WIKIPEDIA
The Free Encyclopedia

Yen's algorithm

In graph theory, **Yen's algorithm** computes single-source K-shortest loopless paths for a graph with non-negative edge cost.[1] The algorithm was published by Jin Y. Yen in 1971 and employs any shortest path algorithm to find the best path, then proceeds to find $K - 1$ deviations of the best path.[2]

Algorithm

Terminology and notation

Notation	Description
N	The size of the graph, i.e., the number of nodes in the network.
(i)	The i-th node of the graph, where i ranges from 1 to N. This means that (1) is the source node of the graph, and (N) is the sink node of the graph.
d_{ij}	The cost of the edge between (i) and (j), assuming that $(i) \neq (j)$ and $d_{ij} \geq 0$.
A^k	The k-th shortest path from (1) to (N), where k ranges from 1 to K. Then $A^k = (1) - (2^k) - (3^k) - \cdots - ({Q_k}^k) - (N)$, where (2^k) is the 2nd node of the k-th shortest path, (3^k) is the 3rd node of the k-th shortest path, and so on.
${A^k}_i$	A deviation path from A^{k-1} at node (i), where i ranges from 1 to Q_k. Note that the maximum value of i is Q_k, which is the node just before the sink in the k shortest path. This means that the deviation path cannot deviate from the $k-1$ shortest path at the sink. The paths A^k and A^{k-1} follow the same path until the i-th node, then $(i)^k - (i+1)^k$ edge is different from any path in A^j, where j ranges from 1 to $k-1$.
${R^k}_i$	The root path of ${A^k}_i$ that follows that A^{k-1} until the i-th node of A^{k-1}.
${S^k}_i$	The spur path of ${A^k}_i$ that starts at the i-th node of ${A^k}_i$ and ends at the sink.

Description

The algorithm can be broken down into two parts: determining the first k-shortest path, A^1, and then determining all other k-shortest paths. It is assumed that the container A will hold the k-shortest path, whereas the container B will hold the potential k-shortest paths. To determine A^1, the shortest path from the source to the sink, any efficient shortest path algorithm can be used.

To find the A^k, where k ranges from 2 to K, the algorithm assumes that all paths from A^1 to A^{k-1} have previously been found. The k iteration can be divided into two processes: finding all the deviations ${A^k}_i$ and choosing a minimum length path to become A^k. Note that in this iteration, i ranges from 1 to ${Q^k}_k$.

The first process can be further subdivided into three operations: choosing the ${R^k}_i$, finding ${S^k}_i$, and then adding ${A^k}_i$ to the container B. The root path, ${R^k}_i$, is chosen by finding the subpath in A^{k-1} that follows the first i nodes of A^j, where j ranges from 1 to $k-1$. Then, if a path is found, the cost of edge $d_{i(i+1)}$ of A^j is set to infinity. Next, the spur path, ${S^k}_i$, is found by computing the shortest path from the spur node, node i, to the sink. The removal of previous used edges from (i) to $(i+1)$ ensures that the spur path is different. ${A^k}_i = {R^k}_i + {S^k}_i$, the addition of the root path and the spur path, is added to B. Next, the edges that were removed, i.e. had their cost set to infinity, are restored to their initial values.

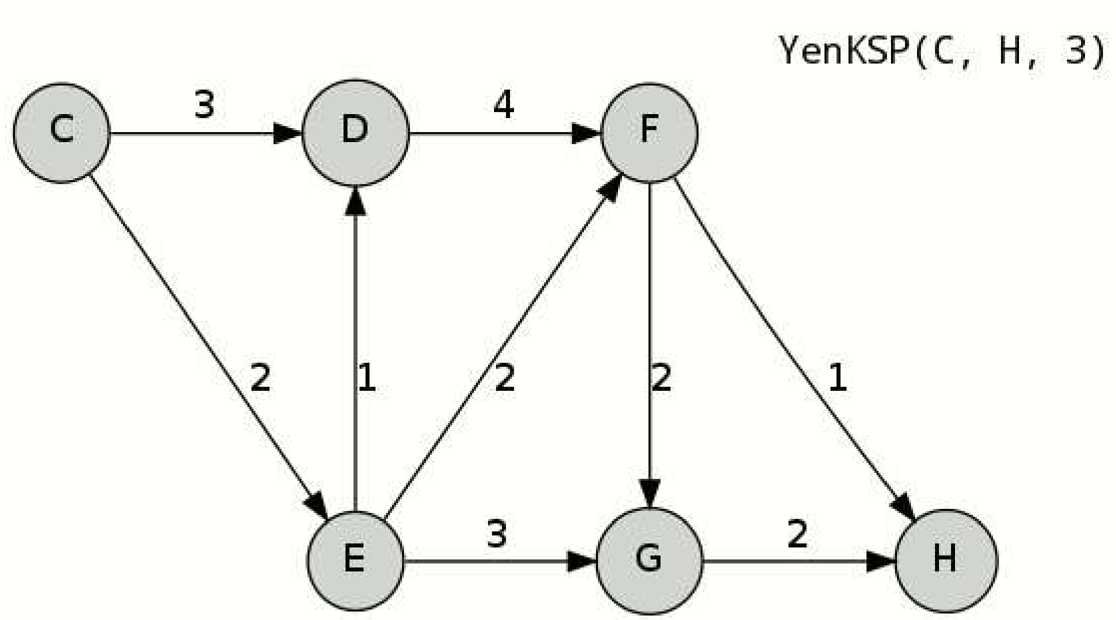
The second process determines a suitable path for A^k by finding the path in container B with the lowest cost. This path is removed from container B and inserted into container A, and the algorithm continues to the next iteration.

Pseudocode

The algorithm assumes that the Dijkstra algorithm is used to find the shortest path between two nodes, but any shortest path algorithm can be used in its place.

```
function YenKSP(Graph, source, sink, K):
    // Determine the shortest path from the source to the sink.
    A[0] = Dijkstra(Graph, source, sink);
    // Initialize the set to store the potential kth shortest path.
    B = [];

    for k from 1 to K:
        // The spur node ranges from the first node to the next to last node in the previous k-shortest path.
        for i from 0 to size(A[k − 1]) − 2:

            // Spur node is retrieved from the previous k-shortest path, k − 1.
            spurNode = A[k-1].node(i);
            // The sequence of nodes from the source to the spur node of the previous k-shortest path.
            rootPath = A[k-1].nodes(0, i);

            for each path p in A:
                if rootPath == p.nodes(0, i):
                    // Remove the links that are part of the previous shortest paths which share the same root path.
                    remove p.edge(i,i + 1) from Graph;

            for each node rootPathNode in rootPath except spurNode:
                remove rootPathNode from Graph;

            // Calculate the spur path from the spur node to the sink.
            // Consider also checking if any spurPath found
            spurPath = Dijkstra(Graph, spurNode, sink);

            // Entire path is made up of the root path and spur path.
            totalPath = rootPath + spurPath;
            // Add the potential k-shortest path to the heap.
            if (totalPath not in B):
                B.append(totalPath);

            // Add back the edges and nodes that were removed from the graph.
            restore edges to Graph;
            restore nodes in rootPath to Graph;

        if B is empty:
            // This handles the case of there being no spur paths, or no spur paths left.
            // This could happen if the spur paths have already been exhausted (added to A),
            // or there are no spur paths at all - such as when both the source and sink vertices
            // lie along a "dead end".
            break;
        // Sort the potential k-shortest paths by cost.
        B.sort();
        // Add the lowest cost path becomes the k-shortest path.
        A[k] = B[0];
        // In fact we should rather use shift since we are removing the first element
        B.pop();

    return A;
```

Example



The example uses Yen's K-Shortest Path Algorithm to compute three paths from (C) to (H). Dijkstra's algorithm is used to calculate the best path from (C) to (H), which is $(C)-(E)-(F)-(H)$ with cost 5. This path is appended to container A and becomes the first k-shortest path, A^1.

Node (C) of A^1 becomes the spur node with a root path of itself, ${R^2}_1=(C)$. The edge, $(C)-(E)$, is removed because it coincides with the root path and a path in container A. Dijkstra's algorithm is used to compute the spur path ${S^2}_1$, which is $(C)-(D)-(F)-(H)$, with a cost of 8. ${A^2}_1={R^2}_1+{S^2}_1=(C)-(D)-(F)-(H)$ is added to container B as a potential k-shortest path.

Node (E) of A^1 becomes the spur node with ${R^2}_2=(C)-(E)$. The edge, $(E)-(F)$, is removed because it coincides with the root path and a path in container A. Dijkstra's algorithm is used to compute the spur path ${S^2}_2$, which is $(E)-(G)-(H)$, with a cost of 7. ${A^2}_2={R^2}_2+{S^2}_2=(C)-(E)-(G)-(H)$ is added to container B as a potential k-shortest path.

Node (F) of A^1 becomes the spur node with a root path, ${R^2}_3=(C)-(E)-(F)$. The edge, $(F)-(H)$, is removed because it coincides with the root path and a path in container A. Dijkstra's algorithm is used to compute the spur path ${S^2}_3$, which is $(F)-(G)-(H)$, with a cost of 8. ${A^2}_3={R^2}_3+{S^2}_3=(C)-(E)-(F)-(G)-(H)$ is added to container B as a potential k-shortest path.

Of the three paths in container B, ${A^2}_2$ is chosen to become A^2 because it has the lowest cost of 7. This process is continued to the 3rd k-shortest path. However, within this 3rd iteration, note that some spur paths do not exist. And the path that is chosen to become A^3 is $(C)-(D)-(F)-(H)$.

Features

Space complexity

To store the edges of the graph, the shortest path list A, and the potential shortest path list B, $N^2 + KN$ memory addresses are required.[2] At worse case, the every node in the graph has an edge to every other node in the graph, thus N^2 addresses are needed. Only KN addresses are need for both list A and B because at most only K paths will be stored,[2] where it is possible for each path to have N nodes.

Time complexity

The time complexity of Yen's algorithm is dependent on the shortest path algorithm used in the computation of the spur paths, so the Dijkstra algorithm is assumed. Dijkstra's algorithm has a worse case time complexity of $O(N^2)$, but using a Fibonacci heap it becomes $O(M + N\log N)$,[3] where M is the number of edges in the graph. Since Yen's algorithm makes Kl calls to the Dijkstra in computing the spur paths, where l is the length of spur paths. In a condensed graph, the expected value of l is $O(\log N)$, while the worst case is N. The time complexity becomes $O(KN(M + N\log N))$.[4]

Improvements

Yen's algorithm can be improved by using a heap to store B, the set of potential k-shortest paths. Using a heap instead of a list will improve the performance of the algorithm, but not the complexity.[5] One method to slightly decrease complexity is to skip the nodes where there are non-existent spur paths. This case is produced when all the spur paths from a spur node have been used in the previous A^k. Also, if container B has $K - k$ paths of minimum length, in reference to those in container A, then they can be extract and inserted into container A since no shorter paths will be found.

Lawler's modification

Eugene Lawler proposed a modification to Yen's algorithm in which duplicates path are not calculated as opposed to the original algorithm where they are calculated and then discarded when they are found to be duplicates.[6] These duplicates paths result from calculating spur paths of nodes in the root of A^k. For instance, A^k deviates from A^{k-1} at some node (i). Any spur path, ${S^k}_j$ where $j = 0, \ldots, i$, that is calculated will be a duplicate because they have already been calculated during the $k-1$ iteration. Therefore, only spur paths for nodes that were on the spur path of A^{k-1} must be calculated, i.e. only ${S^k}_h$ where h ranges from $(i+1)^{k-1}$ to $(Q_k)^{k-1}$. To perform this operation for A^k, a record is needed to identify the node where A^{k-1} branched from A^{k-2}.

See also

- Yen's improvement to the Bellman–Ford algorithm

References

1. Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks" (https://doi.org/10.1090%2Fqam%2F253822). *Quarterly of Applied Mathematics*. **27** (4): 526–530. doi:10.1090/qam/253822 (https://doi.org/10.1090%2Fqam%2F253822). MR 0253822 (https://mathscinet.ams.org/mathscinet-getitem?mr=0253822).
2. Yen, Jin Y. (Jul 1971). "Finding the *k* Shortest Loopless Paths in a Network". *Management Science*. **17** (11): 712–716. doi:10.1287/mnsc.17.11.712 (https://doi.org/10.1287%2Fmnsc.17.11.712). JSTOR 2629312 (https://www.jstor.org/stable/2629312).
3. Fredman, Michael Lawrence; Tarjan, Robert E. (1984). *Fibonacci heaps and their uses in improved network optimization algorithms*. 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi:10.1109/SFCS.1984.715934 (https://doi.org/10.1109%2FSFCS.1984.715934).
4. Bouillet, Eric (2007). *Path routing in mesh optical networks*. Chichester, England: John Wiley & Sons. ISBN 9780470032985.
5. Brander, Andrew William; Sinclair, Mark C. *A comparative study of* k*-shortest path algorithms*. Department of Electronic Systems Engineering, University of Essex, 1995.
6. Lawler, EL (1972). "A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem". *Management Science*. **18** (7): 401–405. doi:10.1287/mnsc.18.7.401 (https://doi.org/10.1287%2Fmnsc.18.7.401).

External links

- Open Source C++ Implementation (http://thinkingscale.com/k-shortest-paths-cpp-version/)
- Open Source C++ Implementation using Boost Graph Library (https://svn.boost.org/trac/boost/ticket/11838)

-