All-pairs shortest path problem

Mărunțiș Andrei, 323CA

University POLITEHNICA of Bucharest, Faculty of Automatic Control

Abstract. This paper will analyze various pathfinding algorithms that solve the all-pairs shortest path problem. The three algorithms will be compared based on their complexity and running time, and various optimisations will be proposed for each algorithm.

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

1 The problem

1.1 Description

Given a graph G with N nodes and E edges, where each edge has a cost C associated, find the shortest path between any two nodes.

1.2 Practical applications

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 the answer to 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 Graph features

The types of graphs where we need to solve the problem can influence the algorithms that we apply.

Undirected vs directed In an undirected graph, an edge connects two nodes both ways, while in a directed graph an edge only allows one-way travel. As such, an undirected graph can be a considered a directed graph with twice as many edges.

Cyclic vs acyclic A cycle in a graph is a set of edges that allows travel from one node back to itself. As such, an acyclic graph can have at most one possible path between two nodes, while a cyclic graph can have multiple paths that need to be checked and compared.

Dense vs sparse To define the notion of dense/sparse graph we need to define *graph density*[1]:

$$D = \frac{E}{N(N-1)} \tag{1}$$

However, the notion of sparse/dense graphs is not clearly defined. A sparse graph is a graph with density D almost equal to 0 (has very few edges compared to the number of nodes), and a dense graph is a graph with density D almost equal to 1 (has a large number of edges compared to the number of nodes).

Generally, it is more efficient to represent a sparse graph in memory as an **adjacency list**, while a dense graph is better represented as a **matrix of adjacency**.

Positive vs negative costs A graph with positive costs has all edges with positive costs, while a graph with negative costs has an least one edge with negative cost. However, a graph must not have a negative cycle (i.e. a cycle whose edge costs sum to a negative value) in order to be solvable (because if there is a negative cycle, then we can find paths between two nodes with infinitely negative cost, obtained by traveling multiple times around the negative cycle).

3 Algorithms

All algorithms will be presented with a brief description of their functionality, pseudocode and description of their complexities.

3.1 Dijkstra's algorithm

Description 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] The algorithm assigns each node a visited/unvisited state, where visited means that it has found the shortest path from the source to that node and unvisited means that the shortest path might still be optimized. Initially, only the source node is visited. With each step, the algorithm chooses the closest, unvisited node n of the source (that is guaranteed to already have

the correct shortest distance) and visits it. Then every other unvisited node is selected as destination and the algorithm asks the question *Is there a shorter path from source to destination that passes through n?* If the answer is positive, then the shortest path between the source and destination is updated. When all nodes are visited, the algorithm is complete and we have the shortest path tree.

Pseudocode The pseudocode for the algorithm is shown below(Source: [2]):

Algorithm 1 Dijkstra's algorithm

```
1: for all node n in Nodes do
2:     dist[n] ← INF
3: end for
4: dist[source] ← 0
5: while Q is not empty do
6:     u ←node in Q with min dist[u]
7:     remove u from Q
8:     for all neighbour v of u still in Q do
9:         altDist ← dist[u]+cost(u, v)
10:        if altDist < dist[v] then
11:            dist[v] ← altDist
12:        end if
13:    end for
14: end while
```

The meanings of the variables are:

- *dist* is the array of shortest path distances from source to every other node known so far, and its initial value should be INF for all nodes except the source
- Nodes is a data structure that describes the nodes of the graph
- Q is a set that contains all the unvisited nodes (therefore at the start of the execution it contains all nodes)

Complexity The complexity of this algorithm is $O(N^2)$ when implemented using a simple array for the set Q, but its complexity can be improved to $O((N+E)logN)$ when using a min-priority queue and to $O(E+NlogN)$ when using a Fibonacci heap (this version of Dijkstra's algorithm is the fastest known algorithm to find the shortest-path tree in a graph). For large graphs, we can consider the following relation between the number of nodes and edges: $E \approx N^2 \cdot D$, where D is the density of the graph as defined in section 2 and is a constant between 0 and 1. Therefore, for large graphs that are not sparse, the complexity of Dijkstra's algorithm can be considered as $O(N^2)$ regardless of implementation.

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, setting each node as the source. This way, the complexity of the all-to-all shortest paths algorithm becomes $O(N^3)$ on directed graphs when using Dijkstra's algorithm.

Other optimizations In an undirected graph, we know for certain that the shortest path from 1 to 2 is also the shortest path from 2 to 1. Therefore, we can make the following optimization: after running the agorithm with node 1 as the source we know the shortest path from 2 to 1, so we can consider node 1 as visited when running the algorithm wih 2 as source. Similarly, when node 3 is set as source, nodes 1 and 2 can be considered visited etc. Therefore, when node 1 is set as source, N nodes need to be visited which requires roughly N^2 operations, when node 2 is the source we need to visit $N-1$ nodes which requires $(N-1)^2$ operations etc. The total number of operations that we need to execute is:

$$N^2 + (N-1)^2 + ... + 1^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3} \tag{2}$$

While this optimization does not reduce complexity, it does reduce the running time considerably when processing undirected graphs.

One last thing worth mentioning about this algorithm is that the multiple runs of Dijkstra's algorithm are **independent** (whichever node is selected as source, it does not require data from when other nodes are selected as source). Therefore, unless the optimization presented above for undirected graphs is used, the algorithm can run on multiple cores, shortening its running time.

3.2 Bellman-Ford algorithm

Description The Bellman-Ford algorithm is similar to Dijkstra's algorithm; 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 that is analysed, the Bellman-Ford algorithm can detect and report it. The algorithm functions very similarly do Dijkstra's algorithm: it sets a node as source and for every edge asks the question *Is there a shorter path from the source to any other node that passes through this edge?* If the answer is positive, then the shortest path is updated. This process needs to be repeated $N-1$ times. As we can see, the Bellman-Ford algorithm functions very similarly to Dijkstra's algorithm, but it improves the distances by checking if the path contains an edge instead of a node.

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 an undirected graph with a negative edge will always define a negative cycle, so the Bellman-Ford algorithm presents no advantage when processing undirected graphs.

Algorithm 2 Bellman-Ford algorithm

```
1: for all node n in Nodes do
2:     dist[n] ← INF
3: end for
4: dist[source] ← 0
5: for N − 1 times do
6:     for all edge e in Edges do
7:         (n1, n2, cost) ← e
8:         if dist[n1] + cost < dist[n2] then
9:             dist[n2] = dist[n1] + cost
10:        end if
11:    end for
12: end for
```

Pseudocode The pseudocode for the algorithm is shown above(Source: [3]):
The meanings of the variables are:

- *dist* is the array of shortest path distances from source to every other node known so far, and its initial value should be INF for all nodes except the source
- Nodes, Edges are two data structures that describe the graph
- each edge contains information n1 - starting node, n2 - destination node, and cost

Complexity The Bellman-Ford algorithm has complexity $O(N \cdot E)$. Making the same assumption used for analizing Dijkstra's algorithm, that for large graphs, the relation between the number of nodes and edges is $E \approx N^2 \cdot D$ with $D \in [0, 1]$, we can say that the Bellman-Ford algorithm has a complexity of $O(N^3)$ for large, dense graphs. This makes it extremely inefficient at finding the shortest path tree, but it can run on graphs with negative costs.

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. We select each node as source and run the algorithm that has complexity $O(N^3)$, making the total complexity for solving the all-pairs shortest path problem $O(N^4)$. This is very inefficient and can only run graphs with up to a few hundred nodes in acceptable time (a few seconds).

Other optimizations Similarly to Djikstra's algorithm, the Bellman-Ford algorithm can be parallelised for increased performance.

Another significant improvement that can be brought to this algorithm is checking only the edges that can improve the shortest path. After a run through all the edges of the graph, the shortest path to some nodes may not have improved, in which case the edges adjacent to those nodes will not improve the shortest paths to any other nodes. Therefore, we can ignore them for the next runs through the edges of the graph. This mechanism can be implemented with

a heap or a queue, in which case the theoretical complexity of the algoritm becomes $O(N \cdot ElogN)$ and $O(N \cdot E)$, respectively[5]. However, in practice, despite the complexities being the same/worse, this version of the algorithm is linearly faster than the normal version.

3.3 Floyd-Warshall algorithm

Description The Floyd-Warshall algorithm is the first algorithm analysed in this paper that is specialised for solving the all-pairs shortest path problem. It 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.

The Floyd-Warshall algorithm tests for all triplets of nodes (source, intermediate, destination) whether a path from source to destination that passes thorugh intermediate is faster than the currently known best path from source to destination. It first sets the intermediate node and then checks if the condition described above is true for all pairs of source and destination nodes, then proceeds to the next intermediate node and repeats the steps. This is a very short, simple and elegant algorithm that can be easily understood by reading its pseudocode or implementation.

Pseudocode The pseudocode for the algorithm is shown below(Source: [4]):

Algorithm 3 Floyd-Warshall algorithm

```
1: for k ← 1, N do
2:     for i ← 1, N do
3:         for j ← 1, N do
4:             if dist[i][k] + dist[k][j] < dist[i][j] then
5:                 dist[i][j] ← dist[i][k] + dist[k][j]
6:             end if
7:         end for
8:     end for
9: end for
```

In the pseudocode above, *dist* is initially the matrix of adjacency of the graph and after the algorithm finishes, it will contain the shortest distances between any 2 nodes.

Complexity As can be easily seen above, the complexity of the algorithm is $O(N^3)$, and not many significant optimizations can be brought.

4 Comparison of the algorithms

4.1 Theoretical complexities

First, let's sum up the complexities of the algorithms in the preceding section (N = number of nodes, E = number of edges):

- Dijkstra's algorithm: $O(N^3)$ for a simple implementation, up to $O(N(E + NlogN)) = O(N \cdot E + N^2 logN)$ for implementation with Fibonacci heap, $\approx O(N^3)$ for large, dense graphs even when using Fibonacci heaps
- Bellman-Ford algorithm: $O(N^2 \cdot E) \approx O(N^4)$ for large, dense graphs
- Floyd-Warshall algorithm: $O(N^3)$

It is easy to rule out the Bellman-Ford algorithm as the least efficient algorithm of the three. When compared to Dijkstra's algorithm, it is always slower but has the advatage of being able to process graphs with negative costs, while it has no advantage over Floyd-Warshall: it is always slower and both can process graphs with negative weights. Therefore, Bellman-Ford is overall a bad algorithm for solving the all-to-all shortest path problem. However, it is used in Johnson's algorithm which has a complexity of $O(N \cdot E + N^2 logN)$ (same as Dijkstra) and is able to process graphs with negative weights. Johnson's algorithm is one of the most versatile and fastest algorithms for solving the all-to-all shortest path problem, but it is not discussed here in detail.

However, when comparing Dijkstra's and Floyd-Warshall algorithms, there are situations where either is more suitable. Since the Bellman-Ford algorithm has a complexity that does not depend on E (the number of edges), as it always checks all possible paths, it is better suited for dense graphs (it will have a similar running time for two graphs with the same number of nodes N but with with different densities $D = 0.2$ and $D = 0.9$). Similarly, for sparse graphs, Djikstra's algorithm implemented with a Fibonacci heap is better than Floyd-Warshall, since its complexity depends on E the number of edges. However, we need to remember that for graphs that can have negative edge costs, we cannot use Dijkstra's algorithm and instead need one of the other algorithms.

4.2 Practical testing

For testing I have implemented a version for each algorithm:

- Simple Dijkstra's algorithm using an array for the set Q
- Bellman-Ford algorithm
- Floyd-Warshall algorithm

The programming language that I used is Java, which is an interpreted lanaguage and therefore, solutions written in lower-level languages like C/C++ might be faster than this implementation. For testing, I have prepared a set of 30 tests containing graphs grouped into 6 families:

- Directed graphs

  - sparse: $D = 0.2$
  - average density: $D = 0.5$
  - dense: $D = 0.8$

– Undirected graphs

  - sparse: $D = 0.2$
  - average density: $D = 0.5$
  - dense: $D = 0.8$

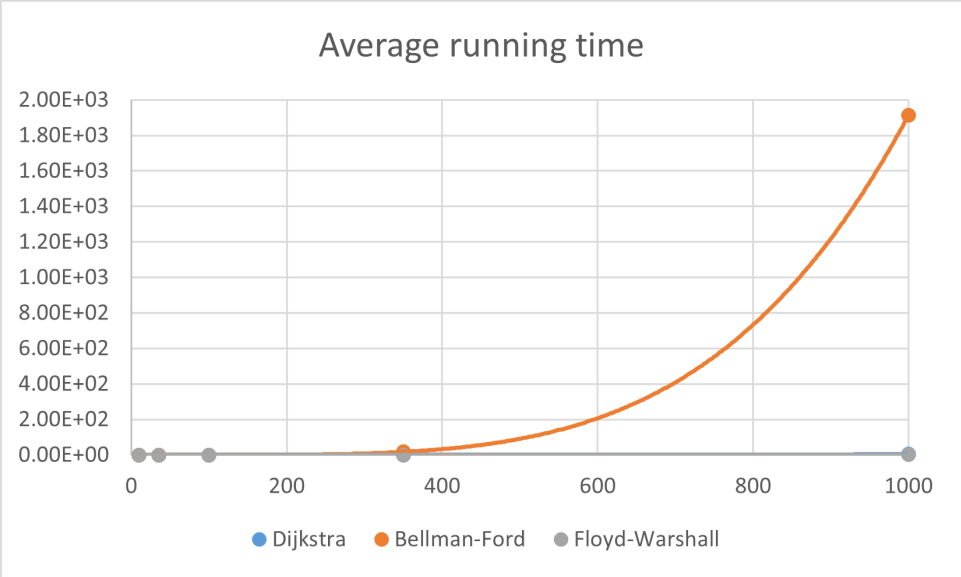
Each graph family contains a randomly-generated graph with 10, 35, 100, 350 and 1000 nodes. This set of 30 tests can test the algorithms against all kinds of graphs, detecting which algorithms are best in which situations.

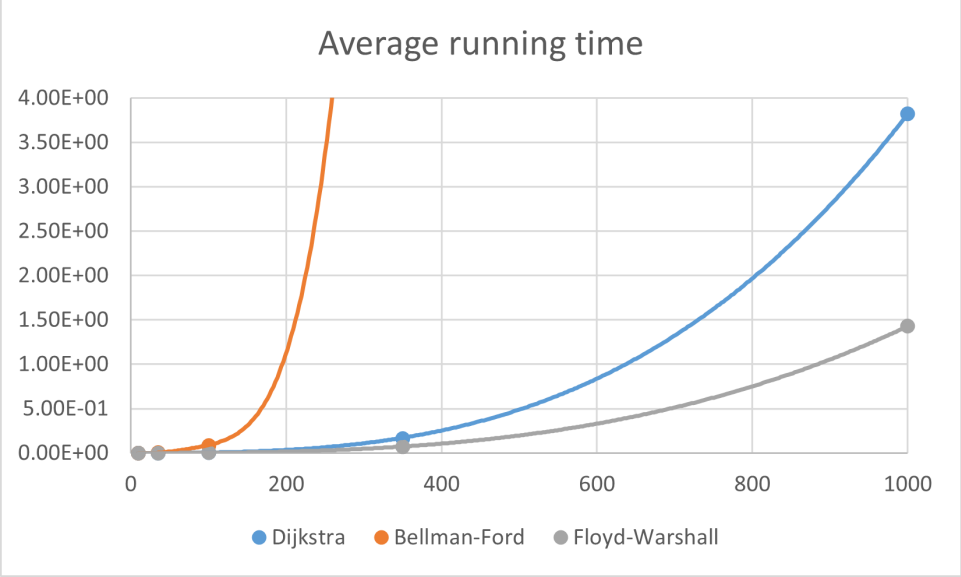
For each dataset I have run the algorithms 5 times and calculated the average time, which appears in the table below. It is worth noting, however, that the Bellman-Ford algorithm has a running time of 10-60 minutes on tests with 1000 nodes, so I only have one set of data for these cases. Below I have selected a smaller set of only 10 data entries significant for my analysis, for the entire table check the Annexes section at the end of the document.

Table 1: Average running times for algorithms (seconds)

Test Nr	**N**	**Type**	**D**	**E**	**Dijkstra**	**Bellman-Ford**	**Floyd-Warshall**
1	10	Directed	0.2	18	5.93E-05	2.57E-04	4.38E-05
2	35	Directed	0.2	238	2.33E-03	7.41E-03	3.21E-03
3	100	Directed	0.2	1980	6.63E-03	6.62E-02	7.14E-03
4	350	Directed	0.2	24430	1.68E-01	6.42E+00	1.19E-01
5	1000	Directed	0.2	199800	3.44E+00	6.24E+02	1.28E+00
6	10	Directed	0.5	45	2.89E-05	1.47E-05	7.25E-05
7	35	Directed	0.5	595	3.22E-04	1.09E-03	2.21E-04
8	100	Directed	0.5	4950	6.12E-03	9.07E-02	2.87E-03
9	350	Directed	0.5	61075	1.92E-01	1.59E+01	9.24E-02
10	1000	Directed	0.5	499500	4.33E+00	1.65E+03	2.07E+00

Before making some conlusions, I will also give a visualisation of the execution time of the algorithms:





As can be seen from the graphics, the Bellman-Ford algorithm is by far the slowest algorithm, confirming that its complexity is higher than that of the other two algorithms by a factor of N. The other two algorithms, Dijkstra and Floyd-Warshall, are much closer in terms of performance. By analyzing the data in the table above, we can see that Dijkstra's algorithm runs faster for a small number of edges and a small density (tests 2,3 and 6), while Floyd Warshall runs faster on larger graphs. This too confirms our assumption, that Dijkstra will be more efficient for small, sparse graphs and Floyd-Warshall is better for large, dense graphs.

5 Conclusions

There are many pathfinding algorithms that can be used to solve the all-to-all shortest path problem and I have analyzed here only 3 of them: Dijkstra's algorithm, Bellman-Ford algorithm and Floyd-Warshall algorithm. After a thorough analysis of these algorithms, I have found that, on average, the Floyd-Warshall algorithm is the fastest and most versatile, being able to process graphs with negative edge costs. Dijkstra's algorithm is faster only on small, sparse graphs. The Bellman-Ford algorithm alone is unsuitable for this problem, but is a good starting point and can be combined with Dijkstra's algorithm to create the Johnson algorithm, which combines the best of both worlds: fast execution of Dijkstra's algorithm and the ability to process graphs with negative edge costs, inherent to the Bellman-Ford algorithm.

References

1. Wikipedia: Dense graphs (Last access: 24 nov 2022)
2. Wikipedia: Djikstra's algorithm (Last access: 22 edc 2022)
3. Wikipedia: Bellman-Ford algorithm (Last access: 22 dec 2022)
4. Wikipedia: Floyd-Warshall algorithm (Last access: 24 nov 2022)
5. Infoarena: Bellman-Ford (Last access: 22 dec 2022)

Annex

Table 2: Average running times for algorithms (seconds)

Test Nr	**N**	**Type**	**D**	**E**	**Dijkstra**	**Bellman-Ford**	**Floyd-Warshall**
1	10	Directed	0.2	18	5.93E-05	2.57E-04	4.38E-05
2	35	Directed	0.2	238	2.33E-03	7.41E-03	3.21E-03
3	100	Directed	0.2	1980	6.63E-03	6.62E-02	7.14E-03
4	350	Directed	0.2	24430	1.68E-01	6.42E+00	1.19E-01
5	1000	Directed	0.2	199800	3.44E+00	6.24E+02	1.28E+00
6	10	Directed	0.5	45	2.89E-05	1.47E-05	7.25E-05
7	35	Directed	0.5	595	3.22E-04	1.09E-03	2.21E-04
8	100	Directed	0.5	4950	6.12E-03	9.07E-02	2.87E-03
9	350	Directed	0.5	61075	1.92E-01	1.59E+01	9.24E-02
10	1000	Directed	0.5	499500	4.33E+00	1.65E+03	2.07E+00
11	10	Directed	0.8	72	1.61E-05	2.06E-05	1.17E-05
12	35	Directed	0.8	952	2.77E-04	1.89E-03	2.31E-04
13	100	Directed	0.8	7920	5.53E-03	1.23E-01	1.83E-03
14	350	Directed	0.8	97720	1.70E-01	2.75E+01	5.82E-02
15	1000	Directed	0.8	799200	3.65E+00	3.32E+03	1.24E+00
16	10	Undirected	0.2	18	4.86E-05	1.09E-05	1.17E-05
17	35	Undirected	0.2	238	5.54E-04	4.87E-04	1.78E-04
18	100	Undirected	0.2	1980	5.03E-03	3.03E-02	1.56E-03
19	350	Undirected	0.2	24430	1.65E-01	7.40E+00	5.62E-02
20	1000	Undirected	0.2	199800	3.35E+00	7.15E+02	1.28E+00
21	10	Undirected	0.5	45	1.67E-05	1.58E-05	8.24E-06
22	35	Undirected	0.5	595	3.11E-04	1.20E-03	9.89E-05
23	100	Undirected	0.5	4950	5.30E-03	7.67E-02	1.98E-03
24	350	Undirected	0.5	61075	1.79E-01	1.81E+01	5.90E-02
25	1000	Undirected	0.5	499500	4.26E+00	2.12E+03	1.30E+00
26	10	Undirected	0.8	72	1.84E-05	2.11E-05	1.05E-05
27	35	Undirected	0.8	952	2.81E-04	1.94E-03	1.02E-04
28	100	Undirected	0.8	7920	5.64E-03	1.52E-01	1.63E-03
29	350	Undirected	0.8	97720	1.62E-01	2.85E+01	5.71E-02
30	1000	Undirected	0.8	799200	3.91E+00	3.05E+03	1.39E+00