Shortest Path Algorithms

The algorithm for the single-source shortest-paths problem (Dijkstra & Bellman-Ford) can solve many other problems, including the following variants.

Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

Shortest Path in Un-weighted Graph

Using BFS

Technically, Breadth-first search (BFS) by itself does not let you find the shortest path, simply because BFS is not looking for a shortest path: BFS describes a strategy for searching a graph, but it does not say that you must search for anything in particular.

Dijkstra's algorithm adapts BFS to let you find single-source shortest paths.

In order to retrieve the shortest path from the origin to a node, you need to maintain two items for each node in the graph: its current shortest distance, and the preceding node in the shortest path. Initially all distances are set to infinity, and all predecessors are set to empty. In your example, you set A's distance to zero, and then proceed with the BFS. On each step you check if you can improve the distance of a descendant, i.e. the distance from the origin to the predecessor plus the length of the edge that you are exploring is less than the current best distance for the node in question. If you can improve the distance, set the new shortest path, and remember the predecessor through which that path has been acquired. When the BFS queue is empty, pick a node (in your example, it's E) and traverse its predecessors back to the origin. This would give you the shortest path. reference

Dijkstra's Algorithm

Can be used for un-weighted graphs since in un-weighted graph all edges can be considered of weight 1.

Shortest path in Weighted Graph

Among the popular algorithms Dijkstra's algorithm, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices is a dynamic-programming algorithm. Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

The algorithms here use the technique of *relaxation*. For each vertex $v \in V$, we maintain an attribute v.d, which is an upper bound on the weight of a shortest path from source s to v. We call v.d a shortest-path estimate. We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$ time procedure

INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = NIL$
  s.d = 0

The following code performs a relaxation step on edge (u, v) in *O*(1) time:
RELAX(u, v, w)
  if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Dijkstra's Algorithm for weighted graph

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following algorithm, a min-priority queue Q of vertices used, keyed by their d values.

DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
    $S = \phi$
    $Q = G.V$
    while $Q \neq \phi$
      u = EXTRACT-MIN(Q)
      $S = S \cup \{u\}$

for each vertex $v \in G.Adj[u]$
RELAX(u, v, w)

Time Complexity

The time complexity of the above code/algorithm looks *O*(V^2) as there are two nested while loops. We can observe that the statements in inner loop are executed *O*(V + E) times (similar to BFS). The inner loop has decreaseKey() operation which takes *O*(Log V) time. So overall time complexity is *O*((E+V)* (LogV)) = O(E*LogV) when Binary Heap is used for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap since Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.

Bellman-Ford Algorithm

Bellman-Ford is simpler than Dijkstra and suites well for distributed systems.

Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i = 1$ **to** $|G.V| - 1$
3 **for** each edge $(u, v) \in G.E$
4 RELAX(u, v, w)
5 **for** each edge $(u, v) \in G.E$
6 **if** $v.d > u.d + w(u, v)$
7 **return** FALSE
return TRUE

Time Complexity

But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra. The Bellman-Ford in the initialization step in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the for loop of lines 5–7 takes *O*(E) time.

All-pairs shortest-paths problem

We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. If all edge weights are nonnegative, we can use Dijkstra's algorithm, but with best improvement of using Fibonacci heap implementation for min-priority

queue, yields a running time of $O(V^2 \lg V + VE)$. If the graph has negative-weight edges, we can use Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$. Below algorithm does better than above mentioned algorithms for all-pair shortest path problems. Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms in all-pairs shortest-paths problem category use an adjacency matrix representation.

Floyd-Warshall Algorithm

This is a dynamic-programming(explore all possible paths and select the best) formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$

To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a **predecessor matrix** $\Pi = (\pi_{i,j})$, where $\pi_{i,j}$ is NIL if either $i = j$ or there is no path from i to j ,and otherwise $\pi_{i,j}$ is the predecessor of j on some shortest path from i.

This algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path p = <v_1, v_2 ... v_l> is any vertex of p other than v_1 or v_l. The Floyd-Warshall algorithm relies on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ \ldots \ n\}$ let us consider a subset $\{1, 2, \ \ldots \ k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ \ldots \ k\}$ and let p be a minimum-weight path from among them.

The below bottoms-up procedure returns the shortest path weight matrix $D^{(n)}$.

FLOYD-WARSHALL(W)
1 $n = W.rows$
2 $D^{(0)} = W$
3 **for** k = 1 **to** n
4 let $D^{(k)} = (d_{ij}^{(k)})$ be the new n x n matrix
5 **for** i = 1 **to** n
6 **for** j = 1 **to** n
7 $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
8 return $D^{(n)}$

Time Complexity

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3–7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. In this

algorithm the code is simple, with no elaborate data structures, and so the constant hidden in the Θ-notation is small.