**Floyd-Warshall Algorithm**

Floyd-Warshall is the simplest algorithm:

**Intuition**: We calculate the shortest possible path from node *i* to *j*using nodes only from the set *{1, 2, …, k}****as intermediate points***between them. We sweep *k* from *1* to *n,*and update *d[i][j],* the distance between nodes *i* and *j*:

*d[i][j] = min( d[i][j], d[i][k] + d[k][j] )*

Below is the implementation in C++:

|  |  |
| --- | --- |
|  | vector<vector<long long>> floydWarshal(vector<vector<int>> &graph) { |
|  | vector<vector<long long>> d(graph.size(), |
|  | vector<long long>(graph.size())); |
|  |  |
|  | //Initialize d |
|  | for (int i = 0; i < graph.size(); ++i) { |
|  | for (int j = 0; j < graph.size(); ++j) { |
|  | d[i][j] = graph[i][j]; |
|  | } |
|  | } |
|  |  |
|  | for (int i = 0; i < graph.size(); ++i) |
|  | for (int j = 0; j < graph.size(); ++j) |
|  | for (int k = 0; k < graph.size(); k++) |
|  | d[i][j] = std::min(d[i][j], d[i][k] + d[k][j]); |
|  |  |
|  | return d; |
|  | } |

### ****Dijkstra Algorithm****

Dijkstra algorithm finds the shortest path between a single source and all other nodes.

**Intuition:** Keep a list of visited nodes. At each step:

1. Find the unvisited node u with shortest distance
2. Relax the distance of neighbors of u
3. Add u to the visited list and repeat

Below is Dijkstra’s implementation in C++:

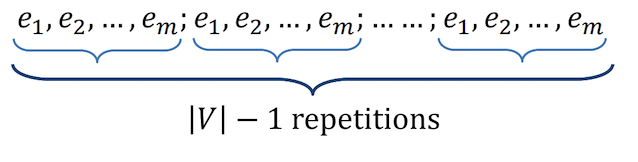
|  |  |
| --- | --- |
|  | vector<long> dijkstra(vector<vector<int>> &graph, int source) { |
|  |  |
|  | vector<long> d(graph.size()); |
|  | vector<int> s; |
|  | for (int i = 0; i < graph.size(); i++) { |
|  | d[i] = graph[source][i]; |
|  | } |
|  |  |
|  | s.push\_back(source); |
|  |  |
|  | while (s.size() < graph.size()) { |
|  | //Find minimum |
|  | int u = findMinInDButNotInS(d, s); |
|  | s.push\_back(u); |
|  |  |
|  | //Relax |
|  | for (int j = 0; j < graph.size(); j++) { |
|  | d[j] = std::min(d[j], d[u] + graph[u][j]); |
|  | } |
|  | } |
|  | return d; |
|  | } |

**Bellman-Ford Algorithm**

This algorithm finds shortest distance from source to all other nodes.

**Intuition:**We have two loops:

* The inner loop: We iterate on all edges. At each iteration we relax the distances by using edges from *0* to *j*.
* The outer loop: We repeat the inner loop *n-1* times



Bellman-Ford Algorithm. Picture taken from [here](https://courses.csail.mit.edu/6.006/spring11/lectures/lec15.pdf).

After the *i-th* iteration of the outer loop, the shortest paths with at most *i*edges are calculated. There can be maximum *n - 1* edges in any simple path, so we repeat *n - 1*times.

Below is an implementation of Bellman-Ford algorithm in C++.

|  |  |
| --- | --- |
|  | vector<long> bellmanFord(vector<vector<int>> &graph, int source) { |
|  | vector<long> d(graph.size(), INT\_MAX); |
|  | d[source] = 0; |
|  |  |
|  | for (int i = 0; i < graph.size() - 1; i++) { |
|  | for (int u = 0; u < graph.size(); u++) |
|  | for (int v = 0; v < graph.size(); v++) |
|  | d[v] = std::min(d[v], d[u] + graph[u][v]); |
|  | } |
|  | return d; |
|  | } |