Unity ID's : amadaan abhutan haggarw sbhatt

Dijkstra vs Floyd Warshall Algorithm

INTRODUCTION

Dijkstra's Algorithm : This is a Single Source Shortest Path algorithm , run on sparse or dense connected graphs. It is based on greedy approach, hence selecting the best path available at an instant of time.

Floyd Warshall Algorithm : This is an All Source Shortest Path algorithm, which is based on Dynamic Programming, where we calculate and store distance between vertices and fetch it when the problem is further broken into smaller instances.

Purpose of the Experiment - The main goal of the project was to compare the efficiency of Floyd Warshall and Dijkstra's Algorithm, when these algorithms are run for all "V" vertices on a given set of Dense and Sparse Graphs.

Hypothesis - Our hypothesis moving forward in the implementation project was " On running the Dijkstra and Floyd Warshall for 'V' vertices in dense and sparse graphs, Floyd Warshall Algorithm runs faster than Dijkstra's algorithm for very dense graphs considering we are using Priority Queues for implementing Dijkstra's Algorithm.

Data Input -

Input data is divided in 2 categories : Sparse Graph and Dense Graph
Sparse Graph: A sparse graph is a graph G (V,E) where $|E| = O(|V|)$
Dense Graph: A dense graph is a graph G (V,E) where $|E| = O(|V|^2)$

To get a bound on the number of vertices and edges we used the functions ***cn*** and ***nlgn*** where n is the number of vertices and growth rate of edges is a function of number of vertices.

Data Structure : Priority Queue for Dijkstra's Algorithm and 2D Matrix for Floyd Warshall Algorithm.

Graph Generation:

./nextgen *type vertices edges seed output-file*

type is random (edges selected completely at random)
vertices and *edges* are the number of vertices and edges, respectively
seed is a random number seed (any integer)
output-file is the name of the file in which graph is stored

Set Of Inputs: We took a sample of 5 sparse graphs and 5 dense graphs, for the purpose of comparison on running time between the two algorithms and to find a conclusion to our hypothesis. The edges ranged from 3K- 45K for sparse graphs, 50K- 250K for dense graphs and 1500K - 3500K for very dense graphs. Corresponding to edges, vertices vary from 1k - 5k.

Unity ID's : amadaan abhutan haggarw sbhatt

Programming Environment
Java (Version "1.7.0_40") - the base language for developing both Algorithms.
Java SE Runtime Environment (1.7.0_40-b43)
Eclipse (Version: Kepler Service Release 1 Build ID: 20130614-0229) - IDE to develop, run and execute algorithms.

Machine architecture
Windows 8.1 Pro OS 64-bit machine
Processor: Intel Core i5-2410M
Processor Speed: 2.3 GHz
Memory (RAM) - 4GB
Cache size - 2.4MB

Runtime Measurement : Java currentTimeMillis() - to collect readings of run time in milliseconds.

Summary: The results of running Dijkstra's on Sparse Graphs and Dense Graphs are tabulated below:

Graph density	**No of vertices/Edges**	**Average Run Time for Dijkstra**	**Average Run Time for Floyd Warshall**
Sparse Graph 1	1000/31623	547.8	1380.2
Sparse Graph 2	2000/8000	1084.4	9640
Sparse Graph 3	3000/18000	2717	33298.2
Sparse Graph 4	4000/47864	6733	82021.2
Sparse Graph 5	5000/40000	8865.2	165949.2
Dense Graph 1	1000/400000	8227	1485
Dense Graph 2	2000/1500000	62969	10129.4
Dense Graph 3	3000/3500000	157217.5	38545.8
Dense Graph 4	4000/5500000	301769	89200.8
Dense Graph 5	5000/8000000	589362	153398.2

Unity ID's : amadaan abhutan haggarw sbhatt

Explanation and Observation:
For all pair shortest path, run times for both the algorithms would be :

Dijkstra's ≈ O(E + Vlg V)

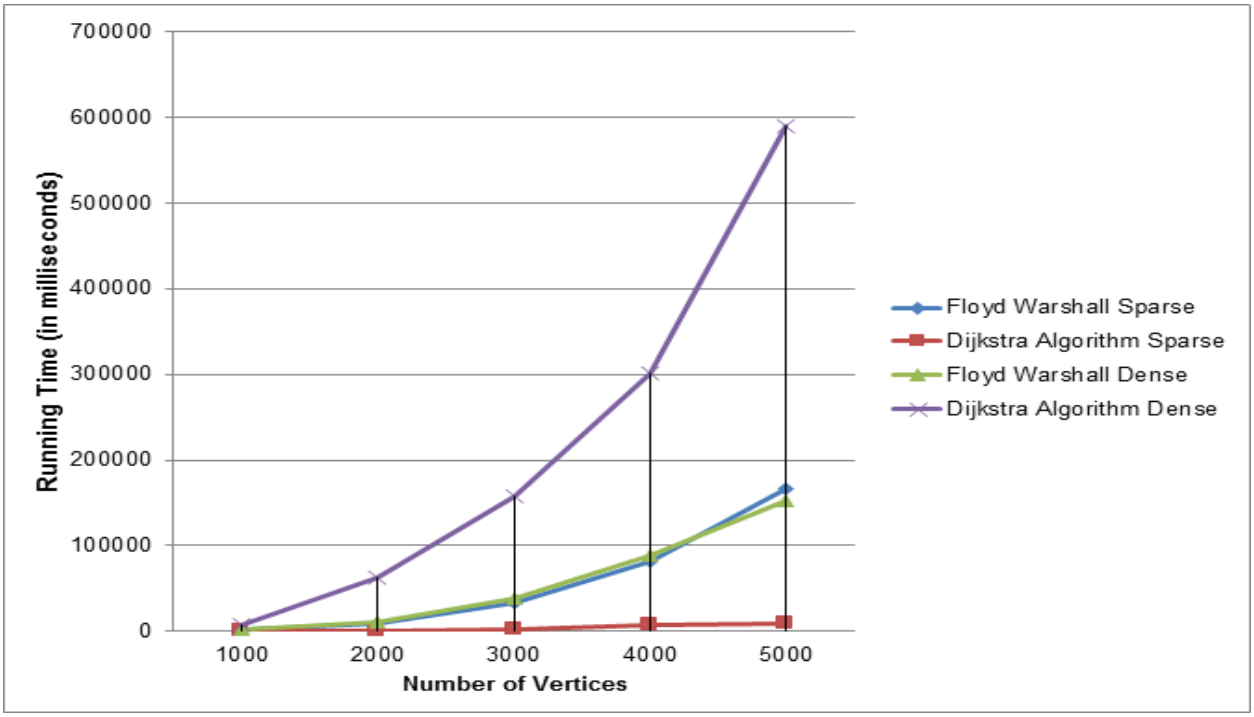
Floyd Warshall's ≈ $O(V^3)$

The data collected through experiments highly correlates with the hypothesis.
Dijkstra implemented with Priority queue and Floyd Warshall implemented with Adjacency Matrix both are theoretically equivalent but behaves slightly different in terms of run time on sparse and dense graphs.

Here Dijkstra implementation is backed by a Priority Queue which is implemented as binary heap as default in Java. Implementation of Dijkstra through binary heap increases it's runtime in case of sparse graphs but it is not always asymptotically faster than Floyd Warshall.

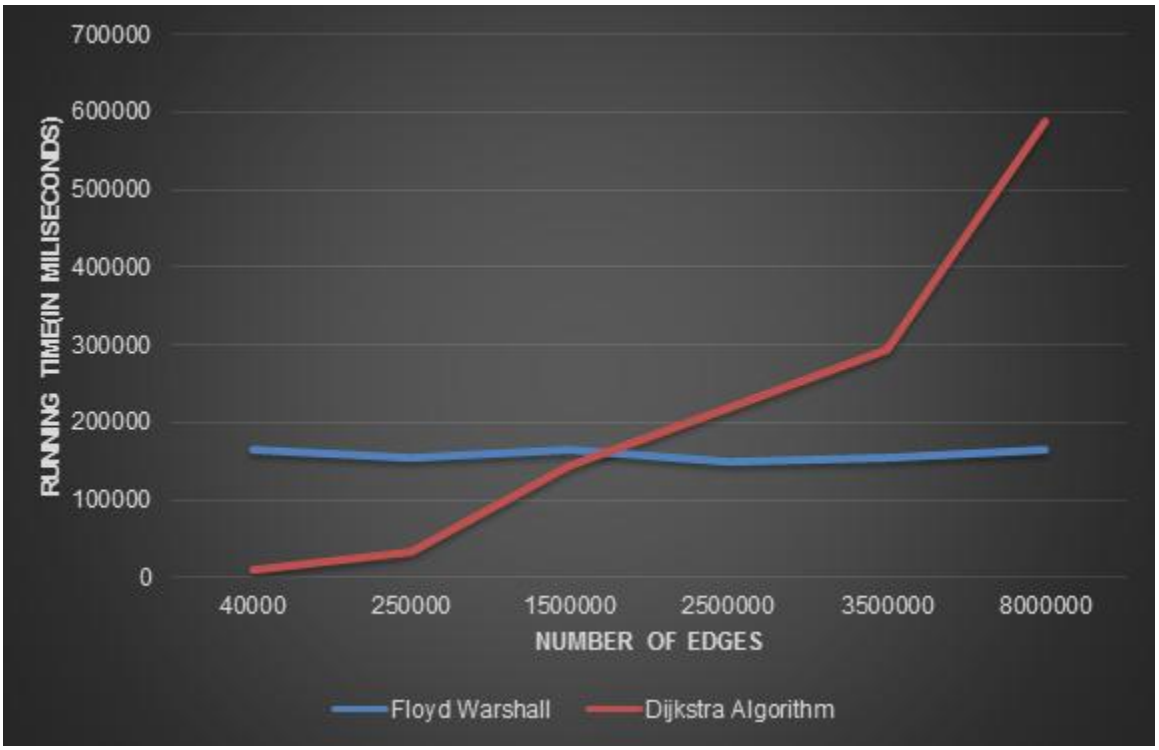
For a sparse graph, when number of edges are of order O(n), Dijkstra run time complexity is of order O(n* mlgn) ≈ $O(n^2 \lg n)$ which is less than run time complexity of Floyd Warshall $O(n^3)$. Inverse to this, for a very dense graph when number of edges are of order $O(n^2)$, Floyd Warshall proves to be asymptotically faster. Dijkstra run time complexity in such case turns out to be $O(n^2 * m\lg n) \approx O(n^3 \lg n) > O(n^3)$.

The chart in below figure shows the total time taken for both algorithms. Vertices range from 1k-5k. The graph below shows the variance of Dijkstra and Floyd Warshall as the density of graph changes.



Unity ID's : amadaan abhutan haggarw sbhatt

Here the graph represents the performance of both algorithms for a particular set of vertices. As the number of edges cross a certain point, Floyd algorithm run time reduces to a great extent in comparison to Dijkstra's.



Conclusion:

Both Floyd Warshall and Dijkstra algorithm may be used for finding shortest path between vertices. The biggest difference is Floyd Warshall algorithm finds the shortest path between all vertices and Dijkstra algorithm finds the shortest path from one vertex to all other vertices. In case when Dijkstra is run for all vertices, its run time increases for dense graphs. So for small and sparse graphs, Dijkstra can be the best choice, whereas for dense graph, Floyd Warshall algorithm seems to be a better bet.

Future Work:

Here we have used Binary Heap for Dijkstra implementation and it shows variance in run time for sparse and dense graphs. We can do a similar experiment by using Fibonacci heap or a different data structure to find out how it behaves in comparison to other algorithms including Floyd Warshall.