Automated Waypoint Generation with the Growing Neural Gas Algorithm

Brian Dellinger
Department of Computer Science
Grove City College
Grove City, PA 16127

Ronald Jenkins
Department of Electrical Engineering
Pennsylvania State University
University Park, PA 16802

Joshua Walton
Insight Global
Cherry Hill, NJ 08003

Abstract

Autonomous navigation in complex continuous spaces often requires the presence of navigation meshes or other methods of determining traversable paths. At present, many navigation meshes are created via a time-consuming manual process. We propose a novel approach for automated generation of waypoints using the Growing Neural Gas algorithm, which generates a graph approximating a complex surface. We illustrate the use of this algorithm on a number of simple two-dimensional mazes and suggest further application for more complex surfaces.

Introduction

Polygon maps are common in games, simulations, and other applications that require automated navigation. As continuous spaces, such maps admit an infinite number of paths between any two points, making automated path generation difficult. Thus, polygon maps are often approximated by finite graphs of *waypoints*; in exchange for some loss of options, such a *navigation mesh* permits rapid navigation via A* (Hart, Nilsson, and Raphael 1968) or similar algorithms. Unfortunately, manually creating a large navigation mesh from a continuous space can be time-consuming, and so automating the task of mesh generation may save significant labor.

The *Growing Neural Gas* (GNG) algorithm (Fritzke and others 1995) is designed to create a finite graph approximating a complex continuous space. Traditionally, this algorithm has been used to approximate high-dimensional topologies with lower-dimensional graphs. We propose to use the GNG algorithm to automatically create a network of navigable paths for a given polygon map. The waypoints resulting from this algorithm could then be used for navigating the map as usual, but without the need for direct human labor.

Related Works

The original implementation of the GNG algorithm (Fritzke and others 1995) leaves several details of the design open for different implementations, resulting in a family of related algorithms. Noteworthy variations include *GNG with Utility Factor* (GNG-U) (Holmström 2002), which adapts to changes in the distribution of navigable space, and *Robust GNG* (Qin and Suganthan 2004), which reduces the time required to form a solution by adjusting the algorithm's response to outliers and grouped nodes. Other variations modify the balance between old and new observations; thus, *Incremental GNG* (Prudent and Ennaji 2005) adds a learning rule to better incorporate new input data, and *TreeGNG* (Doherty, Adams, and Davey 2005) keeps a version history to help correct for early mistakes in mapping.

Automated waypoint generation is not a new topic, with notable existing methods including *decomposition*, which reduces polygon meshes to simpler networks of vertices and edges (Hale and Youngblood 2009). A more recent paper from the same author (Hale 2011) proposes several algorithms to progressively grow waypoint graphs, an approach similar to a GNG's gradual expansion. Alternatively, *voxelization* (Oliva and Pelechano 2013) identifies a world's walkable areas before connecting them together to form a navigation mesh.

Others have used Neural Gas algorithms to emulate player behavior in a game map (Thurau, Bauckhage, and Sagerer 2004). In particular, the *Stable GNG* (SGNG) algorithm (Tencé et al. 2013) was developed specifically to use the GNG algorithm for waypoint generation. To create a navigation mesh producing believable motion, SGNG samples the movement of human players, and its authors consider the effect of a range of parameters on the resulting mesh. While our approach also depends on GNG, our focus was to enable mesh generation with little to no human interaction. We thus sample the underlying polygon map directly; unlike existing approaches, this permits mesh generation even for procedurally-generated spaces or other maps where no prior human navigation records exist.

The Growing Neural Gas Algorithm

The GNG algorithm iteratively approximates topologies with simple finite graphs. Traditionally, such approximations have been used effectively to model continuous spaces. In this paper, we seek to demonstrate GNG's ability to generate a waypoint graph from an arbitrary polygon mesh.

We begin by using the GNG algorithm to approximate the

walkable area of a hallway maze; while the maze walls are technically three-dimensional, its floor is a two-dimensional surface without stairs, pits, or other vertical elements. The approximation produces an undirected graph, consisting of a set of edges E and a set of nodes N. All edges in E have an associated *age*, initialized to 0, and all nodes in N have an associated *error*, also initialized to 0. A series of randomly-generated points called *signals* are chosen from the walkable space of the maze floor; signals are never chosen from points that would lie outside the maze or inside a wall. Through successive iterations of the algorithm, these signals are used to draw the nodes of N closer to the walkable surface of the maze and thereby better approximate the maze topology.

Formally, the base GNG algorithm consists of the following steps:

1. Create two initial nodes a and b at random points within the maze plane, and place them in the set N.
2. Generate a new signal ξ from the walkable area.
3. Find the two nodes $s_1, s_2 \in N$ that are closest to ξ; let s_1 be the closer of the two nodes to ξ. Let δ_ξ be the distance from s_1 to ξ.
4. Increment the age of each edge in E connecting s_1 to one of its neighbors.
5. Add δ_ξ^2 to s_1's error.
6. Move s_1 a distance of $\epsilon_b \delta_\xi$ towards ξ, where $0 \leq \epsilon_b \leq 1$. Likewise, move all of s_1's neighbors towards ξ by some fraction ϵ_n of their respective distances from ξ, where $0 \leq \epsilon_n \leq 1$.
7. If s_1 and s_2 are connected by an edge, set the age of that edge to zero. Otherwise, create an edge between them with an age of 0 and add it to E.
8. Remove all edges in E which have an age larger than some nonnegative value a_{max}. If any node in N has a degree of 0, remove that node.
9. If this is the first pass through algorithm, or if some number λ of signals have been generated since the last node was added, add a new node in the following manner:
   - Find the node $q_1 \in N$ with the largest error. Then, from q_1's neighbors, find the node q_2 with the largest error.
   - Create a new node r halfway between q_1 and q_2, and add it to N.
   - Create new edges between q_1 and r, and between r and q_2, both with an age of 0. Add these edges to E. Remove the original edge from q_1 to q_2.
   - Scale the errors of both q_1 and q_2 by some α, where $0 < \alpha < 1$. Then set r's error to that of q_1.
10. Reduce the error value of each node by multiplying them by some d, where $0 < d < 1$.
11. If a stopping criterion for the algorithm has not yet been met, return to step 2. Otherwise, the algorithm halts, returning the current graph.

Implementation Details

Mazes seemed an appropriate initial testing ground for the GNG algorithm. While a simple two-dimensional maze does not verify the algorithm's suitability for vertical environments, it does allow easy visual inspection of the resulting graph. Furthermore, a maze provides ample opportunity for a graph to produce errors such as disconnections, paths through walls, or empty regions, which again can be intuitively understood from the maze overlay.

As signals are only generated in the walkable area of a maze, the walkable surfaces must first be identified, a process which runs counter to our goal of automation. In many cases, however, the walkable surfaces of a map are simply those oriented roughly horizontally, and so the normal to such polygons will be roughly vertical. In such cases, we automate the process of marking all polygons whose normal is within some angle θ of vertical as walkable, optionally pruning small polygons to remove trivial terrain features.

As noted above, the original GNG algorithm allows different implementations to choose their own stopping conditions. Additionally, in adapting the algorithm to our objective, we found it helpful to make some slight adjustments to the original design. We summarize our choices and modifications below.

Scaling Node Generation

Broadly speaking, the size of a graph needed to approximate a continuous space tends to increase with the size and complexity of that space. As the number of connected nodes in the graph grows, the time required for each iteration of the algorithm increases as well. One way to mitigate this slowdown is to reduce the rate at which new nodes are created based on the current number of nodes; as the graph grows, the number of iterations before creating a new node likewise becomes larger. Formally, we replace the value λ (that is, the number of iterations before a new node is created) by a function $\lambda(N)$ that is linear on the size of N; in the pages that follow,

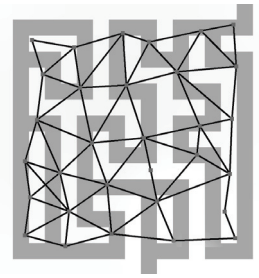
$$\lambda(N) = 100|N|.$$

Intuitively, the algorithm spends more time attempting to optimize a larger graph before adding another node. In practice, we find that this change tends to produce a sparser graph without sacrificing the approximation.

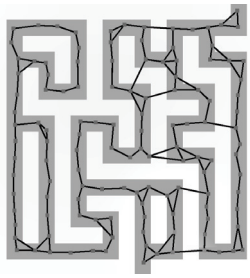
Stopping Criterion

The original GNG algorithm does not define a specific stopping criterion, leaving the choice up to the implementor. To that end, every 500 iterations, we generate a fixed number of random input signals. For each signal, we locate the nearest node in N and calculate the distance to that node; we then sum all of these distances. One can think of the resulting sum as a measure of how well the points of N cover the walkable area of the map. We call this sum the *coverage error*, denoted C.

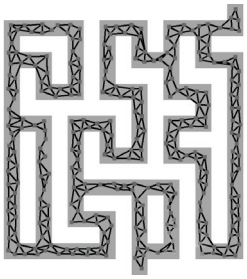
Initially, as the graph changes rapidly, the coverage error tends to decrease sharply; as the number of nodes increases and the graph becomes more settled, the rate of change of C declines. We use the rate of change of this sum as a stopping



(a) After 5,000 iterations.



(b) After 50,000 iterations.



(c) After 200,000 iterations.

Figure 1: Progressive iteration over the same graph. Graph (a) is a roughly uniform distribution of nodes over the entire maze area. Graph (b) is largely correct, but still has a number of "wall-crossing" edges, eliminated in (c).

condition; once the rate of change drops below a predefined threshold, execution halts. In other words, given two subsequent coverage errors C_i and C_{i+1}, execution halts if

$$|C_i - C_{i+1}| < \sigma \quad (1)$$

for some predefined *threshold coverage rate of change* σ. In our research, we found that the optimal value of σ varied from maze to maze.

Results

We ran the GNG algorithm on a variety of mazes, varying the algorithm parameters to search for a workable approximation. We present several notable results below, including the type of each maze, the algorithm parameters chosen, and the number of iterations required for the algorithm to terminate. The walkable area of each maze is a light gray; walls are white, and the graph produced is black.

Iterative Algorithm Progress

Figure 1 illustrates how a GNG graph develops across successive iterations on a braid maze; the three images show the development of a single graph through first 5,000, then 50,000, and finally 200,000 iterations. Note that the 1a is a largely even distribution of nodes, including many nodes inside walls. Such errors are often caused when a single node is "pulled" by signals on either side of a wall, particularly when the graph is still sparse. In 1b, the graph approximates the shape of the maze, though several edges still cross walls. At 1c, the number of nodes has increased further, producing a dense and properly-connected graph with no wall-crossing edges.

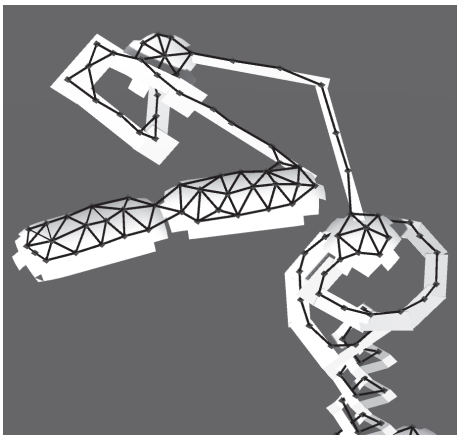


Figure 2: A 3D object with traversable area mapped.

Maze-Dependent Parameter Choice

We next considered the effect of the GNG algorithm on a maze whose length and width were both doubled, producing a maze with many more twists and bends than in preceding examples. Initially, parameters were set to the relatively low values of $\epsilon_\beta = 0.2$ and $\epsilon_N = 0.006$. The result was striking; while the algorithm required only a relatively small increase in the number of iterations, the result was effectively useless. As in the partially-complete graph in Figure 1a, nodes were distributed roughly evenly across the maze, with edges that had little resemblance to the underlying maze. This suggests that there is no single "best answer" set of parameters, even for a domain as restricted as maze navigation; the GNG algorithm must be tuned for its particular target environment. To produce a more meaningful maze required reducing the threshold value σ from 10^{-3}, to 10^{-6}. The resulting graph provides a solid basis for navigation, with no wall-crossing edges or other major undesirable features, but it required well over a million iterations to produce.

Three-Dimensional Figures

While our research primarily focused on two-dimensional mazes, we briefly considered navigating three-dimensional constructions as well. Given a three-dimensional polygon map, we used automated generation of the normals to each polygon to determine which polygons were approximately horizontal, identifying any such polygons as "traversable," and ran the GNG algorithm. Figure 2 shows a typical result; while the figure may be difficult to interpret in two dimensions, it includes a long vertical spiral at the right, followed by several narrow catwalks and an ending half-cylindrical area. The GNG algorithm generally correctly traverses the spiral and catwalks, filling the wider traversable areas with small sub-graphs. There remains a slight tendency for the algorithm to cut across corners, which might have unfortunate consequences for anyone relying on it to traverse the vertical spiral; we suggest some remedies for this problem in the next section.

Future Work

While the preceding tests suggest that the GNG algorithm can be configured to produce results in a variety of mazes, there remain a number of ways in which its behavior might be improved. We consider several of these below.

Wall-Crossing Edge Elimination

It seems reasonable that any path in a navigation mesh should also be traversable in the original polygon map, and so any wall-crossing edges produced by the GNG algorithm must be eliminated. Unfortunately, as the relative thickness of walls to corridors decreases, the base GNG algorithm becomes increasingly likely to produce exactly such edges. In general terms, to avoid wall-crossing edges, the density of nodes must be so great that every node is closer to other nodes in the same "hallway" than to other nodes across the wall from it. In an environment with infinitely-thin planar walls, the required density of nodes would become infinite.

One modification of the GNG algorithm to resolve this problem would be to accept the presence of wall-crossing edges during the algorithm's run, focusing on a desirable distribution of nodes rather than elimination of all undesirable edges. The final graph could then be checked for collisions between edges and non-walkable polygons, with any wall-crossing edges culled. We anticipate that the result would produce a useful, sparser map after a relatively low number of iterations.

Disjoint Graph Detection

A second concern is that the GNG algorithm might terminate while areas that are connected in the original polygon map are still disconnected in the navigation mesh. While such a mesh might fulfill all other stopping criteria, it would obviously be undesirable for agents to fail to reach clearly-accessible areas. We thus suggest that the stopping criteria for the algorithm be expanded to include graph connectivity.

Need-Based Node Creation

As maze complexity and size increases, the number of nodes required to fully approximate the maze increases dramatically. As noted above, our implementation waits for a number of iterations linearly proportionate to the number of existing nodes before introducing an additional node; equivalently, the number of iterations required to produce a given graph is proportionate to the square of the number of nodes in that graph. For larger graphs, this may be too steep of an increase. A function $\lambda(N)$ whose growth was logarithmic or exponential in the size of N, rather than linear, might perform better.

Another alternate approach would be to base the rate of node creation on the rate of change of the coverage error C; intuitively, while the value of C is changing rapidly, the graph is not well fit to its space and so could benefit from additional nodes. Once the rate of change of C drops and the graph stabilizes, the rate of node creation should drop to allow better fitting of the existing nodes.

Benchmarking

A major need at present is some form of benchmarking to compare the GNG algorithm to existing alternatives on a variety of reasonable polygon maps. To this point we have shown only that the algorithm can reliably produce appropriate-seeming navigation meshes; it remains to be seen how its performance compares to alternatives.

Conclusion

We have demonstrated that, with only minor adjustments, the Growing Neural Gas algorithm produces plausible waypoint graphs for two-dimensional mazes. Further, initial evidence suggests that the algorithm would also produce appropriate graphs for three-dimensional spaces. We suggest that further investigation be made to determine the whether the GNG algorithm is a plausible alternative for navigating artificial spaces.

References

Doherty, K.; Adams, R.; and Davey, N. 2005. Hierarchical growing neural gas. In *Adaptive and Natural Computing Algorithms*. Springer. 140–143.

Fritzke, B., et al. 1995. A growing neural gas network learns topologies. *Advances in neural information processing systems* 7:625–632.

Hale, D. H., and Youngblood, G. M. 2009. Full 3d spatial decomposition for the generation of navigation meshes. In *AIIDE*.

Hale, D. H. 2011. A growth-based approach to the automatic generation of navigation meshes.

Hart, P. E.; Nilsson, N. J.; and 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.

Holmström, J. 2002. Growing neural gas–experiments with gng–gng with utility and supervised gng.

Oliva, R., and Pelechano, N. 2013. Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments. *Computers & Graphics* 37(5):403–412.

Prudent, Y., and Ennaji, A. 2005. An incremental growing neural gas learns topologies. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, 1211–1216. IEEE.

Qin, A. K., and Suganthan, P. N. 2004. Robust growing neural gas algorithm with application in cluster analysis. *Neural Networks* 17(8):1135–1148.

Tencé, F.; Gaubert, L.; Soler, J.; De Loor, P.; and Buche, C. 2013. Stable growing neural gas: A topology learning algorithm based on player tracking in video games. *Applied Soft Computing* 13(10):4174–4184.

Thurau, C.; Bauckhage, C.; and Sagerer, G. 2004. Learning human-like movement behavior for computer games. In *Proc. Int. Conf. on the Simulation of Adaptive Behavior*, 315–323.