CHAPTER 2
LITERATURE REVIEW

2.1 Deterministic Time-Dependent Shortest Path Algorithms

The general time-dependent shortest path (TDSP) problem can be stated as follows.

Given a graph G(N, A) having $|N|$ nodes and $|A|$ arcs, and a distinguished source node s and a destination node t, and a set of time-dependent link-delays $d_{ij}(t)$ associated with each arc (i, j), find the shortest delay path from s to t, starting from s at time $t = t_0$.

In the deterministic TDSP problem, the link-delay functions are dependent on arrival times at the tail node of the link deterministically, i.e., with a probability of one.

2.1.1 A Dynamic Programming Formulation for the TDSP Problem

One of the first papers to appear in the literature on TDSP algorithms appears to be by Cook and Halsey [1958]. They extended Bellman's principle of optimality for dynamic programming [1957] to time dependent shortest paths. The problem is formulated as shown below.

Given a set of link delays $d_{ij}(t)$ between nodes i and j for time t, find the shortest path from node i to node N starting at a time $t = t_0$.

The solution for the static delay case can be obtained using Bellman's principle of optimality. Bellman has established that there exist quantities f_i, where f_i are the lengths of the optimal paths from i to N, for $i = 1,.., N -1$, $f_N = 0$, for the static case, where the link delays are independent of arrival times at the tail node of the link. He proved that $f_i = \min (d_{ij}+f_j)$ after initially fixing $f_N = 0$, and solved the problem using an iterative scheme.

Cook and Halsey considered a discrete time set $S = \{t_0, t_0+1, t_0+2,.., t_0+T\}$ and let $d_{ij}(t)$ take any positive integer value. They fixed the upper bound T by computing the time delay caused by traversing the direct path from i to $N = d_{iN}(t_o)$. If this was not possible, they looked for the shortest two-link path, and so on, until they established an upper bound. Using this bound, they assumed that any link delay that was outside this time set could be removed from consideration and set equal to infinity. They then defined $E_i^{(k)}(t)$, for $t \varepsilon S$, to be the set of all paths of at most k links, leaving i at time t and reaching N at or before t_0+T and set $f_N(t) = 0$. The main steps of the algorithm are briefly described below (see Cook and Halsey [1957] for a complete discussion).

Initialize: $k = 0$,

$f_N^{(0)}(t) = 0$, and

$f_i^{(0)}(t) = d_{iN}(t)$, for all $i \neq N$.

Note that the value of $d_{iN}(t)$ is calculated for all values of t in the set S. Thus $f_i^{(0)}(t) = d_{iN}(t)$ if the link delay is finite for starting time $t = t_0$, and is infinite for times $t \notin S$.

Increment: $k = k+1$.

Now $E_i^{(2)}(t)$ is the set of all paths consisting of one or two links, leaving i at time t and reaching N on or before t_0+T, $i = 1, 2,...N$-1.

$f_N^{(1)}(t) = 0$

$$\mathrm{f}_\mathrm{i}^{(1)}(\mathrm{t}) = \begin{cases} \text{minimum path delay in } \mathrm{E}_\mathrm{i}^{(2)}(\mathrm{t}), \text{ if the above set is non - empty,} \\ \infty, \text{ if the above set is empty } \forall \ \mathrm{i} \ = \ 1, \ldots, \mathrm{N}\text{-}1. \end{cases}$$

Then using Bellman’s principle of optimality,

$f_N^{(1)}(t) = 0$,

$f_i^{(1)}(t) = \text{minimum}\{d_{ij}(t) + f_j^{(0)}[t+d_{ij}(t)]\}, \ i \neq j.$

Note that we have already computed $f_i^{(0)}(t)$ for all t, and hence we only need find $d_{ij}(t)$, for all $i \neq j$. We iterate for values of k from 1 upto some finite value, which is less than or equal to N. If we find that $E_i^{(2)}(t)$ is empty, it means that, all the paths having one or two links, reach N only after time t_0+T, and hence we get the value of $f_i^{(1)}(t)$ as defined. If the set is non-empty, then there are some one or two link paths from i to N starting from i at time t_o and reaching N before t_0+T, and there is a minimum among these paths. This minimal path proceeds from i to some j and then to N, unless $j = N$, in which case the validity of the equation is clear. Eventually, the vectors $f_i^{(0)}(t_0)$, $f_i^{(1)}(t_0)$, $f_i^{(k)}(t)$,..., necessarily converge to the shortest route vector. It must be noted that one need not re-compute values for previous iterates for advanced times, because we already have those values for all $t \in S$. Hence, this method requires a total of $(N^2+N)(T+1)$ values to be stored. The efficiency of the procedure depends on the value of T obtained. No implementation scheme has been reported for this algorithm. This algorithm has a worst-case time-complexity of the order $O(T^3|N|)$.

2.1.2 Solving the TDSP problem using Dijkstra's Algorithm

Dreyfus [1969] suggested that instead of using discrete time intervals and a time index set, Dijkstra's labeling algorithm can be used to calculate the TDSP from any node i to N. It must be noted that this method is valid only under the consistency assumption [Kaufman and Smith, 1990]. We shall study this in greater detail later. The Dreyfus procedure is described now.

Let X_i represent the "permanent" label for node i and Y_i represent the "tentative" label. The permanent label represents the shortest travel time from the origin to node i and the tentative label represents the current upper bound on the shortest travel time from the origin to i. The procedure is as follows.

1. $i = 1$. Let $t_1 = 0$ be the starting time from node 1 (origin).
Set $X_i = t_1$, $X_j = \infty\ \forall j$ whose labels are temporary.
2. $Y_j = \min[Y_j, X_i + d_{ij}(X_i)]$.
3. Find the minimum of $Y_j\ \forall j$ whose labels are temporary, and set X_j equal to that value.
4. If $j = N$, then stop, either use the tree built or use time differences from the destination node to construct the shortest path.
Otherwise set $i = j$.
5. Go to step 2.
The algorithm converges after at most N-1 steps.

2.1.3 The Case of Restricted Waiting at Nodes

Halpern [1977] has studied the TDSP problem for networks that have nodes with restricted waiting (RW). He specifies a set of time intervals for a node, when waiting is possible. The algorithm is a variation of Dreyfus' algorithm, but one feature that differentiates this paper from the rest is that one can use any finitely discontinuous link delay function. The problem of cycling, which is possible here, can only be finite here, due to the nature of the delay function and the RW assumption. This algorithm can also handle closing of edges for certain periods of time, etc. A brief description of the procedure is given below.

Define N^* to be the set of nodes in the shortest path, starting from origin node 1, at any iteration. Define A_j, D_j, P_j to be the time sets of feasible arrival times from a node in N^*, departure times and parking times, respectively. Then $D_j = g(A_j, P_j)$, and is proved to be finite, for finite A_j. Define N^{**} to be the set of nodes in the network that have feasible paths from N^*, i.e., arrival at any node in this set is possible if we depart from any node in N^* at a time t belonging to the set D_k ($k \in N^*$), and we can reach $j \in N^{**}$ with a finite delay.

1. Initially $N^* = \{\emptyset\}$, and $N^{**} = \{1\}$. $A_1 = \{0\}$, $D_j = \{0\}$ for all possible candidates j. Then expand the set N^{**}, by including those j which satisfy feasibility requirements. Then our goal is to find the minimum delay path, among all feasible paths, to destination node N.

2. At any iteration, let $k \in N^*$, $j \in N^{**}$. Then similar to the concept of temporary and permanent labels, find that node j, such that, among all $\{t+d_{kj}(t), t \in D_k\}$, the value for node j is minimum. We define $A_j = A_j \cup [t + d_{kj}(t)]$ (there can be cycling).

3. If $j = N$, stop; else, continue with Step 2. If the set N^{**} is empty at any stage of the solution, the problem is infeasible.

The solution is reached after a finite number of iterations, and any cycling caused due to waiting restrictions and the nature of the delay functions leads to a termination after a finite number of iterations, as arrivals to the same node will either become infeasible or impossible after a certain time t (note that $d_{ij}(t)$ is defined over the set of positive values).This algorithm can be applied for various scenarios like restricted nodes, bounded parking after arrival (as against absolute parking intervals), etc.

2.1.4 First-in, First-Out (FIFO) Delays, Non-FIFO delays and Waiting Models

Orda and Rom [1993] studied the shortest path problem in the context of unrestricted (UW) and source waiting (SW) models. They studied the assumptions under which the algorithm is valid, and prescribed an algorithm for both UW and SW for a given starting time and for all starting times. Note that the restricted and forbidden waiting models have already been discussed, based on the assumption that nothing can be gained by waiting. An important result obtained by the authors is that there exist some classes of delay functions, for which the SW model can replace an UW model. They also removed the restriction of FIFO delay requirements for links.

2.1.4.1 Link-Delay Functions

Orda and Rom first differentiate between frozen and elastic links. If we start at time $t = t_0$ from link i to j, then the link delay according to the elastic model is not $d_{ij}(t_0)$, but rather varies continuously along link (i, j) and arrives at j, at the first instance of time $t > t_o$, such that $t \geq (t_0 + d_{ij}(t))$. The behavior of the frozen and elastic link model can be made equivalent by defining the frozen link delay (starting at time t_0) as shown below.

$$d_{ij}(t_0) = \min\{ \tau \mid \tau \geq d^e_{ij}(t_0 + \tau) \}, \tau \geq 0.$$

This means that now, non-FIFO behavior is possible. Hence, the model can account for cases where one can start later from the tail node of the link, and arrive earlier than another vehicle that had started earlier. For differentiable delay functions, the first-in-first out (FIFO) assumption can be derived as follows [Malandraki, 1993].

Delay at time $t+dt$ = $d_{ij}(t+dt)$. This gives an arrival time at j = $dt + d_{ij}(t+dt)$ and must be greater than or equal to the arrival time of a person starting from i at time t, *i.e.*,

$dt + d_{ij}(t+dt) \geq d_{ij}(t)$.

Rearranging this, we obtain the necessary condition

$$\boldsymbol{d'_{ij}(t) \geq -1}$$

where $d'_{ij}(t)$ is the slope of the delay function.

The possibility of late-start and early arrival motivates the development of optimal waiting models. The various cases of waiting that can occur in practice are summarized in Table 1.

Table 1. Waiting Models for TDSP Algorithms.

Model	**Waiting Restrictions**
UW (unrestricted waiting)	Waiting at any node is unrestricted.
SW (source waiting)	Waiting is allowed only at origin and no other node.
FW (forbidden waiting)	Waiting is not allowed at any node in the network.
RW (restricted waiting)	Waiting is restricted at nodes during specified by time intervals.

Now, we can define total delay accrued as the sum of waiting delay and traversal delay.

$D_{ij}(t, w) = w + d_{ij}(t+w)$,

where

w = waiting time at node i, and

t = time of arrival at node i.

Our aim is to find an optimal waiting time which minimizes the total delay (waiting time plus the travel delay) D. This leads to our first restriction on delay functions. They must be such that optimal waiting times can always be found. This is true for continuous functions and piecewise continuous functions (by making them pseudo-continuous, stipulating that the delay at a point of discontinuity be equal to the minimum of the limit from the left and right, at the point of singularity). Also, one can define topological paths, traversal paths and waiting schedules. A topological path is simply the ordered set of links in the path, with no reference to waiting delays. A traversal path is an ordered set of topological paths and waiting schedules. Next we define a concatenated path as one which has each of its sub-paths also optimal. If an optimal topological path and the corresponding optimal waiting schedules is also concatenated, then the entire traversal path is said be concatenated.

2.1.4.2 An Algorithm for the UW Model

The algorithm for the UW model for a given starting time is the same as the Dreyfus' algorithm, except that instead of $d_{ij}(t)$, we use $D_{ij}(t)$, i.e., the minimum traversal path, rather than the minimum topological path. Cycling is not a problem because there is no restriction on waiting, and therefore, an equivalent (FIFO) FW model can be constructed and solved. Orda and Rom prescribe an algorithm for all starting times t by defining the labels to be time-dependent and the delay to be given by $X_k(t)$-t. Here, one cannot identify a node that serves as the center of operations for that and only that operation, and one must use an algorithm that considers all nodes, rather than only those nodes that have only temporary labels.

2.1.4.3 The FW Model

The shortest path may neither be simple (a path having non-repeating nodes) nor concatenated. To demonstrate this, consider the network in Figure 4. We have to find the TDSP from source node 1 to destination node 4, starting from node 1 at t = 0. Let t_i denote the arrival time at node i.

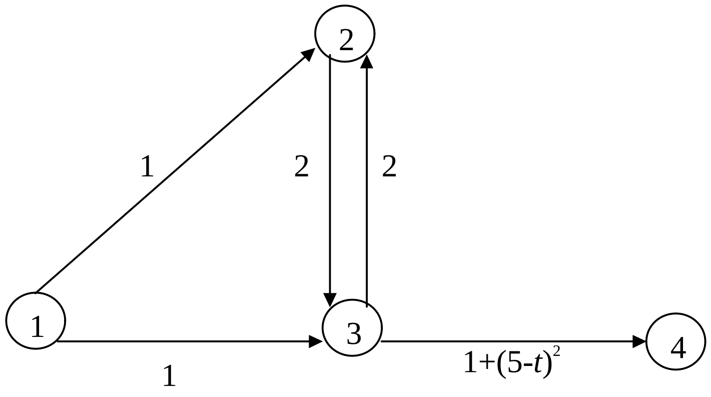


Figure 5. A Network Example with a Non-FIFO Link-Delay.

All links except (3, 4) have static delays. If waiting is forbidden at any node, the shortest simple path solution to this problem is {1-2-3-4} for a total delay of 8. Note that the travel delay on (3, 4) is non-FIFO and monotonically decreases for $t_3 \leq 5$. Hence, it may be beneficial to arrive at node 3 at some suitable time, to reduce the delay on (3, 4). The correct TDSP solution to this problem is {1, 3, 2, 3, 4}, yielding a total delay of 6. Note that this path is non-simple and the cycle in this path contains only a finite number of loops, because, for $t_3 > 5$, there would be a monotonic increase in delay on (3, 4) and further cycling would be of no benefit. An algorithm that can efficiently solve this problem is presented in Section 3.1.7.1. The authors have also shown that Halpern's model may not work unless the restrictions on delay functions are tightened (using FIFO and positive delay functions).

2.1.4.4 An Algorithm for the SW model with Single and Multiple Starting Times

It can be proved that for continuous delay functions and piecewise continuous functions with negative discontinuities, there exists an SW shortest path having the same delay as an UW shortest path, for the case of a single starting time. For these functions we can adjust the waiting at the source so as to arrive at any preset time at any node in the UW shortest path. This is proved using induction, for any two nodes i, $i+1$ (it is trivially true for $i = 0$). Given a preset time of arrival $T \geq T_A(i+1)$ at node $i+1$ (we cannot arrive earlier, as we are not allowed to wait), we have to find the departure time t_d, such that $t_d + d_{i,i+1}(t_d) = T$. T_A and T_D are the arrival and departure times according to the UW model. We use the Intermediate Value theorem of calculus to prove this result as illustrated below.

Replace t_d with x, and define

$$f(x) = x + d_{i,i+1}(x).$$

As $x \rightarrow \infty$, $f(x) \rightarrow \infty$, and $f(T_D(i)) = T_A(i+1) \leq T$.

According to the intermediate value theorem, there exists a value t_d, such that

$$t_d \geq t_D(i) \geq t_A(i) \text{ and } f(t_d) = T.$$

Thus, we can find a time of departure from i for the SW case, which is a value greater than or equal to that for the UW case and can reach $i+1$ at time T. The same result is obtained if functions with negative discontinuities (only) are used (by stipulating that the left-hand limit be greater than the right-hand limit). Hence, the procedure used to obtain the equivalent SW model is to first calculate the shortest path for the UV case, and then, starting from the destination node, calculate the time of departure for the previous node and so on, until we will finally arrive at the origin node. The source delay will then be given by the difference between this arrival time and the starting time.

For the SW model and multiple starting times, we make the latest starting time as a function of time. We execute the UW algorithm, and then compute the source waiting time, for starting time t, WAIT(s, w, t) = max. $\{D \mid X_w(t+D) = X_w(t)\}$.

Since this algorithm is not valid for general piecewise functions, the authors suggest some waiting relaxations for functions having finite positive discontinuities (relaxed SW model). The authors have also proved the general time-dependent shortest path problem (where delays may be non-FIFO) to be NP-Hard[1] [Orda and Rom, 1990]. Recently, Sherali et al. [1996], have shown that this result holds good, even if only <u>one</u> arc in the network has a time-dependent delay.

2.1.5 The Time-Space Network Formulation

2.1.5.1 The Consistency Assumption

Kaufman and Smith [1990] have summarized the computational efficiency and assumptions of generalized shortest path algorithms. They first demonstrate that label setting algorithms or the Cook-Halsey procedure cannot be applied in the absence of the

[1] NP-Hard is a term that arises in the context of computational complexity. An (optimization) problem that is NP-Hard almost surely cannot be solved in “polynomial-time”, unless many classes of notoriously difficult problems can also be solved polynomially. Interested readers can see [Gary and Johnson, 1979] for a complete discussion.

consistency assumption. This consistency assumption is implicitly embedded in Bellman's principle of optimality. The assumption can be stated as follows

For any $(i, j) \in A$ (the set of links), $s + t_{ij}(s) \leq t + t_{ij}(t)$ for all $s, t \in T$ (the time interval under consideration), such that $s \leq t$.

The authors motivate the creation of a time-space network (also called the expanded static network) $G_T(N_T, A_T)$ having static delays, corresponding to the dynamic network $G(N, A)$. Nodes (i, t) are created for each node i and for each time $t \in S$. Corresponding to these nodes, arcs $((i, t), (j, t+d_{ij}(t))$ are created for all $t, t+d_{ij}(t) \in S$. Thus, every node $i \in$ N, is replicated $|S|$ times. Recently, Sherali et al. [1996], have presented a procedure that generates an equivalent time-space network using only a minimal number of replications. Any label setting (LS) or label correcting (LC) algorithm can now be applied to solve the SP problem on G_T.

2.1.5.2 A Summary of Results Obtained by Kaufman and Smith

The authors have proved the following theorems (stated here without proof).

1. Under consistency assumption, one can use the expanded static network formulation to obtain the optimal path, whose sub-paths are also optimal.
2. The worst-case performance of the natural generalization of the LS algorithm has bounds equal to those for the LS for the static network.

An important result obtained by them is that, even if the assumption does not hold, the general algorithm can be used to obtain a feasible upper-bound on the actual TDSP travel time. The authors have implemented the algorithms in INTEGRATION (a software package for traffic simulation) and observed that

$t_{fd} < t_{qd} < t_{ni}$

where,

t_{fd} = travel times for vehicles following dynamic shortest paths,

t_{qd} = travel times for vehicles with quasi-dynamic shortest paths, and

t_{ni} = travel times for vehicles with no prior information.

2.1.6 User-Optimal Starting Time Considerations

Friesz et al. [1986] analyzed TDSP algorithms for the condition of source waiting. They used this method to model traffic equilibrium. They postulated that route choice is not independent of departure times (time-dependent) and calculated optimal source waiting times based on minimal cost to the user, i.e., users seek to minimize their actual costs of travel by adjusting both their routes and departure times. The delay functions are assumed to be linear (which arise out of deterministic queuing). Doing so allows them to model the consistency assumption by demonstrating that the delay function is strictly increasing.

2.1.7 An Efficient TDSP Algorithm for Practical Implementation

Ziliaskopoulos and Mahmassani [1993] and some researchers at the University of Texas, Austin, have most recently studied the TDSP problem, in the context of the development of *DYNASMART*, a software package for dynamic traffic assignment. The path processing component of *DYNASMART* incorporates the implementation of *k*-shortest path, least time path, and least cost path algorithms. A study of these algorithms offers useful insight into implementation aspects for TDSP algorithms. We shall discuss the last two algorithms in this study and develop a pseudocode for their efficient implementation.

The authors have adopted a formulation similar to the label-correcting method, but have suggested an improvement. Instead of scanning all nodes at every iteration, a list of scan eligible nodes (SE) is maintained. SE contains nodes having the potential to improve the labels of at least one other node. Thus the algorithm works like the label correcting (LC) algorithm, and the label vectors are upper bounds on the least-time paths (LTP), until the algorithms terminates. First we define the label vector $_i(t)$ to be equal to the current upper bound of travel time from node *i* to destination node *D*, and all times and for all nodes. In particular, $_D(t)$ is set equal to zero. Associated with each member in this list is the current shortest path.

2.1.7.1 The LTP Algorithm

1. Create the scan-eligible list (SE) and initialize it by inserting the destination node *D* into it. Initialize the label vectors at the following values.

Λ_D = { 0, 0,..., 0}, and

Λ_I = {∞, ∞,..., ∞} for all *i* except *D*.

2. Insert all nodes that can directly reach *D* into SE. Compute the travel times from *i* to *D*.

3. Sequentially select the nodes in the list. For example, for the first node, find all *j* that can directly reach *i*, compute new travel times from *j* to *D*, using the Cook-Halsey formula, and compare with previous values. If there is any change in at least one component of $\lambda_j(t)$, insert *j* into the SE list. Delete the node just processed from the list.

4. Check if SE is empty. If the list is empty, the algorithm has found the optimal shortest paths from all nodes to the destination node *D*, for all times *t*. Otherwise, the problem is infeasible.

It should be noted that for non-FIFO cases, the FW algorithm will find paths which may not be simple. The LTP algorithm can be used to solve the TDSP problem shown in Figure 4, as discussed previously. Recently, Sherali et al. [1996] have presented a procedure that simultaneously generates a time-space network while finding the TDSP to all nodes.

2.1.8 Modeling of Realistic Delay Functions for ATIS Applications

Koutsopoulos and Xu [1993] have studied the effect of “information discounting” on travel times of vehicles and have proposed a dynamic shortest path algorithm with information discounting. This combines the label setting algorithm and an information discounting mechanism. The label setting step of the standard algorithm is modified as follows.

$$P_{sk}(t_0) = \min\{P_{sk}(t_0), P_{sj}(t_0) + [E(T_{jk,h}) + e^{-[\quad {}_{jk,h}P_{sj}(t_0)]}(t^*_{jk,h} - E(T_{jk,h}))]\}.$$

This equation states that the shortest path time from origin *s* to destination *k* is the minimum of the temporary label already calculated and the minimum travel time from *s* to some intermediate node *j* and then the direct travel time from *j* to *k* at time *t* = *h*. This direct travel time is calculated using the information discounting mechanism, where,

$P_{sk}(t_o)$ = the projected travel time from the origin node *s* starting at time t_o to node *j* (using the shortest path from node *s* to node *j*),

$E[T_{jk,\,h}]$ = expected travel time on link (*j*, *k*) in the time period *h* (a random variable),

= an appropriate positive scalar,

$\delta_{jk,\,h}$ = standard deviation of historical travel time, $T_{jk,\,h}$,

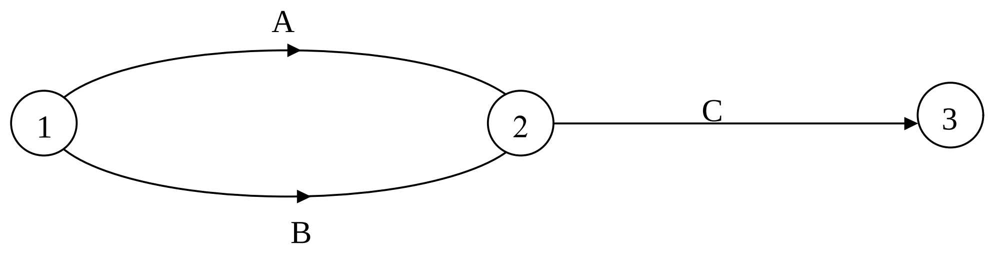
$t^*_{jk,\,h}$ = predicted travel time on link (*j*, *k*) in time period *h* (after discounting), and

h satisfies the inequality: $(h\text{-}1)\Delta t < t_o + P_{sj}(t_o) \leq h \quad t$.

The authors argue that as we move farther away from the origin, the predicted travel times become unreliable (high variance) and one would have to use expected time from historical data profiles to obtain travel time values. These values can be measured using $P_{sj}(t_o)$. Also, if the variance of the information on current travel time increases, then the expected travel time values may be useful. This model thus gives a method of calculating travel times, if such conditions exist. For example, as the distance from the origin becomes very large (and hence the variance is very large), the term $e^{-(.)}$ in the label-setting expression approaches zero, and in effect, the expected travel time is used to set labels. If the distance from the is very small and the variance is relatively smaller, then the value of the term $e^{-(.)}$ in the label-setting expression approaches unity, and the travel time effectively equals the predicted travel time alone (obtained using real-time information). This method is again valid only under the FIFO assumption. A theoretical justification for this procedure is presented in Section 2.2.3.

2.2 Stochastic Time-Dependent Shortest Paths - Hall's Procedure

In the stochastic TDSP problem, the link delays are time-dependent random values and are modeled using probability density functions and time-dependency. Here, link delays take time-dependent values based on finite probability values. Hall [1986] worked on stochastic time dependent shortest paths. He showed that for stochastic time-dependent paths, Bellman's principle of optimality cannot be applied and one has to resort to other methods for finding the shortest path. Bellman's principle of optimality fails because there may be some paths which may have a higher expected value of reaching a certain node, and since link delays depends on the time of arrival, the non-standard shortest path may have a lower expected time delay. This can be illustrated using the example shown in Figure 5. In such cases, the problem can be elegantly modeled using a conditional probability formulation for arrival time distributions, and then solving the same using a dynamic programming algorithm (See Hall [1985]).



Arc	**Travel Time**	**Probability**
A	100 minutes	1.0
B	90 minutes 120 minutes	0.5 0.5
C	30 minutes, for arrival time at node 2 before 3:35 100 minutes, otherwise	1.0 1.0

Figure 6. A Network Example having Random Time-Dependent Travel Times.

Consider the network in Figure 5. We have to find the TDSP from node 1 to node 3, starting from node 1 at 2:00. Arc A clearly has a lower expected travel time to node 2 (100 minutes on A as opposed to 105 minutes on B), and would be chosen as the shortest path by a standard shortest path algorithm. Yet, this is clearly inferior to arc B when continuing on to node 3. Because arc B has a higher probability of arriving at node 2 before 3:35 minutes, the total expected travel time to node 3 is lower (170 minutes as opposed to 200 minutes). The above example proves that Bellman's principle of optimality is not valid when travel times are both random and time-dependent. Hall suggests a branch and bound technique using a *k*-shortest path algorithm to solve the problem. This method can be used when arc travel times can be bounded from below and calculating the expected travel time on any path is difficult (as is the case for networks like the one in figure 5). The procedure combines a branch-and-bound technique along with a *k*-shortest paths algorithm. The algorithm iteratively evaluates the expected travel times of selected paths, and stops when one of the evaluated paths is found to have the minimum expected travel time.

2.2.1 The Branch-and-Bound Procedure of Hall

1. Set $m = 1$. Let the current upper-bound on expected *O-D* travel time be $t_u = \infty$. Calculate the shortest path from the origin to destination using the minimum possible link delays. Let this path be P_1.

2. Increment $m = m + 1$. Let P_m denote the m^{th} shortest path from the origin to the destination. Compute the expected travel time for $P_{m-1} = T_{m-1}$.

3. Compute (P_m) from the origin to the destination based on minimum possible link delays. Set the current lower bound on the expected travel time t_l equal to the travel time for P_m.

4. If $T_{m-1} < t_u$, set $t_u = T_{m-1}$, and let P = shortest path = P_{m-1}.

If $t_u \le t_l$, stop; P_m is optimal, with a minimum expected time of t_u.

If $t_u > t_l$, return to Step 2.

The reasoning is that, at any iteration, the expected travel time of all paths not yet evaluated must be greater than t_l, as this is a lower bound. If it turns out that t_l is greater than t_u, then the minimum expected travel time of all paths not yet evaluated must be greater than t_l (and hence t_u). Note that, since t_u is the minimum of the shortest *m*-1 paths, and less than the expected travel time of all other possible paths, it must then be the overall minimum. If t_u is greater than t_l, then it is still possible for us to find a shorter path, and we evaluate the next shortest path based on the minimum possible delay.

2.2.2 A Time-Space Network Formulation for the Stochastic SP Case

Kaufman and Smith [1990] generalized the rules for the use of LS algorithms for TDSP problems, by including the stochastic case. They prescribe the use of expected values instead of deterministic values and showed that under the consistency assumption, one may obtain results similar to the deterministic case. For the non-FIFO case, these results can be used to obtain a quick upper bound on the initial solution. They have also suggested that the lower bounds may be improved by selecting the *k*-shortest time-dependent paths. In particular, the bounds on Hall's algorithm can be tightened.

2.2.3 A Markov Chain Model for Stochastic Link Delays

2.2.3.1 Definitions

A s*tochastic process* is defined to be simply an indexed collection of random variables $\{X_t\}$, where the index *t* runs through a given set *T*. A stochastic process $\{X_t\}$ is said to be a Markovian process if $P\{ X_{t+1}=j| X_o = k_o, X_1 = k_1, X_2 = k_2, X_{t-1} = k_{t-1}, X_t = i\} = P\{ X_{t+1}=j| X_t = i\}$, for t = 0, 1,... and every sequence $i, j, k_o, k_1,..., k_{t-1}$.

This is equivalent to stating that the conditional probability of any future event, given any past event and the present state $X_t = i$, is independent of the past event and depends upon only the present state of the process. The conditional probabilities $P\{X_{t+1}=j| X_t = i\}$ are called the transition probabilities. If, for each *i* and *j*, $P\{ X_{t+1}=j| X_t = I\} = \mathrm{P}\{ X_1 = j| X_o = i\}$, for all t = 0, 1, 2,..., then the one step transition probabilities are said to be stationary and are usually denoted by p_{ij}. This implies that, for each *i*, *j*, and *m* (m = 0, 1, 2,...),

$$P\{ X_{t+m}=j|\ X_t = i\} = P\{ X_m=j|\ X_o = i\},$$

for all t = 0, 1, 2,...These conditional probabilities are usually denoted by p^t_{ij} and are called *n*-step transition probabilities. Thus p^t_{ij} is just the conditional probability that the random variable *X*, starting at state *i*, will be in state *j* after exactly *t* steps. Hence, they also satisfy the properties:

$p^t_{ij} \geq 0$, for all *i* and *j*, and t = 0, 1, 2,...

$\sum_{j=0}^{M} p^t_{ij} = 1$, for all *i* and *j*, and t = 0, 1, 2,...

A stochastic process is said to be a *finite-state Markov chain* if it has the following:

1. A finite number of states.
2. The Markovian property.
3. Stationary transition probabilities.
4. A set of initial probabilities $P\{X_o = i\}$ for all *i*.

2.2.3.2 Link Travel Times Modeled as a Markov Process

Koutsopoulos and Xu [1993], have shown that time-dependent link delays can be modeled as a Markov process. Without loss of generality, let us assume that the state of a link at time period *t*, s_t, takes five values (-2, -1, 0, 1, 2). The limiting probability $P_{st}(i)$ that the state s_t of a link in period *t* is *i* (-2, -1, 0, 1, 2) can be calculated from the distribution of normalized travel time for the corresponding period. Furthermore, we assume that the state transition from one period to the next can be modeled as a Markov process (in the absence of incidents). Furthermore, for small time periods, we can assume that the travel time can change at most one state in one time period. The state transition matrix P_t, from period *t* to *t*+1, is then given by

$$P^t = \begin{bmatrix} p^t_{-2,-2} & p^t_{-2,-1} & 0 & 0 & 0 \\ p^t_{-1,-2} & p^t_{-1,-1} & p^t_{-1,0} & 0 & 0 \\ 0 & p^t_{0,-1} & p^t_{0,0} & p^t_{0,1} & 0 \\ 0 & 0 & p^t_{1,0} & p^t_{1,1} & p^t_{1,2} \\ 0 & 0 & 0 & p^t_{2,1} & p^t_{2,2} \end{bmatrix}$$

.

The state matrix after *m* time periods is given by

$$P_1 = P^t$$

$$P_2 = P^t P^{t+1} = P_1 P^{t+1}$$

$$.$$

$$.$$

$$P_m = P_{m-1} P^{n+m-1}.$$

Here, the element $p_m(i, j)$ of the matrix P_m is the probability that the link will be in state j after m time periods (transitions) given the current state $s_t = i$ (i = -2, -1, 0, 1, 2). We can now calculate the expected travel time t^*_{t+m} after m time intervals is as follows.

$$E(T_{t+m}|s_t = i) = \sum_{j=-2}^{2} p_m(i, j) * E(T_{t+m}| s_{t+m} = j)$$

$$= \sum_{j=-2}^{2} p_m(i, j) * \int t * f_{t+m}(t| s_{t+m} = j)dt$$

where $f_{t+m}(t|s_{t+m}=j)$ is the travel time distribution given that the link is in the state j in time period $t+m$. If the transition matrix is constant over time, the predictions of travel times over the next m time periods would exhibit an exponential decay towards the expected time after m time periods. The authors use this method to justify the exponential information discounting mechanism for ATIS applications, presented in Section 2.1.8.

2.2.4 Pareto-Optimal Paths

Mahmassani and Miller [1993] discuss the case where one finds multiple shortest paths with non-zero probability (pareto-optimal paths). This makes it necessary to choose one of these paths. One would also like to know the probability distribution of the minimum path, to help us evaluate the quality of the path.

2.2.4.1 Formulation

The proposed algorithm is a specialized form of the label correcting (LC) algorithm, and in a way, similar to the LTP algorithm used for deterministic TDSP problems discussed in Section 2.1.7.1. This facilitates the formulation of the model using Bellman's principle of optimality. The model assumes forbidden waiting conditions. The LC algorithm calculates the optimal path or pareto-optimal set of paths (in terms of travel times) from all nodes in the network to destination node *N*, for every time *t* under consideration (perhaps the peak hour).

We are given the travel times for arcs for a starting time *t*, with the corresponding probabilities of occurrence, and the sum of all probabilities being unity. These probability distributions are converted to cumulative distributions (*cdf*), by sampling the distribution at discrete points (α = 0.1). The existence of pareto optimal paths necessitates the addition of another superscript '*c*' to the travel time label, which then becomes $\{\lambda^{p,\,c}(t)\}$, where

c = the number of the pareto optimal paths at any iteration = (1, 2,...χ^{i}), and

p = the set of sampling points of the *cdf*, $p \in$ (0.1,..., 1.0).

2.2.4.2 Label Correcting Step

The main step in the algorithm is the label correcting step, when one computes all possible permutations (convolution) between the arc *cdf* for arc (*i*, *j*) at time *t* and the *cdf* of the shortest path from *j* to destination node for the time (the sum of *t* and the probable arc travel time for arc (*i*, *j*) associated with the value in the *cdf*). Now, the newly calculated travel time has a different number of sampling points (10*10 = 100). So the new label is modified so as to have the same number of sampling points (10). These new label entries (i.e., possible travel times) are next arranged in ascending order. They are compared with the original set of optimal paths. Mathematically, this step is represented by

$$ {}_{i}^{q}(t) = \Omega_{i,j}^{p}(t) \underset{p,r,D}{\oplus} {}_{j}^{r,D}[t + \Omega_{i,j}^{p}(t)] $$

$${}^{p}_{i}(t) = {}^{(100p)}_{i}(t)$$

where,

${}^{q}_{i}(t)$ = the new label which contains the new set of travel times from node *i* to *N* and the associated *cdf* at time *t*.

${}^{p}_{i,j}(t)$ = the travel times and *cdf* associated with link (*i, j*) at time *t*.

${}^{p}_{i}(t)$ = the new label for node *i*, arranged in ascending order and having the usual number of sampling points in the *cdf*.

r, *p* = the *cdf* sampling points, $\in$ (0.1,..., 1.0).

$q \in$ (1, 2,..., 100).

D = the existing number of pareto optimal paths from *j* to *N*.

$i \in {}^{-1}(j) \equiv$ set of all nodes that belong to the reverse star of *j*.

2.2.4.3 Optimality Conditions

1. Check for deterministic dominance of the new path. This can be achieved by comparing the last (and hence, the largest) entry of the new label, ${}^{p}_{i}(t)$, with all the smallest values of the old label set ${}^{0.1,\ c}_{i}(t)$. If the former is lesser, then deterministic dominance is established and the label for that node is updated and it takes the value of this new label alone.

2. Check if the largest values (for all pareto-optimal paths) in the old label set is lesser than the first (and hence, the smallest) entry in the new label, ${}^{p}_{i}(t)$. If this is true, then the new label can be discarded and the number of pareto-optimal paths remain the same.

3. If neither of the two conditions are satisfied, there is no dominance in either direction and the old label set is expanded to include the new label, ${}^{p}_{i}(t)$.

2.2.4.4 Storing of Path Information in Nodes

The algorithm is a modified LC algorithm. To maintain path information, two pointers are introduced ${}_i^c$ and L_i^c. ${}_i^c$ is the (predecessor) pointer for the c^{th} label. It points to the successor node along the c^{th} pareto optimal path. But at any successor node, there may be more than one label and thus the path pointer cannot uniquely identify the path. Hence, we need another pointer. L_i^c is the label pointer, which points to the c^{th} label of node i which specifies the label at the node indicated by ${}_i^c$. Thus, to specify any particular path, the ordered paired list of ${}_i^c$ and the associated list of L_i^c is required. Except for the label updating step, and the check for optimality criteria, the implementation is the same as that of the LC algorithm.

2.2.4.5 Decision Making Strategies

They are given to select particular paths among the pareto optimal paths. These could be based on the following considerations.

1. Minimum expected value.
2. Minimum Variance.
3. Minimum combined expected value and variance.
4. Threshold value (finding the path which has the best chance of arriving before this time).
5. Label with the largest probability of having the lowest value.
6. Label with the smallest probability of having the lowest value.

One could also modify the optimality conditions by checking for stochastic dominance, rather than deterministic dominance (but, this may result in the dominated path being shorter). Note that, as the FIFO assumption is not used, non-simple paths are allowed. Also, as the number of sampling points increases, the accuracy of the discretized *cdf* increases.

The focus of the review in this section was on time-dependent shortest paths, and we mainly considered a single shortest path. Many a time, multiple shortest paths need to be in routing applications. These multiple shortest path methods are popularly referred to as *k*-shortest path algorithms. In the next section a cross-section of the more popular *k*-shortest path algorithms is presented.

2.3 *k*-Shortest Paths - Algorithms and Implementation Procedures

2.3.1 *k*-Shortest Path Problem Formulation

The general *k*-shortest paths (*k*-SP) problem can be stated as follows. Given a network $G(N, A)$ having $|N|$ nodes and $|A|$ arcs, a distinguished source node s and a destination node f, and a set of link costs c_{ij}, find the first, second,.., k^{th} shortest paths from s to f, for any user-specified value of $k \in$ 1, 2,...

k-SP algorithms have many applications in various fields and problems. The *k*-SP problem can be used in traffic assignment and routing in transportation networks. Other applications include finding multi-criteria shortest paths, solving SP problems subject to side-constraints, or determining pareto-optimal paths. The most important requirement for using the *k*-SP to solve such problems is computational speed, especially when the value of *k* is large.

2.3.2 *k*-SP Obtained as Deviations from the Shortest Path

Dreyfus [1969] presented one of the first reviews on general SP algorithms. He has listed the algorithm due to Hoffman and Pavley [1958] as the first efficient algorithm developed for solving the *k*-SP problem.

2.3.2.1 Hoffman and Pavley's Algorithm

Compute the shortest path from origin s to destination f. Define a *deviation* to be a path that coincides with the shortest path from s up to some node j (j may be the origin or the terminal node), then deviates directly to some node k (not the next node in the shortest path), and finally proceeds from k to the fixed terminal node f, via the shortest path from k. If an average node has M outgoing links, and the average shortest path contains k arcs, an average problem is solved in approximately MK additions and comparisons beyond those required for the solution of the shortest path problem.

2.3.2.2 The Dreyfus Algorithm

Dreyfus [1969] suggested a modification to the Hoffman-Pavley procedure. This modified procedure is as follows:

1. Solve the SP problem (from all nodes to destination node n).

2. Determine v_n, the second SP from n to n (possibly infinity).

3. For each node k, whose SP to n has a single arc, compare the following.

(a) The length of the SP to n deviating at k, and

(b) $d_{kn} + v_n$.

The minimum of these two quantities is v_k, the length of the second SP from node k.

4. Now consider all nodes j whose SPs to n have two links. Compare the following values.

(a) The SP deviating at j, and

(b) d_{ji} (first arc in the SP) + v_i (already determined).

The minimum of these two quantities is v_j, the second SP from j.

5. Repeat this iterative procedure till all nodes are labeled.

2.3.2.3 Bellman and Kalaba's Procedure

Another procedure described by Bellman and Kalaba was shown to be equivalent to this modified algorithm by Dreyfus. The label correcting step (proposed by Bellman and Kalaba) can be mathematically represented as:

$v_i = \min [a, b]$, where,

$a = \min_2[d_{ij} + u_j]$

$b = \min_1[d_{ij} + v_j]$ $i = 1, 2, ..., n-1$, where,

$\min_k[x_1, x_2, ..., x_n]$ = the k^{th} minimum among x.

u_i = SP from i to destination node n.

v_i = second SP from i to n.

Dreyfus showed that b can yield an overall minimum only for $j = k$, where k is the next node in the SP from i. Hence, b can be replaced by $d_{ik} + v_k$. This accelerates the algorithm originally proposed, and shows the equivalence of the two approaches. The order of computational complexity is nML, where L is the average number of iterations till convergence. For $k > 2$, this method (after being generalized for k-SPs) outperforms the original algorithm of Bellman and Kalaba.

2.3.3 Labeling Algorithms for Solving the *k*-SP

Shier [1979] has classified k-SP algorithms, in a way similar to the classification of algorithms for the single SP problem. They are either label setting or label correcting algorithms. Algorithms of the first kind are called label correcting because they start with an initial guess (upper bound) for the k-SP lengths and successively improve the tentative k-SP lengths until they correspond to the actual k-SP lengths. Algorithms of the second type are called label setting because they successively increase the number of components of the current k-vectors associated with the nodes of the network which are actually correct. It should be noted that LS algorithms require link delays to be non-negative. Shier has also studied detailed implementation schemes for both the LC and LS algorithms.

2.3.3.1 General Procedure for LC Algorithms (Forward Formulation)

LC1. Start with an initial (upper-bound) approximation to the required k-SP lengths from node 1(origin) to each node j. That is, assign a k-vector $x(j)$ to every node j, where the entries of $x(j)$ are listed in increasing order. While there are several initial approximations that could be used to begin the process, it is convenient to use a simple initial value.

$x(1) = (0, \infty, ..., \infty)$

$x(j) = (\infty, ..., \infty) \forall\ j \neq 1.$

LC2. Select a new arc and then process the arc (i, j) (see if the k vector of the node j can be improved). Here, d_{ij} has to be calculated and added to each value in $x(i)$. Compare this sum with the original $x(j)$, and replace the old value if any improvement is seen.

LC3: Check for termination criteria or continue with LC2.

Four different variations of this algorithm are discussed next. Let the inverse function, $\Gamma^{-1}(i)$ be defined as the set of head nodes j of arcs (i, j) in the forward-star FS(i).

2.3.3.2 Basic Label Correcting (BLC) Algorithm

Here, the arcs are processed in a fixed manner. thus arcs emanating from node 1 are processed before the arcs emanating from node 2, etc.

```
BLC1: $x(1) = (0, \infty, ... \infty)$
      $x(j) = (\infty, ..., \infty) \forall\ j \neq 1.$
      $i = 1.$
BLC2: If $x(i) = (\infty, ..., \infty)$ then go to Step BLC3.
          Else, $\forall\ j \in \Gamma^{-1}(i)$, process $(i, j)$.
```

BLC3: If $i < N$, $i = i + 1$, go to Step BLC2. Otherwise, check if any k-vector $x(j)$ has been changed in Step BLC2 since the last test. If there has been a change, set $i = 1$, and go to BLC 2. Otherwise, stop.

2.3.3.3 The Sequence List (SL) Algorithm

This algorithm appears to be better than the BLC method. Here, the nodes are no longer examined in a fixed order. Rather, a list of nodes is generated, and at each stage the next node in the list is examined. Once a node is examined, all arcs emanating from that node are processed, whereupon the given node is remove from SL. Nodes j enter SL (if they are not already there) on a FIFO basis. Termination occurs when SL is empty. The steps of the SL algorithm are as follows:

SL1: $x(1) = (0, ..., \infty)$

$x(j) = (\infty, ..., \infty)\ \forall\ j \neq 1.$

$j = 1$. Insert node 1 in the SL.

SL2: Remove the top node i from the list. Process link (i, j). If any change occurs to the label of the node j and if node j is not already in the list, then add j to the bottom of the list.

SL3: If there is another node still on top of the list, go to SL2, else stop.

2.3.3.4 The Double Sweep (DS) Algorithm

This method represents another variation in the BLC algorithm. Instead of examining the nodes in the same order, this method employs a sequence of alternating forward and backward iterations.

2.3.3.5 The Alternating Flag (AF) Algorithm

This is again a simple modification of the BLC algorithm. The examination of a node is skipped if the k-vector associated with that node has not changed since the previous examination of the node. The AF algorithm assigns a flag $T(j)$ to every node j in the network. $T(j) = 0$ indicates that no change has been made to any component of node j since the last examination of node j, and $T(j) = 1$ represents some change.

2.3.3.6 Procedure for LS Algorithms

The main difference between the LC and LS algorithms is that the LC algorithm finds the correct k-SPs simultaneously. It is not possible to recognize that a certain component of the k-vector for a certain node has the correct value until the algorithm terminates. Thus it is no easier finding the k-SP to one node than to find the k-SP to all the nodes. By contrast, the LS algorithm proceeds sequentially and at each step identifies a new component of the k-vector for a node. Such a value is said to be a permanent label, and, unlike the temporary label, does ensure an optimal label value. Thus, there may be possible computational benefits in finding the k-SP vector for a given destination node.

LS1: Set $x(1) = (0, \infty, ..., \infty)$

$x(j) = (\infty, ..., \infty) \forall\, j \neq 1$.

Put all nodes on the list and set $i = 1$.

LS2: Find the smallest temporary component for node i. Add the link delays from i to j to this component $\forall\, j \in \Gamma^{-1}(i)$, and compare with the present label value, and if possible insert the new label into the k vector for node j.

LS3: Make permanent the component of i. Remove i from the list if there are no more temporary components.

LS4: If the list is empty, stop. Otherwise, locate a node i, whose smallest temporary component is the minimum over all nodes in the list and go to Step LS2.

2.3.4 Adaptive Algorithms for a Fixed Destination *k*-SP Problem

Skiscim and Golden [1989] have suggested an improvement to the modified label setting algorithm of Dijkstra, by implementing an adaptive version of the algorithm. The computational results for this algorithm seem to be encouraging.

The authors define E_{in} to be a lower bound on the SP from i to N (terminal node). Intuitively, this refinement changes the way in which nodes are permanently labeled by making nodes closer to the destination more attractive. Assign a k-vector of labels $L_j = \{l_{ij}\}$, $i = 1, 2, ..., k$, to each node j, arranged in ascending order. The components of this vector, either temporary or permanent, are values of the k-SPs from node s (origin node) to j. To efficiently manage the labels, a priority queue $Q = \{ q_i \}$, $i = 1, 2, ..., |N|$ is defined. Q is implemented as a binary heap, whose entries are values of the temporary labels. q_1 is the minimum temporary label value. Maintain a list l at each stage of the algorithm to indicate which nodes still have temporary components left in their k-vector. Define $L'_j = \min \{ l_{ij} \}$, $i = 1, 2, ..., k$. The procedure of the adaptive algorithm is shown below.

1. Set $L_s = (0, \infty, ..., \infty)$

Set $L_j = (\infty,..., \infty)$, $\forall j \neq s$.

Insert l^1_s into Q.

Make all labels temporary and place all nodes on list l.

2. Set $t = q_1$, and call its associated node i^*. If $q_1 > 0$, set $T = t - E_{i^*N}$.

Otherwise, set $T = t$. For each $j \in \Gamma^{-1}(i^*)$, if $T + d_{i^*j}$ is smaller than at least one component of node j's vector, then retain the k smallest values among $\{l_{ij}\}$, $T + d_{i^*j}$, as node j's k vector. If L'_j is less than node j's entry into the priority queue, replace node j's entry by $L'_j + E_{jn}$ and re-order the queue.

3. Make permanent the component L_{i^*} associated with the value T.

If there are no more temporary labels for node i^*, remove node i^* from list l and its associated value from Q. In addition, if $i^* = N$, stop. L_n contains the k-SPs from node s to node N. Otherwise, replace node i^*'s entry in Q by its smallest remaining temporary component and re-order Q.

4. Go to Step 2.

Note that, for transportation networks, E could represent the free-flow shortest path time, since a shortest path value obtained for any other traffic condition would necessarily be greater than E.

2.3.5 Lawler's Algorithm for *k*-SPs having No Repeated Nodes

Sometimes, one may need to find the k-SP, where every path in the set is simple. Note that previous algorithms did not prevent paths from being non-simple. To find the simple k-SP solution may be computationally expensive. An adaptive implementation of this algorithm may help in this regard. A procedure initially described by Lawler [1976] is listed below.

Define P_i = i^{th} shortest path from s to N, i = 1, 2,..., k.

1. Find the SP from s to n.

2. Find all deviations from the SP. This is done by first excluding the last link of the SP, then the second last link, etc., till the first link is excluded. The minimum of the these deviations give us P_2.

3. Set $i = 2$. P_2 becomes the next candidate for partitioning. Now, the conditions under which this path was found must be carried along, *i. e.*, if P_i deviated at node j from some shorter path P_m, ($m < i$), the link (j, k) of the path P_m, is always excluded in the present iteration (k is the next node in the path P_m).

4. The process is repeated till $i = k$ SPs are found.

$O(kn)$ shortest paths need to be computed in determining the k^{th} shortest path. The computation time is dominated by the running time of the SP algorithm. The adaptive algorithm discussed in the previous section was applied for this case and the results were encouraging.

2.3.6 Special Forms of LS and LC Algorithms for the Single SP (*k* = 1) Case

2.3.6.1 Double Sided LS Algorithm for Finding the SP from *s* to *t*

This method uses the label setting algorithm simultaneously from *s* and *t*. The algorithm terminates when a node *i* has been permanently labeled in both directions. Let the shortest path from *s* to *t* be denoted by d_{st}. Then

$$d_{st} = \min_{i \in W}\{d_i^s + d_i^t\} = \min_{i \in W \cup V}\{d_i^s + d_i^t\} = \min_{i \in V}\{d_i^s + d_i^t\}$$

where,

W = subset of nodes that are permanently labeled, while scanning the forward stars of nodes, starting from *s*, and

V = subset of nodes that are permanently labeled, while scanning the reverse stars of nodes, starting from *t*.

This algorithm shows that the SP from *s* to *t* cannot contain any node $j \notin (W \cup V)$, unless *j* and *i* are equally close to *s* and *t*.

2.3.6.2 The Threshold Algorithm

The premise of this algorithm is that it is generally a good policy to remove a node with a relatively small label from the candidate (scan-eligible) list. The candidate list is organized into two distinct queues *Q* and *Q*', using a threshold parameter *s*. Hence, this algorithm tries to emulate the LS algorithm. *Q* contains "small" labels. The labels are removed from *Q* and new insertions are made only in *Q*'. When *Q* is empty, the entire candidate list is re-partitioned and *Q* and *Q*' are recalculated using a new *s*. A number of heuristic methods have been used to select the threshold value *s*.

2.3.7 The Work of Mahmassani and the Authors of *DYNASMART*

Mahmassani [1994] and other authors of *DYNASMART* have emphasized the efficiency of the label-correcting algorithm for finding the *k*-SPs. They have made the following observations.

1. Finding *k*-SPs with non-repeating nodes is computationally intensive and since repeating nodes are not encountered in general transportation networks, the loopless *k*-SP algorithm can be ruled out for practical (real-time) applications.

2. Previous implementation schemes which demonstrate the superiority of the LS algorithm over the LC algorithm have not used efficient schemes for the LC algorithm implementation. If an efficient scheme is developed for the LC algorithm, it will perform much better.

3. They have suggested finding parallel (disjoint) shortest paths, which will be useful in the computation of time-dependent *k*-SPs, apart from being amenable for parallel processing.

2.3.7.1 The Label Correcting *k*-SP Algorithm in *DYNASMART*

Initialization step: Set labels of all nodes equal to the upper bound value of the SP from origin *s* to *i*. This is done by assigning a *k*-vector $\lambda_j = ({}^{1}_{j,} \lambda^{2}_{j},.., \lambda^{k}_{j})$ to every node *j*, where the components of the *k*-vector are listed in increasing order.

1. ${}_{s} = (0, \infty,..., \infty)$

${}_{j} = (\infty,..., \infty), \forall j \neq s.$

2. Create SE.

3. Select the top node in SE. If there are no other unscanned components of the *k*-vector label for this node, delete the node from SE and scan it. If there is an unscanned label associated with this node, leave the node in the list. Let *i* denote the chosen node, and $\forall\ j \in \Gamma^{-1}(i^{*})$ and all $m = 1, 2,,... k$, scan node *j*. If ${}^{m}_{i}+ d_{ij} < {}^{k}_{j}$, then replace ${}^{k}_{j}$ by ${}^{m}_{i}+ d_{ij}$ and insert node *j* in SE (if it is not already there). Otherwise do nothing. If SE is empty, go to Step 5.

4. Repeat Step 3.

5. Terminate the procedure. The *k*-vector label λ_i for every node *i*, contains the *k*-SP labels from origin *s* to node *i*.

2.3.7.2 Update Path Algorithm

This method consists of updating previously computed paths at every step of the simulation. This algorithm is based on a straightforward tree traversal procedure. The main steps of the procedure are listed below.

1. Call the procedure *Build_priority_tree*().

This procedure sequentially traces, for every node path, its path to the destination node or to an already traced node-path. Thus a serial number (priority number (#)) is assigned to each node-path, so that if a node on the path *m* of node *i* points to node *j* and path *l*, then #(*i*, *m*) > #(*j*, *l*).

2. For # *ip* = 2 to *Nk*, repeat the following operation:

Set the label of the node-path with # = *ip* to be equal to the label of predecessor node-path plus the new travel time of the arc that connects the two labels.

3. Sort the labels of the paths for every node,.

4. Terminate the algorithm.

The update algorithm requires comparatively lower computational time when compared with other *k*-SP algorithms.

2.4 Implementation Issues for Shortest Path Algorithms

2.4.1 Implementation Methods for LC Algorithms

We discuss a sequence of implementation procedures of the general LC algorithm in this section. Dial et al. [1969] have done pioneering work in studying and classifying various implementation procedures for both LC and LS algorithms. Some of these procedures are described in the current and the next section.

If each arc $(u, v) \in \mathrm{FS}(u, v)$ has been examined and found to satisfy the condition $D(u) + L(u, v) \geq D(v)$, then it is unnecessary to re-examine these arcs until the "node potential" of *u* decreases. This observation is one of the primary motivating factors for storing the network in a forward star form. It is convenient to keep a sequence list of nodes whose node potentials have decreased since their forward stars were last examined. The sequence list can be managed in several ways.

1. FIFO sequence list - The list is processed by using two pointers, *s* and *e*, where *s* points to the entry whose forward star is to be examined next and *e* is the position of the last node added.

2. Two-way sequence list - The list is a node length array, called CL, identified by node numbers that are assigned as shown in Table 2.

Table 2. The Two-Way Sequence List.

Node Pointer	**Node Status**
-1	if node *x* was previously on the list but no longer in the list
0	if the node is not currently there and has never been on the list
+*y*	if the node *x* is on the list and *y* is the next node of the list
∞	if the node *x* is on the list and *x* is the last node on the list

In addition, start and end pointers, *s* and *e*, respectively, are maintained. Empirical tests show that the second method performs much better than the first method.

2.4.2 Implementation Techniques for LS Algorithms

1. A naive implementation scheme: Examine all arcs in the arc set *A* and then calculate and discard node potentials to finally arrive at the node with the smallest value.

2. **Dijkstra address calculation sort (Dial's method, 1969):** The chief feature is the use of SE. It uses an "address calculation scheme" (also known as a bucket structure). Nodes are arranged in buckets according to the value of their label. Inside the bucket, the nodes are connected by a double linked list and sorted according to their label sizes. If the label of a node is updated, this node is moved to another position in the list, or into another bucket. The most costly operation is the node insertion. This implementation can be coded using one-dimensional array pointers, each pointing to the first node of a linked list (a *N**2 array).

2.4.3 The Double-Ended Queue (*DQ*)

The previous section discussed implementation procedures for the general SP problem. In this section a detailed implementation procedure will be discussed specifically for the *k*-SP problem. Mahmassani [1994], presents a comprehensive survey of the literature on implementation aspects, including the work of Dial. Van Vliet [1978] has implemented Dial's algorithm along with other designs and concludes that the four factors that affect the efficiency of the structures used are the following.

1. Network size.
2. Mean link cost.
3. Network shape.
4. Ratio of number of links to number of nodes.

A number of structures like simple linked queues, circular lists, stacks, double linked lists, up to double ended queues have been tested. **Most of these results indicate that the double ended queue (*DQ*) tends to perform better for large sparse networks**. Ziliaskopoulos and Mahmassani [1994] recommend the use of the *DQ* for implementing the node-processing list. The implementation procedure used in *DYNASMART* is described next.

2.4.3.1 Network and Path Representation

As mentioned in the previous section, the network is stored in the forward star form. If the nodes are denoted sequentially and those in FS(i) appear immediately after the nodes in FS(i-1), only N+2A units of memory are required. The reverse star form is used when the SP tree is rooted at the destination node. Pointers are used to store the SP obtained. They point to the previous node in the path. For each element of the *k*-vector, a two dimensional pointer is defined. The first dimension holds the previous node in the path and the second dimension holds the rank (k) of the node.

The *DQ* structure is proposed for SE. One then needs to specify, for every node updated by the algorithm, if the node is currently in SE, etc. Depending on the answer, the node is inserted from a different end in the *DQ*. The internal arrangement of nodes in the *DQ* is done by using the CL implementation proposed by Dial, which is described in Section 2. 2. 2. Let dQ(i) denote the node status according to the CL implementation scheme. Then dQ(i) = 0, -1, y or infinity. The two pointers are now called *FQ* (points to the first element) and *LQ* (points to the last element). Three basic operations are required for processing DQ.

1. **Creation**: Initially, create a *DQ*. dQ(i) = 0, $\forall\ i \in \{N\} - s$, where s is the origin node, and dQ(s) = 99999 (a large number).

2. **Deletion**: Identify the first element of the list and delete it. Then set *FQ* = dQ(*LQ*).

3. **Insertion**: If *DQ* is empty, then the inserted node is both the first node and last node and dQ(*FQ*) = dQ(*LQ*) = 99999. If this list is not empty, the node is inserted at the beginning or the end of the list according to its dQ value.

2.4.3.2 Search-Sorting the *k*-Vector

When a node is inserted in *DQ*, it must carry its label value with it. This accomplished in case of LS algorithms with an ID pointer to this specific label. A similar concept is used here. Each node can have more than one label associated with it, instead of multiple instances of the same node in *DQ* having different labels. When a node is deleted and scanned, all its labels can be scanned simultaneously. This leads to substantial savings in execution time.

Label comparing is a time-intensive task in the LC algorithm. A simple linked list has been proposed to store the labels. A pointer is used to indicate the largest element in the structure. Every time a new label value is obtained for the node, it is first compared with the label associated with this pointer. If it is smaller, it enters the structure. The internal pointers are then arranged to perform the internal comparisons. Implementation schemes with labels in descending order seem to yield better results.

In the next section, a time-dependent *k*-SP algorithm that was developed in this study is presented. This algorithm combines the features of TDSP and k-SP algorithms and is especially ideal for ATIS applications.