. T. Montelione, F. DiMaio, Z. Popović, F. Khatib, S. Cooper, D. Baker, De novo protein design by citizen scientists, Nature 570 (2019) 390–394.

[26] G. Smith, J. Whitehead, Analyzing the expressive range of a level generator, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010, pp. 1–7.