All-pairs shortest path problem

Mărunțiș Andrei, 323CA

University Politehnica of Bucharest, Faculty of Automatic Control

Abstract. This paper will very succintly analyze the all-pairs shortest path problem and mention 3 algorithms that can be used to solve it. It will then explain how tests will be created to test the algorithms.

Keywords: Graphs · Shortest path · All-to-all

1 Problem description

The all-pairs shortest path problem allows us to find the shortest path between any two nodes in a graph (that can be a set of cities with roads connecting them, or computers connected to the internet via cables). This optimization problem allows us to save resources such as time or fuel by finding the most efficient path between two points. Practical applications for this problem can be transportation of goods between cities, or internet communication between computers.

This problem is used when we need to find to answer multiple queries of the type *"what is the shortest path from node A to node B?"*. The environment where this problem is applied needs to be static, meaning the costs of the vertices need to remain constant over time or at least change infrequently, so that a large number of interrogations can be executed. If this condition is not met (for example, a GPS application cannot use this type of preprocessing, since traffic changes all the time), then it is more efficient to answer each individual query by running a shortest path algorithm.

2 Algorithms

2.1 Dijkstra's Algorithm

Dijkstra's Algorithm *fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.*[2]

In order to solve the all-pairs shortest path problem with this algorithm, some modifications need to be made: we need to run Dijkstra's algorithm multiple times, once for each node except for the last one. This is because when the first $N - 1$ computations are completed, we have found the shortest path from the first $N - 1$ nodes to the N-th node and no more processing is necessary.

Furthermore, with each iteration of the algorithm, the next iteration becomes faster, as it needs to analize fewer nodes (the k-th iteration needs to analize $N - k + 1$ nodes, as the first $k - 1$ nodes have already been processed).

2.2 Bellman-Ford Algorithm

The Bellman-Ford Algorithm is similar to Dijkstra's Algorithm, with some differences: it is slower, but *more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.*[3] If there is a negative cycle in the graph being analysed, the Bellman-Ford Algorithm can detect and report it. Similarly to Dijkstra's Algorithm, we need to apply the Bellman-Ford Algorithm multiple times to solve the all-pairs shortest path problem: once for each node, except for the last one.

One of the restrictions of this algorithm is that it can only be applied efficiently on directed graphs with or without negative edges. In an undirected graph, Dijkstra's algorithm is asymptotically faster than Bellman-Ford and it can only run on graphs with positive edges, since any negative edge would define a negative cycle. Therefore, Dijkstra's Algorithm is always faster on undirected graphs.

2.3 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is an exhaustive algorithm that tests every possible combination of edges in a graph to calculate the shortest path between any 2 nodes. Unlike the other two algorithms mentioned, a single execution will find the lowest cost of the paths between any 2 nodes.

Similarly to the Bellman-Ford Algorithm, the Floyd-Warshall Algorithm can work on directed graphs with or without negative edges, but without negative cycles. It is also capable of detecting negative cycles.

3 Evaluating the solutions

The parameter that I am interested in the most when evaluating each algorithm is running time, as it directly represents the complexity and efficiency of each algorithm. Another metric that I am interested in is the correctness of the solutions, as some algorithms might be unable to run on some graphs (for example, Dijkstra's Algorithm on a graph with negative costs).

To evaluate the solutions I will use multiple tests with various types of inputs representing various types of graphs. The tests will cover a large variety of graphs: directed and undirected, dense and sparse, with positive and negative costs (combinations of these). Each type of graph will have multiple tests, with increasingly large numbers of nodes and edges to thoroughly test their efficiency. I am expecting some algorithms to give wrong answers to some graphs, as per the example given above.

The numerical values in these tests will be randomly generated with various upper bounds. The upper bound will increase with the number of nodes to have as many edges with different values and I will also test the inputs to have as few acyclic graphs or graphs with a small number of cycles, since an acyclic graph has at most exactly one path from any node to any other node. In addition, I will be creating the tests to have graphs of each type, as mentioned in the paragh above.

References

1. Wikipedia: Dense graphs (Last access: 24 nov 2022)
2. Wikipedia: Djikstra's Algorithm (Last access: 24 nov 2022)
3. Wikipedia: Bellman-Ford Algorithm (Last access: 24 nov 2022)
4. Wikipedia: Floys-Warshall Algorithm (Last access: 24 nov 2022)