MSc Computer Animation and Visual Effects
University of Bournemouth

Connected Attribute Labelling in Houdini

Parker Britt
January 2025

A project submitted for Computer Graphics Techniques

Contents

1. Introduction

1.1 Problem statement, Objectives, and Applications

This project aims at designing a method to solve continuous indexing of nodes within a finite graph based on the connectivity of vertex values. The method indexes unique connected node groups, which can also be referred to as "islands", in the graph and assigns a continuous integer identifier. The island indices are ordered sequentially based on the lowest point number in the set, starting at 0 and ending at $n - 1$, where n is the total number of islands. Vertices are considered to share an island index if two vertices share a primitive and the value associated with each vertex (attribute) is greater than a given threshold. This threshold is important to provide a binary separation between background and foreground values. This approach provides a deterministic way of classifying and categorizing data within a graph, with a specific focus in the field of computer graphics for processing geometry, and it's associated attributes. Uses for this are broad, in the field of FX this could be leveraged to index source islands of particles, driving characteristics like velocity, radius, etc. In the field of procedural modelling, the island index could be used as a seed for different foliage characteristics when scattered onto a surface. Previous approaches to similar problems focus on the fields of discrete mathematics, graph theory, filling algorithms, and more. In this paper, I aim to explore how each of these approaches solve their challenges and use this research to inform my own approach.

2. Background Research

2.1 Graph Theory

While the problem I present is referred to as a graph, it's important to note this is just one of many ways to model geometry data, however within the context of connectivity with geometry, graph theory provides a very appropriate way to model our data. Graph Theory is the study of graphs, one of the core concepts of discrete mathematics. The initial model for graph theory was introduced in the book *Graph Theory with Applications by* J.A. Bondy and U.S R. Murty (West 2001), where the authors describe a graph as "... a diagram consisting of a set of points together with lines joining certain pairs of these points." (Bondy and Murty 1976) These points and lines in graph theory are sometimes called nodes and edges, but in our case, using the vocabulary of points and vertices provides a convenient parallel with the concepts used in computer graphics. In the text, Bondy and Murty (1976) demonstrate how graphs can be modelled using formulas and sets, such as $G = (V(G), E(G), \psi_g)$ to represent complex connections using maths. For this project, graph theory provides a common language that we can use to describe our problems mathematically.

2.2 Traversal Algorithms

In order to effectively index our connected vertices, we first need to traverse the graph, progressively collecting and evaluating data. Traversal algorithm serve an important place in computer graphics, from finding discrete pieces of data, colouring an area, or solving a maze. Making these algorithms are essential for a lot of the work that we do. Graph traversal comes in many forms, two common approaches of which are breadth first search (BFS) and depth first search (DFS). Both of these techniques are commonly used in computer graphics to operate on images and geometry. You can see them used in raster graphics, taking the form of the popular flood fill algorithm. André (2005) defines it as, “a generally recursive algorithm that finds connected regions based on some similarity metric, used mostly in filling areas with colour in graphics programs”. He goes on to point out that the algorithm can be used in other applications, like the video game, Tetris (André 2005). One of the algorithm’s earliest definitions is by Newman and Sproull (1973) in *Principles of interactive computer graphics*. In the chapter *Interactive Raster Graphics* they mention an algorithm that, given a defined boundary, can be used to fill the interior with said boundary (Newman and Sproull 1973). This is a short section in an otherwise long book, defining the simple recursive algorithm. It describes first fetching the intensity of the current pixel, then setting itself as well as its neighbouring 4 pixels on perpendicular axis to the new value, afterwards recursively calling the function again (Newman and Sproull 1973). This recursive approach uses DFS, it first fills all point to the right, then left, then up and down. Many iterations and improvements have sprung from this work over the last 50 years, linear flood fill, scribble flood fill, bounded flood fill, snake flood fill and sequence flood fill (André 2005) just to name a few. Some use differing data structures to reduce the memory overhead, reduce the recursiveness, and transition to a more linear approach (André 2005). Different data structures also allow the use of BFS as well as DFS, leading some to compare the performance of the two approaches. Sadik et al. (2010) analyses graph traversal algorithms in the context of maze solving, in doing so they find traditional DFS flood fill to be more memory intensive and less appropriate for their task when compared to BFS. This is especially applicable to larger data sets, “If the maze is large then Floodfill should be avoided. BFS will give satisfactory performance in this scenario.” (Sadik et al 2010)

2.3 Connected Component Labelling

During developing my method, I independently implemented an approach to solving the problem of identifying and indexing islands on the basis of connectivity. It was only after implementation that I found my problem aligns with the well documented research of connected component labelling. Because of this, my method was not influenced or developed by referencing existing research on the subject of connected component labelling. Connected component labelling is the process of sorting components based on their connectivity. These components can take many forms. They can be pixels in a raster image, nodes in a graph, or point attributes on geometry. The field of connected component labelling thoroughly encapsulates the aims of this project.

3. Implementation

3.1 Graph Theory

Looking at the field of graph theory provides us a consistent language to model our problem. In order to model our data with graph and set theory, first we must define our graph, G.

G is defined as the set $(V(G), E(G), \psi_G, \phi_G)$ consisting of a non-empty set $V(G)$ of vertices, a set $E(G)$, disjoint from $V(G)$, of shared islands. Where ϕ_G is as a function that associates with each vertex of G an unordered pair of primitives of G and ψ_G has the similar association, mapping edges to their corresponding vertices.

Then we need to check mesh connectivity between points by confirming that the intersection between the two primitive sets is not empty.

$$\phi_G(v_1) \cap \phi_G(v_2) \neq \emptyset$$

However, to be considered part of the same island, we also require both vertices to share an attribute greater than our threshold, t. We can call this a “valid” or a “foreground” point

$$Valid(v_1) = attr(v_1) >= t$$

So v_1 and v_2 are considered sharing an island if $g(v_{1,} v_2)$ is true, defined by

$$g(v_1, v_2) = \phi_G(v_1) \cap \phi_G(v_2) \neq \theta \wedge Valid(v_1) \wedge Valid(v_2)$$

Since our vertices connect if they share an island, where our edge is represented by e, we can say

$$\psi_g(e_n) = v_1, v_2 \text{ if } g(v_1, v_2)$$

In summary, the aforementioned formula models our graph, and the formula $g(v_1, v_2) = \phi_G(v_1) \cap \phi_G(v_2) \neq \theta \wedge Valid(v_1) \wedge Valid(v_2)$ allows us to identify whether two neighbours share a foreground island.

3.2 DFS and BFS

For each step of development, I made sure to consider the complexity of the data set. Since my method is designed for implementation in Houdini, it will be performed on geometry meshes. These meshes can easily range from containing a couple of hundred points to tens of millions of points. This is why I needed to consider this scale in every process. A single unoptimized time or space complexity could slow down my computation exponentially. My first approach to traversal used a recursive approach similar to the first flood fill algorithm. I first input a seed point as an argument, the function would perform indexing operations on itself, and its valid neighbours, then the function would call itself for each valid neighbour, using the neighbours as arguments. This presents a significant issue when operating on such a large number of points. Most of these calls have to wait on the execution stack until the stack is fully unwound, when the entire island is discovered.

Assuming the execution stack is 8192 kilobytes (KB), or 8 mebibytes (MiB) it would likely only be able to hold roughly four to five digits of function calls. Each point is represented by the GA_Offset type, when broken down as shown in fig. 2, it evaluates to a long, which is 32 bytes on AMD x64 CPUs. Considering Houdini only supports hardware with a x64 instruction set architecture (SideFX 2024), we can ignore the second branch and assume each point is represented by 32 bytes in all situations. If each call on the execution stack has to store at least one point offset, then in the case of a one million points island we would have to store at least 32 MB in our roughly 8MB buffer. Of course, we don't know how large islands will be, so when working with geometry that's in the seven to eight digits, we can't risk a stack overflow occurring from an island that contains too many points. To solve this, we can switch to an iterative approach using the std::stack class, a dynamically resized container that uses the heap, instead of the execution stack. Since the heap has a wider access to system memory, this will often have many times the capacity of the execution stack, causing the chances of an overflow to be significantly lower.

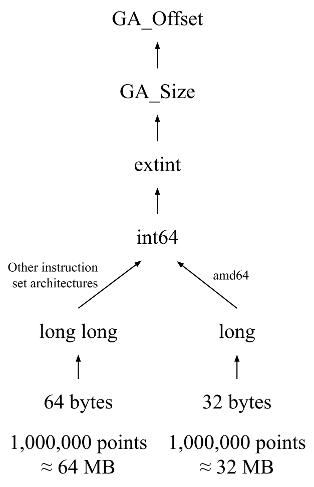

Fig. 2: typedef heirarcy

Instead of using recursion, this iterative approach loops through a std::stack buffer, which is first given a seed point to initiate the loop. Inside the loop, we remove the top item from the stack, treating it as our current point. We then analyse the neighbours of this point, and perform our necessary comparisons. However, instead of recursively calling the fill function with the neighbouring points, we add them to our stack. After the loop completes, the next item pulled from the stack will be our neighbouring point, repeating the cycle. The transition from a recursive to an iterative approach allows for more flexibility with which data structure to use for our buffer.

Std::stack is a LIFO (last-in, first-out) data structure, meaning the last contributed point will be the next computed point. When in the context of flood fill, LIFO constructs a depth first search algorithm, meaning it will continue adding points to the container until it can't find any more, only then will it unwind to resolve the waiting points. In contrast, a std::queue is a FIFO (first-in, first-out) data structure, providing a breadth first search. This method empties the buffer from the bottom and adds to the top. By using this approach, I found a significant reduction in points held in memory at a given time. As noted by Sadik (2010) "If the maze is large then [DFS] should be avoided. BFS will give satisfactory performance in this scenario."

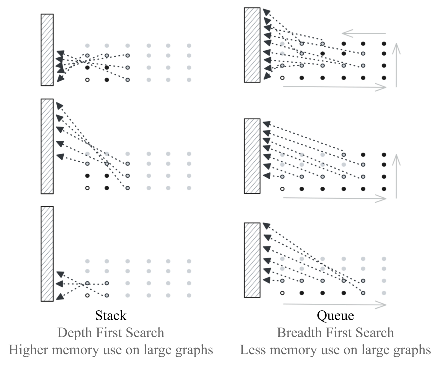
Stack
Depth First Search
Higher memory use on large graphs
Queue
Breadth First Search
Less memory use on large graphs

Fig. 01: Stack vs Queue traversal simulation

3.3 Multithreading

When using low level nodes in Houdini, it’s imperative that they complete their operations quickly. The designer of these nodes can’t know how many times the user may use them and for what data set. Users will often have chains of hundreds of nodes, each slowing down the process as a whole. Additionally, each node can contain many child nodes, each of which could contain more nodes inside of them. Any speed up you can make to your node makes a big difference in the context of a complex network. To improve speed and take full advantage of hardware performance, I spent a lot of time drafting various algorithms for multithreading, considering locking, time complexity, and the space complexity of memory usage. In the end, I decided that given the time I had, implementing multithreading wouldn’t be feasible, and polishing existing features would provide a better user experience. I’ve included a detailed explanation of these methods in the appendix.

Appendix

A.1 Multithreading First Approach

My first approach to multithreading involved two parts, the runner and the filler. The runner would execute in the main thread, iterating over every point in the mesh (just like in the single threaded approach), when the runner found a point where $Valid(v)$ the main thread would spawn a new thread in which the filler would execute an n-way flood fill on the connected island. After which all the found points are returned to the main thread. This approach had a few issues to work out.

The first of which is "what does the runner do while the filler is indexing the island?" In my single threaded approach, the runner skips over already found points. They do this using a shared set containing already traversed points. But if the runner and the filler are executing concurrently, then the runner may find a connected point before the filler and spawn a new thread (and filler) to populate it, leading to many duplicate sets and wasted computation. We could have the runner wait for the thread to finish, but this would be the same as the single threaded approach. My solution to this problem is twofold, I first mitigate the amount of duplicate sets created, then perform a deduplication step at the end. To reduce the created duplicate sets, I use a state machine (fig. 3). The runner starts with an active state, as it's traversing, if it finds a valid point it creates a filler and sets its state to locked. While in this state, the runner can no longer create new points. Its state is reset to active once it finds a non-valid point. Consecutive valid points will always be in the same island, so by adding this logic we avoid most, but not all, duplicate states. The next step is deduplication, after the runner and all the threads have finished, we take all the resulting sets and compare them. The naive approach to this would be to loop over each set, comparing every point in that set with every point in all the other sets. This would have an extremely inefficient complexity of, $O(S^2 * P^2)$ where S is the number of sets and P is the number of points in a given set. By using unordered sets, which have a query complexity of $O(1)$ we can use a shortcut. Because each set is sorted, this means the first value of each set can be used to check if the sets are equal, $A_1 = B_1$ bringing the complexity down to $O(S^2 * 1)$ or $O(n^2)$ where n (the number of sets) is significantly smaller than the number of points.

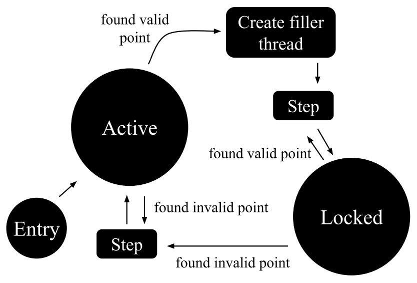

Fig. 3: Runner state machine

The second issue is, how do we return the points the filler found to the runner in real time. If the runner doesn't know what points have been traversed, it will end up creating many duplicate sets, even with our concurrent points optimization from earlier. The solution would be a shared data set, but this introduces the problem of locking. This data set would be getting written to at every step by both the runner and the fillers, locking it every time would slow things down, almost nullifying the benefit of threading. We can fix this by utilizing two convenient properties of STL data types, first "C++ standard library implementations are required to avoid data races when different elements in the same sequence are modified concurrently." (C++ 2025). Meaning, multiple threads can write to different elements without locking. Another property of STL containers is the ability to read a single container from multiple threads at the same time. This means the main thread can use a nested dynamically resizeable container, for example, std::vector<std::vector>. The outer container belongs to the main thread, while the child containers belong to the filler threads. Only the main thread can modify the outer vector by adding new elements when a new thread is created. It can pass a pointer to the filler threads, who are then able to write to the child threads without worrying about race conditions.

A.2 Multithreading Second Approach

This approach was born from technical constraints in the HDK multithreading library, making the dynamically created threads that the first approach required impossible. Due to constraints on time and resources, I'm unable to document the second approach in sufficient detail. However, I can provide a brief overview. While the HDK multithreading library is unable to create dynamic threads, it is able to create an arbitrary number of threads simultaneously, but not sequentially. It does so, locking the main thread in the process. To work around this constraint, I first segment the mesh into sections based on ranges or point offsets rather than region. Each of these batches are processed by a different thread, each with their own single threaded runner and filler dynamic. This creates discontinuous islands, where a single island is processed by 2 or more threads. The next step is to find a way to stitch and order these islands. This proved to be extremely complex to do with any acceptable time complexity. Most solutions I designed would offset the threading efficiency with an inefficient border matching algorithm. The solution I came up with involved using adjacency list graphs structures, along with various addition data structures, to efficiently parse and compute the border connected islands.

Bibliography

André, P., 2005. *Intelligent flood fill or: The use of edge detection in image object extraction. Doctoral dissertation*, Southampton: University of Southampton.

Bondy, J.A. and Murty, U.S.R., 1976. *Graph theory with applications.* Macmillan, *290*.

CppReference, 2024. *Stack*. No place: CppReference. Available from: https://en.cppreference.com/w/cpp/container/stack [Accessed 27 January 2025]

CppReference, 2024. *Queue*. No place: CppReference. Available from: https://en.cppreference.com/w/cpp/container/queue [Accessed 27 January 2025]

He, L., Chao, Y., Suzuki, K. and Wu, K., 2009. Fast connected-component labeling. *Pattern Recognition*, 42 (9), 1977–1987.

He, L., Ren, X., Gao, Q., Zhao, X., Yao, B. and Chao, Y., 2017. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition,* 70, 25–43.

Law, G., 2013. Quantitative comparison of flood fill and modified flood fill algorithms. *International Journal of Computer Theory and Engineering*, 5 (3), 503–508.

Newman, W.M. and Sproull, R.F. eds., 1979. *Principles of interactive computer graphics*. McGraw-Hill, Inc., 253.

Sadik, A.M., Dhali, M.A., Farid, H.M., Rashid, T.U. and Syeed, A., 2010, October. A comprehensive and comparative study of maze-solving techniques by implementing graph theory. In *2010 International Conference on Artificial Intelligence and Computational Intelligence.* IEEE, 1, 52–56.

SideFX, (n.d). *System Requirements*. Toronto: SideFX. Available from: https://www.sidefx.com/Support/system-requirements [Accessed 27 January 2025].

The Standard C++ Foundation, (n.d). *C++11 Standard Library Extensions — Concurrency*. Washington: The Standard C++ Foundation. Available from: https://isocpp.org/wiki/faq/cpp11-library-concurrency [Accessed 27 January 2025]

West, D.B., 2001. *Introduction to graph theory*. Prentice Hall, 2.