The University of Texas at Austin

Algorithmic Foundations

Software Engineering - Option III

Determining Similar Subreddits from Graph Traversal Algorithms

Authors:
Howie Benefiel
Kelsey Pate II
Tyler King

Professor:
Dr. Sarfraz Khurshid

April 30, 2017

TEXAS
The University of Texas at Austin

Abstract

1 Introduction

2 Reddit Scraper

3 Dijkstra's Shortest Path Algorithm

The main idea of Dijkstra's Algorithm is to initialize all path distances to infinity, then starting from the source node, check all adjacent nodes for a shorter path. If a shorter path is found, save the path and push it into a priority queue where it will be sorted by minimum distance. After all adjacent nodes are checked, you add the current node to a list of traversed nodes and move to the next smallest node in the priority queue, which will be at the top of the heap. The process is repeated until the priority queue is empty and you are left with the shortest path to each node from the source node.

3.1 Implementation

Dijkstra's Shortest Path Algorithm was implemented in python as a function that accepts an adjacency list representation of a graph and the source node. The implementation is shown in Figure X. XXX: INSERT CODE SCREENSHOT The python library heapq was used to implement the priority queue structure. The function heappush() adds an element into the queue and heappop() removes and returns the top (smallest) element. Both functions run in

$$O(log(h_n))$$

time, where

$$h_n$$

is the number of elements in the queue.

3.2 Analysis

The priority queue implementation of Dijkstra's Algorithm runs in

$$O(Elog(V))$$

time where E is the number of edges and V is the number of vertices in a graph. In the worst case (a fully connected graph), each vertex is connected to V-1 other vertices, therefore the max size of our heap is O(V) and the heappush() and heappop() functions become O(log(V)). We analyze each connected edge of each vertex, which has a maximum value of E, giving a total function order of

$$O(Elog(V))$$

.

4 Bellman-Ford's Shortest Path Algorithm

Bellman-Ford's is more complex than Dijkstra's, but can handle negative edge-weights. The algorithm is built around the concept of "relaxing." An infinite amount of distance is assumed at first. Over time, the value is replaced with increasingly accurate values until the optimal path is achieved. This is done on all edges V - 1 times (V = vertices).

4.1 Implementation

Bellman-Ford's algorithm was implemented in python as a function that accepts an adjacency list representation of a graph and the source node. The implementation is shown in Figure X XXX: INSERT CODE SCREENSHOT The python library defaultdict is used to represent the destination and predecessor to keep track of distance. Function Initialize creates the destination and predecessor. Relax gradually determines shortest path and then negatative edge-weights are handled last.

4.2 Analysis

The initialization of dictionaries runs in

$$O(1)$$

. Relax runs in

$$O(1)$$

as well. Where the additional overhead comes in the algorithm itself taking

$$O(V * E)$$

where V is vertices and E is edges. This gives us a total run time of a constant C +

$$O(V * E)$$

.

5 Testing Implementation Correctness

In order to ensure algorithm correctness, a test suite was implemented to check the resulting shortest path results from each algorithm. The correctness check only checks the distance between the source vertex and the destination node and does not compare the path taken. This is because there is no 'correct' path when multiple exist of the same distance between source and destination. To illustrate this, imagine a four vertex undirected graph connected by edges (1,2),(2,4),(1,3),(3,4). You can travel from vertex 1 to vertex 4 in two steps either through vertex 2 or vertex 3. They are both valid shortest paths from vertex 1 to vertex 4, as would any path with a distance of two steps. The test suite contained in the test.py source file contains two test cases. Test Case 1 is a randomized test that compares the resulting shortest path result for each destination vertex of the Dijkstra and Bellman-Ford functions for the same graph and 50 randomly selected source vertices. This test case is intended to provide additional coverage each time the test is executed. A full-coverage test could have been implemented, but the execution time would not have been ideal for a demonstration. Test Case 2 runs each algorithm against a relatively small graph for a predetermined source node and checks the results against precomputed results. This test case ensures there is no logical flaw that exists in both algorithms that would cause Test Case 1 to report a false pass.

6 Performance Comparison

7 Conclusion