K Shortest Path Routing for SIMON

William Willie Wells
University of Nebraska-Lincoln, Lincoln, NE 68588-0115
wwells@cse.unl.edu

ABSTRACT

Simulation is valuable. Capital expenditures prior to manufacturing a product may be reduced by using a simulator to test new concepts. Simulators are used to teach established concepts as well as being a testbed for new concepts. SIMON is a simulator for optical networks. Dijkstra's shortest path algorithm is the default SIMON routing algorithm. With the addition of the K shortest path algorithm the functional capability of the simulator is expanded by eventually allowing the network to attempt to use another path when the shortest path is unavailable.

INTRODUCTION

In accordance with Oxford Dictionaries, a simulation is an imitation of a situation or process. SIMON simulates optical networks.

There are multiple operational choices for wavelength assignment within SIMON but only a single fully implemented routing algorithm. Shortest hop path is the implemented routing algorithm in SIMON. The Shortest hop path algorithm uses Dijkstra's algorithm and positive hop distances as weights, costs. Dijkstra's algorithm is an algorithm conceived by Edsger W. Dijkstra for finding the shortest paths between vertices in a graph. This algorithm is widely used in present day routing protocols such as Open Shortest Path First. K-shortest path as well as other algorithms employ Dijkstra's algorithm as a subroutine.

K shortest path while always using a shortest path algorithm has several variants. There are many different implementations of k shortest path. A common variant used in network routing is K shortest loopless path. The aim of the algorithm is to find k (typically within [2,10]) shortest paths in a graph between a given source and destination. Individual paths are not required to be path disjoint but must be link disjoint for at least one link.

Yen's algorithm is a K shortest loopless path algorithm that operates in the absence of negative loops, which is accomplished by using positive weights.

METHODOLOGY

The c++ class kShortest is a modification of the shortest hop path routing algorithm implemented in shortest.cc. The K shortest algorithm used follows Yen's algorithm with adjustments. The method ShortestHopPath takes an integer number of vertices in the graph as one of its inputs. This input was moved to the constructor of the kShortest class. In addition to this input, an integer typed value for K is taken as the second input to the constructor. These two values are used to initiate the private dynamic arrays: visited, back_ptr, path, paths, dist, reduced_edges, paths_dist, path_dist, and route_dist. Private variables that are exactly equivalent to the input variables are initiated within the constructor as well as initiating a variable, route=0, that keeps track of how many routes have been found so far. The destructor frees the member allocated for the dynamic arrays.

The inputs to kShortestPath are in order: graph edge weights—the graph is a single dimensional float array that realizes a V-by-V matrix in accordance with c conventions, source—

an integer that represents the source node, destination—an integer that represents the destination node, and a pointer to a list of nodes—realized by a SIMON specific struct RouteElement that contains a list of nodes on the shortest path from source to destination excluding source but including destination. Source and destination vertices of the graph are copied into class specific versions, which are accessible by the other two class methods without directly passing these values to the specific class method.

An inherited #ifdef, inherited from ShortestHopPath, asserts that the values of source and destination are within [0, V-1].

The private method Dijkstra is called. The private method Dijkstra requires a graph, a source node, and a RouteElement pointer to a list of nodes to be passed as input in that order. The method implicitly uses the class destination variable stored earlier. This is done since; the K shortest path algorithm changes the source node on subsequent iterations but does not change the destination node. The number of vertices (V) is constant, although some nodes become "disconnected", so V is not passed to the method Dijkstra but set to a class variable in the constructor.

All operations within the method Dijkstra are inherited from ShortestHopPath but variables are renamed such as the bool array visited that keeps track of visited vertices. Thus, the method Dijkstra does more than the just Dijkstra's algorithm discussed above. Operations performed within the method Dijkstra that are directly related to the basic Dijkstra algorithm are initialization of distance weights, update of distance weights, and path construction. An added feature for reduced operations further in the parent K shortest path algorithm is within the path construction step of Dijkstra path distances are recorded in float array path_dist. A SIMON specific operation within the method Dijkstra is linking RouteElements into a forward linked list. The method Dijkstra returns the distance to the destination node. For the first call to the method Dijkstra this distance is stored as the first route distance in route_dist[0].

The values recorded in path_dist are copied into paths_dist which is of size kV while path_dist is of size V. Likewise, the values recorded in path are copied into paths. The arrays path and paths are integer arrays with the same array sizes as path_dist and paths_dist respectively. The array path is arranged such that the V-1 element is the destination and the path is constructed backwards through the array. For $p \epsilon [1, V]$, a path of size p terminates at V – p at the source node. For all $v \epsilon [0, V-p]$, path[v] is set to the source node value while path_dist[v] is set to 0.

A loop from route $\epsilon [2, k]$ is entered. The route_dist for the current route is set to the MAX_NUM. A comparison of the previous routes distance versus the MAX_NUM is performed. If the distance is < MAX_NUM, a search for the current next shortest route is commenced, otherwise the remaining routes are set to MAX_NUM and the route_dist array is returned.

The initial spur is set to the source node. The initial next_spur is set to the destination node. The graph is copied into a graph that will potentially have edges set to MAX_NUM, which effectively disconnects the edge from the graph. If all the edges of a vertex are set to MAX_NUM the vertex is effectively disconnected from the graph.

A loop from the source to the vertex prior to the destination node in the previous path is entered.

A loop from $p \epsilon [1, V-2]$ to search for the next spur, p = V-1 is the last element and there is no next node past that element. In the case where the previous path is s→d, spur and next_spur are already set nothing to be done. In the case where there is no path from s to d, the previous

loop is not entered. For all other path lengths there is no need to search the p = V-1 element, since the comparison if the spur vertex is not the source searches for itself in the paths array and sets the next element in the array to the next spur. The p = 0 case does not need to be searched since the next_spur will never be the source vertex (s) and the paths[0 + (route-1)*nodes]=s for all possible paths as discussed above. As hinted at, two search conditions are checked while searching for the next_spur if the spur is not s and the current element in the previous path is the spur, then the next element in the previous path is the next_spur. The root element is set to the current element if these conditions are met, and the loop is broken out of. If the spur is s and the current element in the previous path is not the spur, then the current element in the previous path is the next_spur. The root element is set to the previous element, which is the last s value in the previous path, and the loop is broken out of.

Both edges spur→next_spur and next_spur→spur are set to MAX_NUM removing both directed edges from the copied graph.

The method Dijkstra is called with inputs: the modified copied graph, source=spur, and next_route_ptr in that order. The distance from the spur to d is returned.

A comparison of the spur path dist + the root path distance—that was recorded by the previous route in paths_dist and indexed with the root element found while searching for the next spur—with the current route_dist is executed. If the sum is < than the current route distance, the current route distance is overwritten with the sum, the current paths and current paths_dist arrays are overwritten with the previous path up to the root element. The remaining current paths array entries are overwritten with the path from the root index to d. The remaining current paths_dist array entries are overwritten with the sum of the current path_dist and the previous paths root element distance from the root element to d.

The spur node is then disconnected from the graph by setting all incoming and outgoing edges equal to MAX_NUM. The spur vertex is set equal to the next_spur. If the spur vertex is not d the search for the next potential path commences otherwise route is updated and if route < k the search for the next shortest route commences otherwise the route_dist array is returned.

If it is necessary to retrieve an alternative path from the shortest path, the method alternativePath can be called. The method alternativePath requires an integer value of the requested path and a pointer to a list of RouteElements as inputs.

A check is made to verify that at least one path was found and that the requested route < k, requested route ϵ [1, k-1]. The RouteElement pointer to a list of nodes output by the kShortestPath method is treated as route 0 in the method alternativePath.

A search for the last s in the paths array corresponding to the requested route is conducted. The first RouteElement in the RouteElement list of nodes is initialized. A list of RouteElements is generated from the path data stored in paths. The route_dist of the requested route is returned.

DISCUSSION

Since the SIMON simulator is written in c++, it is only fitting that the k shortest routing algorithm for SIMON be written in the same programming language. The author currently uses c++ on a weekly basis.

Compared with other implementation of K shortest path no extra space is used for a potential shortest path heap, therefore, no separate find minimum or sort function is needed. Without a potential shortest path heap there is no early out of this implementation of K shortest path if equal shortest cost paths equivalent to the number of remaining paths to find is found.

Instead a comparison is performed on each potential shortest path cost and the previously found minimum cost. If new cost < previous cost the current, nth (n ϵ [2, k]) path is overwritten with the new path and the minimum cost is updated. Additional memory is used to record path distances. This saves space for graphs with low k values and high potential paths per spur vertex.

An additional float array with edge weights is used. For each route ϵ [2, k], the original edges are copied into this array at the beginning of the search. Deleted edges are given an edge cost of MAX_NUM. Thus, the only action to restore the graph and follow a spur path up the most recently found path is to copy the original graph into the working graph. No additional bookkeeping is required to reset the graph.

An additional struct RouteElement pointer to a list of nodes on the shortest path from source to destination excluding source but including destination is used and reused for routes in [2, k]. This allows the output to remain the same as in the ShortestHopPath method but allocate minimal additional space. Though, this may be improved if the RouteElement linking was removed from the Dijkstra method and made its own public method called at the end of kShortestPath and by alternativePath. This hypothetical method could even replace alternativePath and one might desire to rename the method something similar to kthPath or nthPath. Thus, a RouteElement pointer to a list of nodes would not be required to be passed to the Dijkstra method and a second struct RouteElement pointer to a list of nodes would not be necessary.

The time complexity of this implementation is O(worst case Dijkstra*(worst case nodes in shortest path – 1)*k). Worst case Dijkstra = $O(V^2)$. Worst case nodes in shortest path = V, thus V -1 spurs. Therefore, worst case is $O(kV^3)$.

CONCLUSION

Having an alternative routing path available decreases blocking probability, increases network utilization, and thus in a non-simulated scenario increases provider revenue. It is beneficial to step back and write a description of code you have written such as in the methodology section. Corner cases, better implementations, and bugs may become more noticeable.

FUTURE WORK

Next steps include modifying the SIMON code base to accept alternative route routing when a route is blocked and only in the case where K shortest path routing is selected as the routing algorithm. Other full integration steps with SIMON unforeseen by this author may be required for perfect simulation of the real operation.

REFERENCES

[1] [1]E. Hadjiconstantinou and N. Christofides, “An efficient implementation of an algorithm for finding K shortest simple paths,” Networks, vol. 34, no. 2, pp. 88–101, Sep. 1999.
[2]E. L. Lawler, “A Procedure for Computing the K Best Solutions to Discrete Optimization Problems and Its Application to the Shortest Path Problem,” Management Science, vol. 18, no. 7, pp. 401–405, Mar. 1972.
[3]Jin Y. Yen, “Finding the K Shortest Loopless Paths in a Network,” Management Science, vol. 17, no. 11, pp. 712–716, Jul. 1971.
[4]D. Eppstein, “Finding the k Shortest Paths,” SIAM J. Comput., vol. 28, no. 2, pp. 652–673, Jan. 1998.
[5]B. Ramamurthy, D. Datta, H. X. Feng, J. P. Heritage, and B. Mukherjee, “SIMON: a simulator for optical networks,” 1999, vol. 3843, pp. 130–135.