CHAPTER 3
A TIME-DEPENDENT *k*-SHORTEST PATH ALGORITHM FOR ATIS APPLICATIONS

3.1. Extension of a Static *k*-SP Algorithm to the Time-Dependent Case

Kaufman and Smith [1993] showed that under the consistency assumption, any static LS or LC algorithm can be extended to the time-dependent case (using the time-space network formulation). This section describes the extension of the static LC *k*-SP algorithm to the deterministic time-dependent case, by introducing time-dependent arc costs. Thus the labels (*k*-vectors) of the nodes now become time-dependent. This can be done trivially by storing these vectors for every time $t \in S$ under consideration. The problem can be solved for two cases. We can find the SPs either from a common origin (forward formulation) to other nodes, or from each node to a common destination (backward formulation). The problem statement for the backward formulation is stated below.

Given a network *G*, having |*N*| nodes and |*A*| arcs, a distinguished destination node *f*, a set of time-dependent link delay functions $d_{ij}(t)$ associated with each arc $(i,j) \in A$, and a discrete time set *S*, find the *k* shortest paths from each node in *N* to the destination node *f*, for every starting time $t \in S$.

This problem can be solved by using a hybrid version of *DYNASMART*'s TDSP and *k*-SP algorithms [Mahmassani, 1994]. Defining $L_j^k(t)$ be the label value associated with the k^{th} time-dependent shortest path from node *j* to destination *f*, starting from node *j* at time *t*, a brief statement for such an algorithm that finds these time-dependent *k*-SP is given below.

1. Initialize labels.

$L_f^k(t) = \{0, \infty, ..., \infty\}$ and

$L_i^k(t) = \{\infty, ..., \infty\}, \forall i \neq f, k = 1, 2, ..., K, t \in S.$

Insert *f* into a scan eligible list (SEL). Let $i = f$.

2. Correct labels.

$_j(t) = d_{ji}(t) + L^k_i[d_{ji}(t)]\ \forall j \in \Gamma^{-1}(i),\ \forall t,\ k = 1, 2,.., K$ (i = first node in SEL).

where $_j(t)$ represents the tentative label value for node j for starting time t.

Retain the k smallest values among $L^k_j(t)$, $\lambda_j(t)$. If i has no unscanned label, delete i from SEL. If any component of j has changed, insert j into SEL, if it is not already there. If SEL is empty, stop. The label values give the k-shortest paths from every node to destination node f, for all times $t \in S$. Else go to Step 2.

Note that in this algorithm, the label correcting step includes an extra set of calculations for each k-component of the nodes in the candidate list. These calculations involve $O(k)$ comparisons to insert the label in the tentative k-vector. We can use dynamic programming arguments to demonstrate the convergence of this algorithm. Next, we consider the forward formulation of the time-dependent k-SP problem. The forward formulation behaves in a slightly different manner. The procedure is similar to a simple LC algorithm, but does not utilize dynamic programming calculations like the backward formulation, where label values from previous calculations are used. This algorithm is in fact only an approximate method for finding the TDSP if the delays are non-FIFO. On the other hand, it has some advantages over the exact backward formulation extension of TDSP algorithm discussed earlier. To illustrate these and other features, first consider the problem statement.

Given a network G, having $|N|$ nodes and $|A|$ arcs, a distinguished source node s, a set of time-dependent link delay functions $d_{ij}(t)$ associated with each arc $(i, j) \in A$, a discrete time set S, find the k shortest (simple) paths from the source node s to each node in N.

In the forward formulation, the problem can be solved separately for each starting time $t_s \in S$ from the origin node *s*. Hence, there is no correlation in the time-components for any node. Herein lies one advantage of the algorithm suggested in this report. If we have a large network with a large time set *S*, discretizing *S* would result in a very large number of time components, and therefore an extremely large number of function evaluations if we were to use any backward (dynamic programming) formulation if we were to solve the special case of finding the *k*-SP from *s* to *f*. Also, the backward formulation will determine the optimal, but not a simple shortest path. Define $L^k_j(t_s)$ be the label value associated with the k^{th} time-dependent shortest path from origin node *s* to node *j*, starting from the origin at time $t = t_s$, where $t_s \in \{t_0, t_0+\delta, \ldots, t_0+(M-1)\delta\}$, and where δ represents the smallest time-period that can be used update the state of the network without significant loss in accuracy. Then, the stepwise procedure for the algorithm can be stated as follows.

1. $L^k_s(t_s) = \{0, \infty, \ldots, \infty\} \forall\, t_s$.
   $L^k_i(t_s) = \{\infty, \ldots, \infty\}$, $\forall i \neq s$, $\forall\, t_s$, and $k = 1, 2, \ldots, K$.
   Insert *f* into the scan eligible list (SEL). Let $i = s$.

2. For each $j \in \Gamma(i)$, $\forall t_s$, $k = 1, 2, .., K$ (i = first node in SEL), compute
   $_j(t) = d_{ij}(L^k_i[t_s]) + L^k_i[t_s]$.

Retain the *k* smallest values among $L^k_j(t_s)$, $\lambda_j(t_s)$. If *i* has no unscanned label, delete *i* from SEL. If any component of *j* has changed, insert *j* into SEL, if it is not already there. If SEL is empty, stop. The label values give the (approximate) *k*-shortest paths from the source node *s* to every node in *N*, for all (*M*) starting times t_s. Else repeat Step 2.

The algorithm (TD-*k*SP) proposed in this report sacrifices (theoretical) accuracy for speed in certain situations and is especially advantageous under the following assumptions.

1. An optimal set of paths is needed only for a few starting times from the origin node *s*.

2. The cardinality of the discrete time set *S* is extremely large compared to the value of *k*. We are (also) interested in finding the shortest (simple) path from *s* to *f* for each given starting time (t_s) from the origin node *s*.

Algorithm TD-*k*SP can be effectively used to solve the following problems.

1. To determine the optimal time for an user to enter the network.

To solve this problem, we set $k = 1$. The algorithm behaves like a discretized version of Dreyfus' algorithm by implicitly enforcing the consistency assumption in the resulting label-correcting procedure. Based on the optimal arrival times (via simple paths) at the destination node *f*, we can determine the optimal starting time at the source node *s*. Note that this case is similar to the UW algorithm of Orda and Rom discussed earlier.

2. Determine the shortest paths for a given starting time t_s from the source node *s*, given that $|S|$ is very large (greater than 1000, say).

To obtain a simple path, we set $k = 1$ for the algorithm. In general, this algorithm may not always find the optimal path if it is non-simple, but interestingly enough, we can solve another related hypothetical problem that can be used to derive a heuristic that can be used to obtain an approximate solution to the TD *k*-SP problem.

Determine the shortest path from the source node *s* to all other nodes in the network, for a given starting time t_s given that $|S|$ is very large, and the link-delays are such that a waiting time $> w_i$ time units beyond the earliest arrival time via a simple path at any node *i* will result in unnecessary delays. Also, let r_i (>0) be the minimum possible time to return to a node at any time.

We set k = max. $\lceil (w_i+1)/r_i \rceil$, noting that in the absence of ties, the k-SP values for any node i form the set of k different arrival times $T^k_i = \{t^0_i, t^1_{i,}.., t^{k-1}_i\}$, where t^0_i is the earliest possible arrival time and t^{k-1}_i is the arrival time for the k^{th} shortest path. For any arc(i, j), the optimal arrival time (possibly via a non-simple path) at node i, is necessarily contained in the set T^k_i, since any later arrival at node i would result in an unnecessary delay in arrival at j. Hence, t^0_j (in the sorted set T^k_j) represents the shortest (possibly non-simple) path. Of course, in general, one may not always be able to impose a tight bound on waiting times or return times. In the worst case, we may have to settle for the inequality $0 \leq w \leq |S|$. The algorithm is generally efficient for small values of w (<10), while for the general case, the optimal solution represents a set of shortest paths obtained using a waiting time restriction of k+1 units and need not be a true optimal solution. To illustrate this concept, consider the dynamic network depicted in Figure 4. The only time-dependent link delay is associated with the link from node 3 to node 4. The earliest possible arrival time at node 3 = 1.The value w_3 can be calculated[1] by finding the smallest t such that

$$t + d_{34}(1+t) \geq d_{34}(1).$$

Substituting the expression for $d_{34}(t)$ in this inequality, the value of t can then be obtained by solving the equation

$$t+1+(4-t)^2 = 17.$$

Discarding the trivial solution (t = 0), we obtain w_3 = 7. Also, the minimum re-arrival time at node 3 equals 4 (by traveling to node 2 and returning). Hence, the desired value of k = $\lceil 7/4 \rceil$ = 2. It can be verified that using a value of k = 2 and solving this problem using algorithm TD-kSP yields the optimal (non-simple) shortest path. In general, if a good upper bound on the value of w_i can be obtained, we can set k = max. w_i (if this value is small). Note that if max. $\lceil (w_i+1)/r_i \rceil$ = 1 for all nodes, then a value of k = 1 should be sufficient. In general, however, the algorithm can be run for some specified value of k to quickly obtain a bound on the TDSP. In addition, this formulation can be easily modified

[1] In general, a non-linear optimization method may be required to calculate w.

to develop an adaptive *k*-SP algorithm (similar to the consideration of Skiscim and Golden, 1989) that can be ultimately used to obtain the (true) time-dependent *k*-shortest simple paths using a suitable modification of the method described in Lawler [1976].

3.2. **Program TD-*k*SP**

The program operates in the following four stages.

1. Input (a) -Reading of network data from the file *od.dat.*
   Input (b) -Reading of link-delay data from the file *delay.dat.*
2. Preprocessing (a) - Transformation from global to local coordinates/references.
   Preprocessing (b) - Construction of the node adjacency *linked-list*.
3. Path Processing - Using the path processing data structure (PPDS) to find the final SP labels.
4. Post Processing (a) - Construct the spanning *tree data structure*.

Post Processing (b) - Output of the link numbers of any desired path to the file *out.dat*.

In addition to these components, a random network generator was developed for the purposes of testing. This network generator can generate a set of nodes with random arcs connecting these nodes. The user can specify the degree of connectedness (density) of the network desired and the number of nodes in the network. Another important feature of the program is the implementation of linked lists to store the link delays.

3.2.1. **Generation of Time-Dependent Link Delays**

The following scheme was used to model the time-dependent delay functions. For any link, the time set *S* was partitioned into a random number of sub-intervals. For each of these sub-intervals generated, a polynomial was generated having random degree and coefficients. A simple data structure consisting of a list of lists for each arc was used to implement this formulation. The first linked list stores the extreme points of each sub-interval. For each of the elements (sub-intervals) in this list, the coefficients of the

generated polynomial is stored. Using dynamic memory allocation, we can generate any number of sub-intervals, coefficients, etc. Further, we can also constrain the delay functions to be continuous at extremities of the sub-intervals, and the delays to be non-negative. Usually, a linear or quadratic delay function is sufficient for each time-subinterval. Another feature of the delay function implementation is that the nature (FIFO or non-FIFO) of delay functions can be ascertained by finding the first derivative of the (non-negative) polynomial expressions (a general delay function may not be differentiable at points where the delay values change sign). The first derivative of the polynomial expression can be calculated by a simple manipulation of the coefficients.

To test the computational performance of the algorithm, the random network generator program was used to obtain random networks. In addition, specially structured layered networks were also generated. The implementation schemes for these network generators are based on the methods suggested by Golden and Skiscim [1989] and Sherali et al. [1996]. The results obtained are presented in the next chapter.