Graph Traversal for Procedural Fantasy Map Generation

Kenny Hunt
khunt@ulwax.edu
The University of Wisconsin La Crosse
La Crosse, Wisconsin

Figure 1: Random fantasy map generated using graph-based generation.

ABSTRACT

This paper describes a novel application domain for teaching graphs, graph traversal, and spanning tree algorithms. A technique for generating randomized fantasy maps backed by a randomized graph data structure is presented. Terrain elevation is generated via bread-first search, roads are generated using Prim's minimum cost spanning tree, and continents are segmented from surrounding seas using bread-first search as well.

CCS CONCEPTS

• **Applied computing** → **Interactive learning environments**; • **Social and professional topics** → **Computer science education**; • **Theory of computation** → **Graph algorithms analysis**.

KEYWORDS

fantasy map, graph, graph traversal, spanning tree, computer science education

1 INTRODUCTION

Common graph algorithms taught in either a CS2 or Analysis of Algorithms course include breadth-first traversal, depth-first traveral, and Prim's minimum cost spanning tree algorithm. These algorithms are typically motived by their application in computer networks, the flow of water through a system of pipes [6], electronic circuit design [1] and, more recently, map-based path-planning [4]. While these applications are extremely important within their field, they tend to be difficult to engage with and visualize in a way that captures the nature of the algorithms that they motivate. How is the edge-cost of a resistor depicted, for example, when visualizing the flow of electrons through a circuit; or how is the flow of water through a piping system visualized in terms of volume? This paper describes the use of graph traversal algorithms to generate randomized fantasy maps in such a way that the algorithmic data can be more readily conveyed and that is likely more engaging to students than the flow of petroleum through an oil refineries piping system.

2 TERRAIN GENERATION WITH BREADTH FIRST TRAVERSAL

We seek to generate a map of arbitrary width, W, and height, H, without concern for cartographical map projections. First, we outline how to generate a graph that serves as a model of the map. We then describe how to traverse the graph in order to generate elevation data.

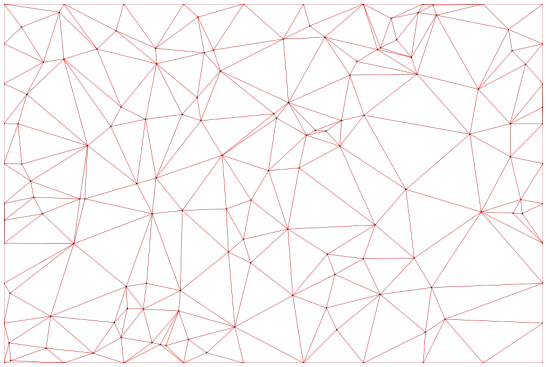
2.1 Graph Generation

As a first step, we generate a set of N two-dimensional coordinates. To ensure that later steps in the processing pipeline yield aesthentically pleasing results, $k << N$ of these points are distributed uniformly on the bounding box of the mapping space and each corner of the bounding box is explicitly represented. We then generate $N - k$ random coordinates, uniformly distributed across the mapping space.

These N coordinates are converted into a graph through Delaunay [1] triangulation. The vertices in this graph can be understood as random locations in the world and the edges in the graph denote line-of-sight connections to the closest locations in a variety of directions. Each triangular region of the map represents a planar

approximation to the terrain assuming that each vertex is associated with an elevation. This graph serves as the foundational data structure for the fantasy map.

(a) Point Cloud

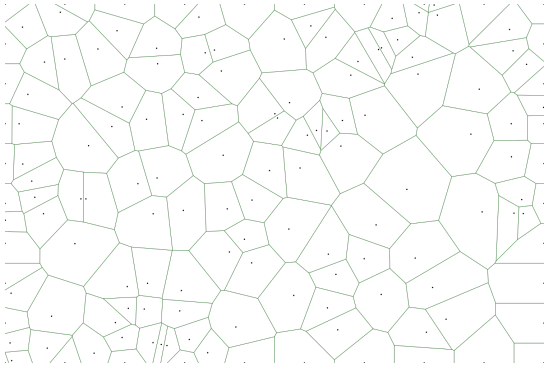


(b) Delaunay Triangulation

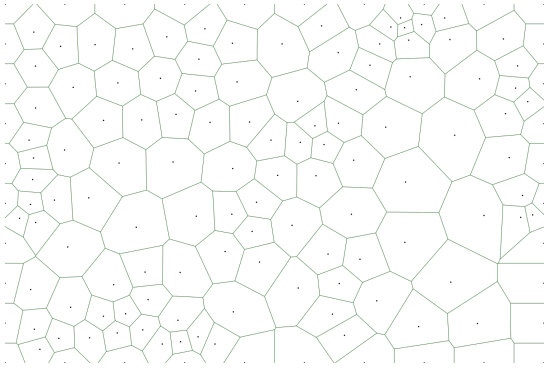
Figure 2: Delaunay Triangulation

The random generation of vertices often results in regions of the map being overly dense compared to others and so we perform a relaxation of the map coordinatess. Towards this end we compute the Voronois diagram [5] in order to obtain a set of polygons such that each coordinate point is associated with a single polygon. Several iterations of Loyd's relaxation algorithm [2] is then employed. Relaxation is a straightfoward algorithm that computes the centroid of each polygon in the Voronois diagram and then relocates the coordinate corresponding to that polygon to the centroid. This produces more evenly sized regions of more uniform spatial distribution throughout the map. In our process, the k edge points are not included in this relaxation algorithm. If we allow these points to move from their original location, the entire graph tends to collapse inwards towards the map center and distort the map edges.

We now have a Delaunay graph of N semi-randomly distributed vertices formed by computing the Delaunay trianguation of those vertices. This graph serves as the backing data structure for all of the operations that follow.



(a) Voronoi Diagram



(b) After Relaxation

Figure 3: Relaxation

2.2 Terrain Elevation

For each vertex in our graph we generate an elevation such that the elevation values are reasonable approximations to physically real topographies. We generate the elevations by choosing a single graph vertex and applying a generating function that originates from that starting point and then performs a bread-first traversal to modify the elevation settings of each visited vertex.

The algorithm of Algorithm 1 accepts a starting vertex and a visitor function. The algorithm proceeds by applying the visitor function to each vertex in the graph (at least each vertex to which the function should be applied) such that vertices closer-to the starting vertex are visited before vertices farther away. In this code, proximity to the starting vertex is given not in Euclidean distance but in the length of the path from root to vertex.

The visitor is given here as a function such that all logic related to visiting a vertex is off-loaded into this visitor function. The visitor function returns a boolean flag denoting whether the just-visited vertex should be treated as a dead-end in the traversal or whether graph traversal should continue. We also augment the breadth-first search algorithm with a pruning mechanism that allows us to ignore some vertices in the graph. This feature is implemented by a predicate function, $CanVisit$. We will give an example in Section 4 of how this predicate can be used to speed up certain map-processing traversals.

Algorithm 1: BreadthFirstTraversal($V_{start}, CanVisit, Visitor$)

```
let Q be a Queue;
let Visited be a Set;
let Parents be a Map;
Parents[V_start] ← null;
Q.add(V_start);
while Q is not empty do
    let V_current be Q.remove();
    let P_current be Parents[V_current];
    Visited.add( V_current);
    let shouldContinue be Visitor( V_current, P_current );
    if shouldContinue then
        for let N of the neighbors of V_current do
            if CanVisit(N) and N is not in Visited and not in
            Q then
                Q.add(V_current);
                P[N] ← V_current;
            end
        end
    end
end
```

Algorithm 1 simply provides the traversal framework but doesn't describe what it means to *visit* a vertex. For our purpose, when a vertex is *visited*, the elevation of the vertex is modified according to a generating function. Our generating function creates a single, randomized yet plausible region of terrain that is centered at a starting location and then tapers off towards zero in a roughly circular pattern. The generating function is parameterized on the size, overall elevation, and roughness of the affected region. We choose to model this terrain element as a radially symmetric (apart from the arbitrary injection of randomness) exponentially decay. Given the elements starting height H_{start}, ending height H_{end}, radius R, and variance V we define a function that generates a sequence of elevations E such that the k^{th} elevation is given as

$$E_k = E_{k-1} \times (1 + V \times (\frac{1}{2} - rand()) \times FACTOR \quad (1)$$

where

$$FACTOR = e^{\frac{ln(H_{end}/H_{start})}{R}} \quad (2)$$

This discrete function injects white noise riding atop a continuous exponential decay function of the form $H_{start} * SLOPE^k$. This randomness is sufficient to generate plausibly realistic elevation profiles of regions where, for example, a large starting height and smaller radius will be seen as a mountain and smaller starting heights with a large radius will be seen as a continent. Also, the generator can be controlled via the variance, radius, height start and height stop parameters to generate what might be considered continents or smaller islands. Figure 4 compares a profile of our noisy discrete terrain generator against a smooth exponential decay. In this figure, $H_{start} = 50$, $H_{stop} = 2$, and $R = 50$.

Although beyond the scope of this paper, a more thorough map-generating system will include data layers such as average annual temperature and average annual rainfall. These additional layers would later be combined to generate appropriate biome classifications for map regions.

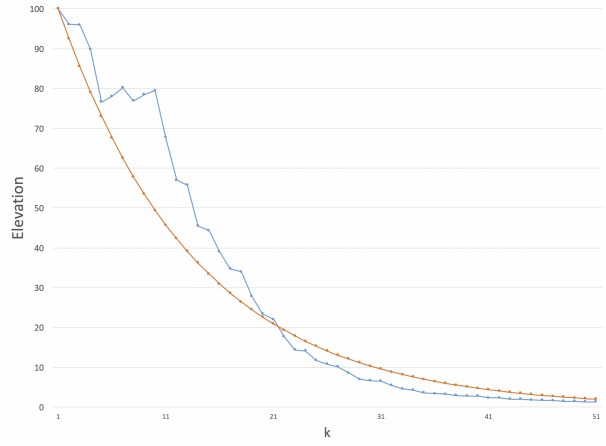


Figure 4: Terrain profile from 100 to 2 in 50 steps

Figure 5 shows an elevation map that is the result of applying several terrain generator functions layered atop each other. We have selected several small-height, large-radius functions to model the continental bodies while applying several large-height, small-radius functions to model mountains. The blue regions have elevations that fall beneath a user-selected *sea-level* elevation while lighter grayscale regions denote the highest elevations and black denotes the lowest above-sea-level elevations.

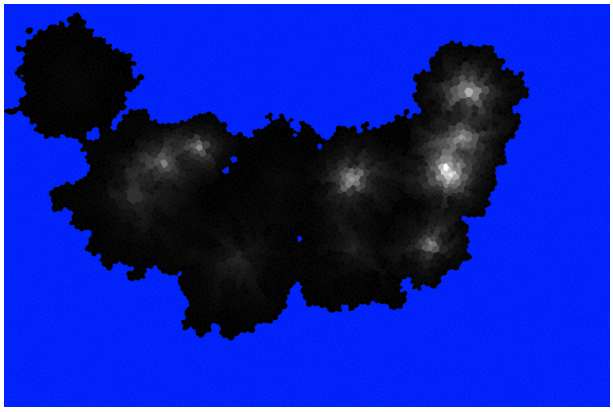


Figure 5: Elevation map

3 ROAD GENERATION

Having generated elevation data we now need to identify cities and towns, connecting them with a plausible network of roads. These roads should be realistic in the sense that they will not typically ascend straight up mountainous slopes nor move through underwater regions. Large cities should also tend to be central spokes in any transportation network while smaller towns would likley have correspondingly fewer road connections. This section briefly

describes the selection of city locations after which we describe the use of Prim's algorithm [3] for computing a plausible road network.

3.1 City Locations

We first assume that the graph contains layers of data that denote properties such as elevation, average annual temperature, average annual rainfall and, potentially, resource allocations such as mineral wealth or fecundity. For each vertex having an elevation above some arbitrarily chosen water-level, we assign a *desirablity* rating. The *desirablity* rating takes into account rainfall, temperature, elevation and even whether or not a vertex is adjacent to a body of water. The vertices are then ranked by *desirablity* and filtered in such a way that cities are not located overly close. The output of this process is a ranked ordering, $Cities$, of vertices such that higher *desirability* values denote larger cities.

3.2 Road Network

We now generate a plausible road network by iterating over every vertex in $Cities$ in ranked order such that larger cities are processed before smaller cities. For each city, we apply Prim's algorithm for finding a minimum spanning tree. However, with each of these applications we do not seek to generate a minimum spanning tree of the entire graph but proceed only until reaching a terminating condition.

Prim's algorithm, shown in Algorithm 2, is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. Interpreted in the context of our mapping application, each edge in the Delaunay graph is a potential road such that the weight of that edge is determined by the difficulty of travel (described later). Prims algorithm will then find the easiest way to get from a starting city to every other vertex in the graph; a minimum cost spanning tree. However, as noted earlier, we terminate the algorithm once we generate a least-cost path between the starting city and some other location of interest; either another city or an already-existing road in our ever-growing road network.

In Algorithm 2, the $Visited$ set holds all vertices for which there is already a computed road from the V_{start} city. The *Cost* map is keyed on vertices and holds the cost of traveling to that location from the V_{start} city. *CostFn* is a function that computes the cost of moving between two vertices in the graph. The cost is a combination of the length of the edge, the grade (slope) of the edge, and the absolute elevation since mountainous roads are less desirable than roads at lower elevations. *PQ* is a priority queue holding un-reached vertices keyed on *Cost*. When selecting a path to extend we are greedily selecting the least-cost path at every loop iteration.

The $Visitor$ function accomplishes two objectives. First, it offloads any application specific work out of the generalized Prim's algorithm and into the $Visitor$ function itself. Secondly, the $Visitor$ function is responsible for terminating Prim's algorithm once a sufficient number of sites have been explored. For our application, the $Visitor$ function keeps a network of roads along with unvisited cities. Whenever any call to Prim's algorithm visits a vertex that is either an unvisited city or already on the road network, the algorithm is signaled to terminate.

The tree itself is held in the local *Parent* map which is keyed on graph vertices having values that are the parents of the keys. In this representation, a road can be generated by following a chain of parents from the far end of the road back to the V_{start} city.

Algorithm 2: Prims(V_{start}, $Visitor$, $CostFn$)

```
Visited ← new Set();
Parent ← new Map();
Cost ← new Map();
PQ ← new PriorityQueue();

Parent[V_start] ← null;
Cost[V_start] ← 0;
PQ.add(V_start);

while PQ is not empty do
    V ← PQ.remove();
    Visited.add(V);
    P ← Parents[V];
    shouldContinue ← Visitor(V, P);
    if shouldContinue then
        for let N of the neighbors of V do
            if N is not in Visited and not in PQ then
                Cost[N] ← Cost[V] + CostFn(V, N);
                PQ.add(V);
                Parent[N] ← V;
            end
        end
    else
        return Parent
    end
end
return Parent
```

Figure 6 shows a low resolution continental map. The elevations are given in grayscale where higher elevations are lighter and lower elevations are darker. The cities have been connected via a road network generated by repeated application of Prim's algorithm. The roads prefer to avoid steep terrain and also avoid high elevations. In order to prevent roads from crossing through water, the *CostFn* returns ∞ if either one of an edges vertices are below sea level.

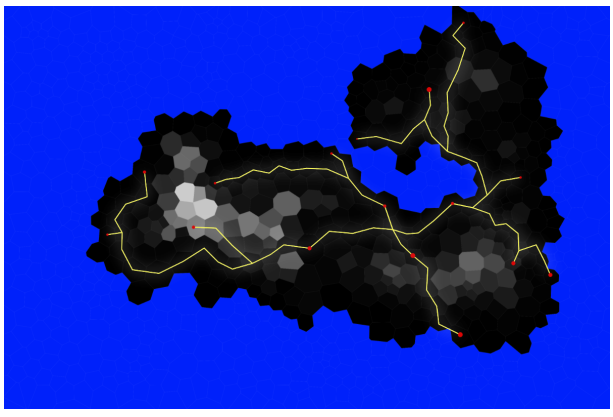


Figure 6: Road network for a low resolution continent

4 SEGMENTATION WITH BREADTH FIRST TRAVERSAL

We now seek to identify continents so that we can further segment those land masses into larger sub-regions according to either geopolitical factors (i.e. nation-states) or geophysical factors (i.e. biomes). We can accomplish this segmentation by using the breadth first search traversal of Algorithm 1 and using the pruning predicate $CanVisit$ to filter out underwater vertices.

Algorithm 3: Continents($Vertices$, $Heights$)

```
Continents ← new Array();
Visited ← new Set();
CanVisit ← (n) => Heights(n) > 0;

for let V of Vertices do
    if CanVisit(V) and V not in Visited then
        Continent ← new Set();
        BreadthFirstTraversal( V, CanVisit, (n, p) => {
            Continent.add(n);
            Visited.add(n);
            return true;
        });
    end
end

return Continents;
```

5 CONCLUSION

Students learning graph traversal will be able to visualize the output of their work in an engaging manner that, while not as useful as electrical circuit diagrams or piping designs, is better able to capture the nature of the traversal algortihms themselves. The algorithms described in this paper have been implemented in JavaScript and have a reasonable real-time performance for graphs on the order of |V| = 15,000.

While the output on graphs of such relatively low resolution are not aesthetically pleasing, this work can be extended in a large number of dimensions. The inclusion of geophysical properties, geopolitical properties, as well as depth shading and color blending modes will produce high quality maps. An example is given in Figure 1 at the beginning of this document. The image was generated without human intervention.

REFERENCES

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[2] S. Lloyd. 2006. Least squares quantization in PCM. *TIT* 28 (01 2006), 129–137.

[3] R.C. Prim. 1957. Shortest Connection Networks and Some Generalizations. *Bell Syst. Tech. J.* 4 (01 1957), 53–57.

[4] James D. Teresco, Razieh Fathi, Lukasz Ziarek, MariaRose Bamundo, Arjol Pengu, and Clarice F. Tarbay. 2018. Map-Based Algorithm Visualization with METAL Highway Data. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 550âĂŞ555. https://doi.org/10.1145/3159450.3159583

[5] G. Voronoi. 1908. Nouvelles Applications des ParamÃĺtres Continus Ãă La ThÃĭorie des Formes Quadratiques. *Journal fÃijr die Reine und Angewandte Mathematik* 134 (01 1908).

[6] Mark Allen Weiss. 2005. *Data Structures and Problem Solving Using Java (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.