A Brief Assessment of Breadth First Search Implementations

Lists vs. Matrices

Wesley R. Abbey
Zhishan Guo

University of North Carolina at Chapel Hill

April 2015

Preface

This brief paper is a very informal assessment of the different implementations of graphs running the breadth first search algorithm. We will specifically be going over the implications of using a linked list vs. a matrix when attempting to determine adjacent vertices within a graph. Each method has different benefits in regards to memory, speed, and overall usefulness within different situations. We will be observing the difference in speed and memory usage during each situation and then attempt to create a guideline for when the most optimal time to use a matrix instead of a linked list would be and vice versa. This was initially an assignment given out by Professor Zhishan Guo. There was minimal direction and although I would have liked to do a lot more, specifically implementing and assessing weighted graph algorithms as well, there was a time constraint. It is expected that this will be expanded on much further with different data structures and algorithms in the near future.

Introduction

This is an assessment of the basic implementation of the Breadth First Search algorithm. All code was written using Javascript and it is all the original work of the author. The *CLRS* and the *Data Structures in Java* book by Alan Weiss were used as basic resources. The algorithm being evaluated was the breadth first search algorithm. This is used to determine the distance of all vertices within a graph from a given source vertex. Farther vertices are evaluated last while the vertices closest to the source are evaluated first. Each vertex is an object with specific properties that describe it. These properties include the **distance**, which is the number of vertices

that must be traversed from the source vertex in order to reach the specified vertex, a boolean value called **visited**, that supplies information on whether or not it's been accessed during the implementation of the breadth first search (BFS), the **id**, which is simply the number of the vertex that is a unique identifier, the **parent,** which is the node that visits the specified node during BFS, and an **adjacency list** if the implementation was utilizing a list structure, otherwise there is a adjacency matrix that is a property of the graph itself.

The two implementations that will be assessed are the **adjacency matrix** implementation and the **adjacency list** implementation. The matrix implementation is a simple two-dimensional array that keep track of edges from each vertex using a simple boolean interpretation with either a 1 or 0 in each space to indicate whether an edge exists between nodes. The list implementation utilizes a linked list that is a part of every vertex object. This list points to all other vertex objects that it is connected to. The ordering of the list is arbitrary. It should be made clear that throughout the this paper all graphs are utilizing directed edges. Undirected edges will have to wait for a different day.

Memory

Before we move on to computational complexity we will first discuss the amount of **memory** that each graph uses and how it's impacted differently by a list and by a matrix. A matrix will always have to set aside enough memory for $O(V^2)$. This is perfectly fine if we are dealing with **dense** graphs. Dense graphs have vertices that have edges directed to every other vertex, thus all that we have set aside will be used

in order to indicate an edge from each vertex. However, this implementation can easily be seen as troublesome if we are dealing with **sparse** graphs. Sparse graphs are defined as not dense. For the purposes of this paper we will assume a sparse graph is a graph that utilizes the minimum amount of edges in order to have that graph be satisfied as connected. If we have a graph that only links one node to another, then we are utilizing very little of the matrix.

For example, *if we have 100 vertices that all link in a row, we would only have 100 edges, but a matrix will set aside memory for 10,000 edges!* This is obviously a poor way to represent the graph so instead an easier way would be to use a linked list representation for the graphs. This means that we will only use enough memory as we have too for each edge. This is a very appealing process when we're dealing with sparse graphs, but the overhead of creating more edges or attempting to find a specific edge is will take a larger amount of time when compared to its counterpart.

Two common ways of implementing lists are through a hash table that has a linked lists for all adjacent vertices with the ID of each vertex used for the hash key. The second option which is the option that was implemented in this report was to create a list datatype for each vertex that pointed to all of its adjacent vertices. Creating a graph can take a much longer time to implement as well when building each edge one at a time. It should also be noted, although this wasn't implemented into the source code, that it is theoretically *a lot easier to add a new vertex into the graph when utilizing an adjacency list*, however if a matrix was used, then in order to add a new node, the entire matrix would have to be copied into a brand new matrix. If

a rather large graph was in use this can take an excessive amount of time when compared to simply adding a new node into an adjacency list.

Speed

Now we will discuss the speed of building a graph and the speed of running BFS throughout three different graph sizes: sparse, dense, and “thick” (Thankfully software engineers aren’t sought after for their naming capabilities, for the most part). Thick is just a graph that utilizes $O(V^2/4)$ edges. This is completely arbitrary and is simply meant to simulate a graph that is not dense but not necessarily sparse or close to empty. “Sparse” in this scenario means a graph where each vertex has exactly one edge. All of the data gathered is represented in the tables below. All data is an accumulated average across multiple test cases.

Sparse Graph

Size (Vertices)	BFS Time (ms)	Build + BFS Time (ms)	List / Matrix
100	1	2	List
100	1	16	Matrix
1,000	5	13	List
1,000	12	66	Matrix
10,000	479	1,418	List
10,000	958	38,535	Matrix

Wesley R. Abbey
Zhishan Guo

Dense Graph

Size (Vertices)	BFS Time (ms)	Build + BFS Time (ms)	List / Matrix
100	3	15	List
100	4	22	Matrix
500	475	1,830	List
500	351	366	Matrix
1,000	4,250	17,500	List
1,000	2,850	2,906	Matrix

Thick Graph

Size (Vertices)	BFS Time (ms)	Build + BFS Time (ms)	List / Matrix
100	2	5	List
100	3	7	Matrix
500	97	370	List
500	85	105	Matrix
1,000	700	3,950	List
1,000	620	670	Matrix

Analysis

Looking through the data is is clear that for the most part when dealing with sparse data BFS performs much better and when dealing with very dense graphs it's a lot easier to use a matrix as a representation for the adjacency list. Build time is the same as well. It's much faster to create graphs with sparse data then it is to create graphs with dense data using lists and vice versa for matrices. Build time can be very cumbersome when we're dealing with lists and dense graphs however and the amount of time can be incredibly detrimental. While looking at the time it takes to implement breadth first search on thick graphs the difference is minimal. Matrix implementation is slightly faster, however it makes little difference especially if we're dealing with a small data set. Once again it should be noted that although it would take a very long time to create the initial graph, if we wanted to add a new node it should take a shorter amount of time if we were to use lists. Using a matrix would require a complete rebuild while an adjacency list would only require an enqueue to the list of vertices. The add node function was not implemented in the first version of the code base.

Conclusion

When we build graphs and want to attempt to determine the shortest path between specific vertices one of the most common algorithms to use is Breadth First Search (BFS). This can be utilized using two different data structures, a linked list or a matrix.

When we know that the graph is going to be sparse, it is almost universally a better option to use an adjacency list as the data structure representation instead of an adjacency matrix.

When we are dealing with a large number of vertices but not necessarily a complete graph it can be difficult to determine what representation to use. If we know that we don't have to constantly rebuild the graph over and over again it could be more beneficial to use linked lists. Otherwise if for some reason we were creating large graphs over and over again it would be better to use a matrix representation. When dealing with adding random new nodes it would be a lot better to use a linked list representation instead of a matrix since the cost of rebuilding an entire matrix would be a lot worse than simply adding an additional node.

Dense graphs execute BFS faster all of the time, although it isn't an exceptionally large difference in time when compared to the amount of time it takes to build the initial graph. If building a large graph is required an adjacency list will take way too long to be considered beneficial in any way. A matrix will probably be the way to represent the graph in that situation.

Work Cited

Weiss, M. *Data Structures and Algorithms Analysis in Java 3rd.* Pearson. 2012

Cormen, T. *Introduction to Algorithms 3rd.* MIT Press. 1990