Time-Dependent, Shortest-Path Algorithm for Real-Time Intelligent Vehicle Highway System Applications

ATHANASIOS K. ZILIASKOPOULOS AND HANI S. MAHMASSANI

An algorithm is introduced that calculates the time-dependent shortest paths from all nodes in a network to a given destination node for every time step over a given time horizon in a network with time-dependent arc costs. Unlike other time-dependent algorithms, this approach can handle networks where the travel cost is not necessarily the travel time itself. The algorithm is based on the general Bellman's principle of optimality. It discretizes the horizon of interest into small time intervals. Starting from the destination node, it calculates the paths operating backwards. A proof of the correctness of the proposed algorithm is presented. The algorithm is efficiently implemented and coded on a CRAY Y/MP-8 supercomputer and tested on a large actual street network as well as several random networks. The motivation for this study was the need to compute time-dependent shortest paths in a real-time environment in connection with intelligent vehicle highway systems. The suitability of the proposed algorithm for such applications is demonstrated.

The development of intelligent vehicle highway systems (IVHS) has brought renewed interest in the subject of shortest-path algorithms. For an IVHS control system to respond to rapidly changing traffic conditions in urban street networks, it must be able to calculate optimum routes dynamically. In particular, the calculation of shortest paths in networks with time-dependent arc costs is needed for the dynamic assignment of traffic by a central controller seeking to optimize overall system objectives. This calculation will most likely be repeated a number of times, as part of a real-time decision system, thereby requiring an efficient implementation of a fast dynamic shortest-path algorithm (*1*,*2*). In addition, the calculation of time-dependent shortest paths will be needed in other parts of the overall system. For example, routines that calculate one-to-all dynamic optimum paths are necessary in simulation models. In addition, on-board computers must be able to calculate one-to-one dynamic shortest paths for the individual needs of the driver.

The above requirements of IVHS control systems provided the primary motivation for this study. To our knowledge, no algorithm in the literature can be implemented in a way that fulfills all of the above requirements. Time-dependent, shortest-path problems are also encountered in a variety of applications in logistics and distribution. This study attempts to fill this gap by presenting and implementing a time-dependent, shortest-path algorithm in a manner that allows it to efficiently calculate paths for large street networks on commercially available computers. The period of interest (e.g., peak period) is discretized into very small intervals. Working in a label-correcting fashion, the algorithm calculates for every interval the time-dependent shortest paths from all the nodes in a network to a given destination node. The algorithm is implemented and coded on a CRAY Y-MP/8 supercomputer and tested on real streets as well as large random networks.

Department of Civil Engineering, The University of Texas at Austin, ECJ Hall 6.204, Austin, Tex. 78712-1076.

LITERATURE REVIEW AND EVALUATION

The first paper dealing with the time-dependent, shortest-path algorithms appears to be by Cooke and Halsey (*3*). These authors developed an iterative function, which is an extension of Bellman's principle of optimality (*4*), that gives the time-dependent shortest paths from every node in the network to one destination node for a set of discrete departure time steps. The travel times on the arcs are defined in multiples of a positive unit of time δ for every time step of the discrete scale $S_M = \{t_0, t_0 + \delta, t_0 + 2\delta, \ldots, t_0 + M\delta\}$. The integer number M is chosen so that the travel times are defined for any $t \in S_M$. The travel times for $t > t_0 + M\delta$ are assumed to be infinite. This assumption eliminates all paths with arrival time to a destination node beyond $t_0 + M\delta$, leading possibly to undetermined paths for some nodes and time steps. This algorithm has theoretical computational complexity $O(V^3M)$, where V is the number of vertexes in the network. However, no implementation scheme for this approach has been reported and hence no computational results are available to determine its actual performance.

Dreyfus (*5*) proposed a label setting approach that generalizes Dijkstra's (*6*) static shortest-path algorithm. This approach calculates the time-dependent shortest path between two nodes for one departure time step with the same computational effort as for the static case $[O(V^2)]$. However, if the paths from all the nodes to a destination node are sought, and for every time step, this approach has the same complexity as Cooke and Halsey's (*3*) algorithm.

An implicit assumption in the Dreyfus approach is that the first-in-first-out (FIFO) property holds on the links of the network. If this assumption does not hold, then the Dreyfus algorithm fails to detect the shortest paths. That was stated in some fashion by several authors, such as Kaufman and Smith (*2*), Halpern and Priess (*7*), Malandraki (*8*), and Orda and Rom (*9*). Orda and Rom recently proposed an approach that is not restricted to FIFO links only. This approach can identify optimum waiting times on the visited nodes when

such a waiting is allowed, or the optimum waiting time in the source node if waiting everywhere else is disallowed. However, their approach fails to efficiently find the best path if waiting is not allowed everywhere along the path.

DESCRIPTION OF THE ALGORITHM

Formulation of the Problem

Let $G = (V,E)$ be a V node finite directed graph with E directed edges connecting the nodes. Let $d_{ij}(t)$ be the nonnegative time required to travel from Node i to Node j when departure time from Node i is t; $d_{ij}(t)$ is a real-valued function defined for every $t \in S$, where $S = \{t_0, t_0 + \delta, t_0 + 2\delta, \ldots t_0 + M\delta\}$, t_0 is the earliest possible departure time from any origin node in the network, δ is a small time interval during which some perceptible change in traffic conditions may occur, and M is a large integer number such that the interval from t_0 to $t_0 + M\delta$ is the period of interest (e.g., the traffic peak period).

It is assumed that $d_{ij}(t)$ for $t > t_0 + M\delta$ is constant and equal to $d_{ij}(t_0 + M\delta)$. This is a reasonable assumption for urban transportation networks where, after the peak hour, somewhat stable travel times can be assumed. Nevertheless, it is not a restrictive assumption because M is user defined and can always be increased to include periods with variable travel times on some arcs. It is also assumed that $d_{ij}(\tau) = d_{ij}(t_0 + k\delta)$ for every τ in the interval $t_0 + k\delta < \tau < t_0 + (k + 1)\delta$. This is not a restrictive assumption, considering that by definition δ is very small. Node N denotes the destination node of interest in the network. The algorithm proposed in this paper calculates the time-dependent shortest paths from every node i in the network and at every time step t to the destination node N.

At each step of the computation, denote by $\lambda_i(t)$ the total travel time of the current shortest path from Node i to Node N at time t. Let $\Lambda_i = [\lambda_i(t_0), \lambda_i(t_0 + \delta), \ldots, \lambda_i(t_0 + M\delta)]$ be an M-vector label that contains all the labels $\lambda_i(t)$ for every time step $t \in S$ for Node i. Every finite label $\lambda_i(t)$ from Node i to Node N is identified by the ordered set of nodes $P_i = \{i = n_1, n_2, \ldots, n_m = N\}$.

According to Cooke and Halsey (*3*), $\lambda_i(t)$ is defined by the following functional equation:

$$\lambda_i(t) = \begin{cases} \min_{j \neq i}\{d_{ij}(t) + \lambda_j[t + d_{ij}(t)]\} & \text{for } i = 1,2,\ldots,N-1; t \in S \\ 0 & \text{for } i = N; t \in S \end{cases} \tag{1}$$

A modified version of this equation is the building block of the approach. Instead of scanning all the nodes in every iteration, a list of scan eligible (SE) nodes is maintained, containing the nodes with some potential to improve the labels of at least one other node. The proposed algorithm operates in a label correcting fashion; therefore, the label vectors are just upper bounds to the shortest paths until the algorithm terminates.

The Algorithm

Initially the SE list contains only the destination node N. In the first iteration all the nodes that can directly reach N are updated according to Equation 2 and inserted in the SE list.

$$\lambda_i(t) = d_{iN}(t) + \lambda_N[t + d_{iN}(t)] \qquad i \in \Gamma^{-1}\{N\} \tag{2}$$

where $\Gamma^{-1}\{N\}$ is the set of nodes that can directly reach N. The rest of the labels are set equal to infinity. Next, the first node i of the SE list is scanned according to the following equation:

$$\lambda_j(t) = \min\{\lambda_j(t), d_{ji}(t) + \lambda_i[t + d_{ji}(t)]\} \qquad j \in \Gamma^{-1}\{i\} \tag{3}$$

for every time step $t \in S$. If at least one of the components of Λ_j is modified, Node j is inserted in the SE list. This scheme is repeated until the SE is empty and the algorithm terminates. Relations 2 and 3 are modifications of Equation 1, which in turn is an extension of Bellman's principle of optimality (*4*).

The steps of the algorithm are as follows:

Step 1. Create the SE list and initialize it by inserting into it the destination node N. Initialize the label vectors at the following values: $\Lambda_N = (0, 0, \ldots, 0)$ and $\Lambda_i = (\infty, \infty, \ldots, \infty)$ for $i = 1,2, \ldots, N - 1$.

Step 2. Select the first node i from the SE list, name it "Current Node," and delete it from the list. If the SE list is empty, go to Step 4. Scan the current node i according to Relation 3 by examining each node j, $j \in \Gamma^{-1}\{i\}$. Specifically, for every time Step $t \in S$, check whether $\lambda_j(t)$ is greater than $d_{ji}(t) + \lambda_i[t + d_{ji}(t)]$. If it is, replace $\lambda_j(t)$ in the label vector Λ_j at position i with the new value. If at least one of the M labels of Node j has been improved, insert Node j in the SE list. The details of the structure of the SE list and the associated operations of creation, insertion, and deletion are discussed in the next section.

Step 3. Repeat Step 2.

Step 4. Terminate the algorithm. The M-dimensional vectors Λ_i for every node i in the network contain the travel times of the time-dependent shortest paths from every node i to the destination node N for each time step $t \in S$.

Without proving it, the following theorem is stated: On termination of the algorithm, every element of the vector label is either an infinite number, meaning that no path exists from this node to the destination node at the corresponding time step, or a finite number that represents the shortest path from this node and time step to the destination node.

The proof of this theorem is given by Ziliaskopoulos and Mahmassani (*10*).

IMPLEMENTATION

The implementation of this algorithm is similar to the implementation of a static label correcting algorithm. The three principal implementation issues are the network representation, the data structure of the SE list, and the path storage.

The network representation is more complicated than in the static case because travel times need to be specified for

every time step (M steps) for every arc. The most efficient way to store the network is the "backward star" structure because at Step 2 of the algorithm we need all the arcs that end at a specific node. A description of the backward star structure is given by Dial et al. (*11*). To handle the time-dependent trip times, we use the second dimension of the backward star to store pointers to an $E \times M$ matrix, where E is the number of arcs of the network. The required memory to store this structure is the minimum possible, $N + E(M + 2)$ units.

The structure of the SE list for the label correcting algorithms has been studied extensively in the literature (*11–14*). Any SE list structure is appropriate for the proposed algorithm: a simple list with any priority rule, a queue, a double-ended queue, as well as Glover et al.'s partitioning shortest-path scheme with two SE lists (*13*). In this paper, a double-ended queue (deque) structure is implemented. Deque was introduced by D'Esopo and tested by Pallotino (*14*). The deque structure allows the insertion of nodes at both ends of the SE list according to a predetermined strategy and removal from the beginning of the SE list.

The deque is implemented as suggested by Pape (*12*). A one-dimensional array, called deque, holds an integer number and can take the following values:

$$\text{Deque}(i) = \begin{cases} -1 & \text{if Node } i \text{ has been in the SE list at least once but is not there any longer;} \\ 0 & \text{if Node } i \text{ has never been in the SE list;} \\ j & \text{if Node } i \text{ is currently in the SE list and } j \text{ is the node next to it in the list; and} \\ +\infty & \text{if Node } i \text{ is the last node in the SE list.} \end{cases}$$

In addition, two pointers are kept, one pointing to the first (FIRST) and the other to the last (LAST) node in the deque.

We define the following operations associated with this structure:

- Creation: Creation is an initialization step, which is activated just once to set Deque(i) = 0, i = 1,2, . . . ,$N - 1$ and Deque (N) = ∞. Infinity is defined practically as a very large number—for example, 999,999. This operation also sets the variables FIRST = LAST = N. The whole operation requires $N + 3$ computational time units.
- Insertion: Insertion involves inserting a node at the beginning or the end of the deque. To determine the insertion point, the operation checks the value of Deque(i). If it is 0, indicating that Node i has never been in the deque, the node is inserted at the end of the SE list and the value of the pointer LAST is set equal to i and Deque(i) = ∞. If Deque(i) = -1, Node i is inserted at the end of the deque; Deque(i) is set to FIRST, and the value of FIRST = i. Otherwise, it does nothing because the node is already in the deque. The computational effort required by this step is three time units.
- Deletion: Deletion selects the first element of the deque and assigns it to the variable "CurrentNode." Then, it changes the value of the FIRST to the second element in the deque [which is the Deque(FIRST) node]. It sets the values of Deque(CurrentNode) to -1. The computational effort for this operation is three time units.

The creation operation is called only once from Step 1 of the algorithm and does not contribute significantly to the total computation time of the algorithm. On the other hand, deletion and insertion are called repeatedly from Step 2; as such, they are critical in the determination of the total computational effort of the algorithm.

Finally, the paths are maintained in an $M \times 2$-dimensional array of pointers for each node. These pointers point to the successor node and its label address. This arrangement requires $2VM$ memory locations—the least possible.

In pseudocode form the algorithm is summarized in Figure 1.

The most time-consuming part of the algorithm is Step 2 (see description of the algorithm) in which each element of the M-vector is updated for every node adjacent to the scanned node. This step corresponds to Loop 2 in the pseudocode and requires $4Md$ computational time units, where d is the in-degree of the scanned node (number of iterations of Loop 2). Inner Loop 3 can be efficiently vectorized because of the absence of interdependencies, and the number of iterations M is usually greater than 64 (the number of registers in the CRAY's vector processor), which leads to maximum vectorization speed-up (*15*).

The efficiency of the algorithm, however, depends essentially on the total number of scanned nodes before the process terminates. The lower bound on this number is the total number of nodes in the network (V), whereas the upper bound is V^2M. The upper bound is obtained by direct extension of the results for the corresponding static label correcting case. As shown in the next section, this upper bound is not a tight bound in practical applications. The complexity of the algorithm is that of Step 2 (Loop 2) multiplied by the upper bound of the number of repetitions of this step (iteration number of Loop 1) or $O(V^3M^2)$ in the general case that the maximum indegree of a node is $V - 1$.

This implementation of the algorithm was coded in the FORTRAN CFT77 language and run on a CRAY Y-MP/8 supercomputer. The results from the tests are presented in the next section.

COMPUTATIONAL EXPERIENCE

Four different sets of networks are used to test this new algorithm. Set 1 consists of five random networks with a structure similar to that of street networks and with the number of nodes ranging from 100 to 2,500. The number of time steps is held constant at 240. The travel times for each time step are generated in such a way that the FIFO property holds. Specifically, a randomly generated number is accepted as travel time for a given time step only if the absolute value of its difference from the travel time of the previous time step does not exceed the length of the interval between the two steps. Set 1 was designed to test the relation of the performance of the algorithm to the network size.

Set 2 contains five different representations of the same random network consisting of 1,000 nodes, 2,500 arcs, and varying numbers of time steps that range from 120 to 640. In Set 3, the number of arcs ranges from 1,000 to 11,500, whereas the numbers of both nodes and time steps are kept constant at 1,000 and 240, respectively. This set is used to estimate the

```
Call Creation
Call Insertion(N)
Do 1, While (SE list is not Empty)
  Call Deletion(CurrentNode)
  Do 2, For (All nodes J that can directly reach CurrentNode)
    NextNode = J
    InsertInSEList=FALSE
    Do 3, For (t=1,M)
     CurrentTravelTime=TravelTime(NextNode, CurrentNode,t)
     NewLabel=LABEL(CurrentNode,t+CurrentTravelTime)+CurrentTravelTime
     If (LABEL(NextNode,t) ≤NewLabel) Then
       LABEL(NextNode,t)=NewLabel
       InsertInSEList=TRUE
       PathPointer(NextNode,t,1)=NodeCurrent
       PathPointer(NextNode,t,2)=t+CurrentTravelTime
     End If
3   Continue
    If (InsertInSEList) Call Insertion(NextNode)
2 Continue
```

FIGURE 1 Algorithm for program time-dependent shortest path.
(continued on next page)

relationship between the execution time and the average degree of a node in a network.

Finally, Set 4 consists of one real street network—that of the core area of Austin, Texas—consisting of 625 nodes and 1,724 arcs. Time-dependent travel times for this network were produced from a simulation model called DYNASMART (Dynamic Network Assignment Simulation Model for Advance Road Telematics), for a simulated peak period of 50.3 min. This peak period is discretized into 503 intervals of 0.1 min each, and the travel time for each interval is generated.

Tables 1 through 4 present the computation times in CPU milliseconds for each set. All the runs were performed on a CRAY Y-MP/8 supercomputer, using the CFT77 FORTRAN compiler. This computer has eight CPUs with vector pipeline architecture. The algorithm is coded to allow vectorization when applicable. Vectorization is especially well suited for Step 2 of the algorithm, in which M iterations are performed because no dependency exists between any two iterations, and the number M is usually larger than the number that CRAY considers the minimum number of iterations for maximum speed-up. However, no attempt was made to exploit other hardware characteristics of the CRAY beyond vectorization. In addition, to smooth out the effect of the destination node choice on the execution time, 30 runs were performed for 30 different destinations for every network, and the average computation time is reported.

Tables 1 and 2 contain the results for Network Sets 1 and 2, which indicate that the computation time increases almost linearly with the number of nodes and the number of time steps in the network.

Table 3, on the other hand, suggests that a nonlinear relationship exists between the execution time and the average degree of a node in the network. An exponential model was calibrated from these data using regression, yielding the following relationship:

$$\text{Computation time} = 22.13\, d^{1.4}$$

where d is the average indegree of a node in the network.

Table 4 contains the averages and standard deviations of the computation time and the total number of scanned nodes for the real street network of the Austin, Texas, core area. The total number of scanned nodes is the main factor that affects the performance of the algorithm. The lower bound for this number is the number of nodes in the network (V), whereas an upper bound was found to be V^2M in the previous section. Table 4 shows that for the tested network of 625 nodes, the total number of scannings was 736, or $1.18V$, which is considerably less than the theoretical upper bound. Moreover, from the low values of the standard deviations, it can be inferred that the algorithm is reasonably stable.

Combining all the above results, we can conclude that as is common with shortest-path problems, the actual computational performance for the networks considered here is on the order of $VMd^{1.4}$, which is far from the worst-case theoretical complexity $O(V^3M^2)$. In addition, as mentioned ear-

```
1       Continue

        Procedure Creation
          Do, For (Node=1, V-1)  Deque(Node)=0
          Deque(N)=999999
          FIRST=N
          LAST=N

        Procedure Deletion(CurrentNode)
          CurrentNode=FIRST
          FIRST=Deque(CurrentNode)
          Deque(CurrentNode)=-1

        Procedure Insertion(Node)
          If (Deque(Node)=0) Then
            Deque(LAST)=Node
            LAST=Node
            Deque(Node)=999999
          Else
            If (Deque(Node)=-1) Then
              Deque(Node)=FIRST
              FIRST=Node
            End If
          EndIf
```

Where:

LABEL(Node,t) is a variable that holds the M-vector labels for every node.

PathPointer(Node,t,1) is a pointer that points to previous node while PathPointer(Node,t,2) points to the corresponding time of arrival at the previous node of the shortest path from this node to the destination node N.

InsertInSEList is a logical variable which is used to determine if a label of a node was changed.

NewLabel is an auxiliary variable that temporarily holds the new label of the next node.

FIGURE 1 (*continued*)

lier, the algorithm was vectorized efficiently. The algorithm was tested on the Austin core street network with the vectorization feature disabled for the same destination nodes as above, and the average execution time was found to be 728.02 msec. This means that the vectorization in this case yielded a speed-up of 6.74 (speed-up is defined as the ratio of the total computation time of the algorithm without vectorization to the corresponding computation time with vectorization).

Next, we compare our approach with the expanded static case proposed by Dreyfus (*5*). We implemented that scheme as efficiently as we could and achieved an execution time of 2.2 msec to find the time-dependent shortest path between

TABLE 1 Computation Times (msec) for Various Network Sizes

Network	Nodes	Arcs	Time intervals	Comp. time
1	100	250	240	5.97
2	500	1250	240	35.73
3	1000	2500	240	73.28
4	1500	5000	240	141.42
5	2500	8000	240	235.04

TABLE 2 Computation Times (msec) for Various Numbers of Intervals

Network	Nodes	Arcs	Time intervals	Comp. time
1	1000	2500	120	37.95
2	1000	2500	240	73.28
3	1000	2500	360	102.46
4	1000	2500	480	131.56
5	1000	2500	640	158.54

TABLE 3 Computation Times (msec) for Various Numbers of Arcs

Network	Nodes	Arcs	Time intervals	Comp. time
1	1000	2000	240	55.89
2	1000	3000	240	91.88
3	1000	6000	240	253.95
4	1000	9000	240	448.19
5	1000	11500	240	624.86

TABLE 4 Performance of the Algorithm on Part of Austin's Core Street Network Consisting of 625 Nodes and 1,724 Arcs for 503 Intervals

	Computation time in milliseconds	Total number of scanned nodes
Mean	107.41	736
Standard Deviation	11.82	81

one origin and one destination for one time step on the Austin core network. To compare it with our proposed algorithm, the time-dependent shortest paths must be calculated from all 625 nodes of the network to one destination and for 503 time steps for each node. This calculation would require a total time of 0.0022625503 = 691.25 sec. However, the calculation of one path for one node and one time step produces at the same time the paths to the destination from every node along the path for one time step. The maximum number of nodes in a path for the real street network was 72. Therefore, we can estimate a lower bound on the total execution time by assuming that every time one path is computed, 72 other not previously calculated paths are obtained at the same time. This lower bound is 9.6 sec, which greatly exceeds the 0.107 sec achieved with the proposed algorithm.

The proposed algorithm takes advantage of two main characteristics of networks with time-dependent arcs. One is that only a few paths between a given OD pair become best paths at any point in time. Usually, three or four paths are interchanged as best paths at different time steps, with one path often maintaining its best path status for most of the time. For example, the maximum number of paths observed during the testing of the real street network was 17 (out of a possible 503).

The second characteristic of dynamic networks is that even if various paths were best at different times between a given OD, these paths would be likely to share the same next-to-the-origin node (i.e., second node in the path). This means that most of the best paths from a given node result from the scanning of just one of the neighboring nodes. The effectiveness of our algorithm is attributed to these two reasons. Specifically, the fewer the paths that are best at different times, the closer is the behavior of the algorithm to that of static label correcting algorithms. In the extreme case that the same path remained best between a given OD pair for all the time steps, the corresponding origin node would contribute to the total computation time of the algorithm as if the network were static. Even if more than one path were best for a given OD pair, these paths could be calculated in just one scanning of a neighbor node. From Table 4, we can see that for the real street network of Austin, Texas, only 111 (736 − 625) nodes were scanned for a second time, although in general different paths were best at different times for a given origin node.

Finally, the proposed algorithm does not require the FIFO that the Dreyfus approach requires because it operates in a label correcting fashion. That makes the algorithm applicable to networks with time-dependent arc costs that arise in a variety of areas, not just transportation networks. Examples include equipment replacement policy, vehicle routing and scheduling, capacity planning, and communication networks.

SUMMARY

In this paper, a new time-dependent shortest-path algorithm was introduced. It calculates simultaneously all the shortest

paths from all nodes to a given destination node and for every discrete time step in a network with time-dependent arc costs. The algorithm is based on Bellman's general principle of optimality and can be applied to any network with time-dependent arc costs. The correctness of the algorithm was analytically established.

An implementation scheme for the algorithm was proposed, coded, and run on a CRAY Y-MP/8 supercomputer. The coded scheme was tested on a set of random networks and one real street network. The computational results demonstrated the efficient performance of the proposed algorithm for a broad range of network structures. This efficiency is attributed to the fact that the algorithm is not just an extension of a static algorithm but is designed to take advantage of the specific characteristics of networks with time-dependent arc costs.

ACKNOWLEDGMENTS

This paper was based on work funded by FHWA, U.S. Department of Transportation, through a contract entitled Traffic Modeling to Support Advanced Driver Information Systems. The computing time and the technical support provided from the Center for High Performance Computing at the University of Texas at Austin are also gratefully acknowledged.

The authors appreciate the discussions with the other project researchers, especially Ta-Yin Hu, Srinivas Peeta, and Richard Rothery.

REFERENCES

1. H. S. Mahmassani, G. L. Chang, S. Peeta, and T. Junchaya. *A Review of Dynamic Assignment and Traffic Simulation Models for ATIS/ATMS Applications*. Technical Report DTFH61-90-R-00074-1. CTR, The University of Texas at Austin, 1992.
2. D. E. Kaufman and R. L. Smith. Minimum Travel Time Paths in Dynamic Networks with Application to Intelligent Vehicle/Highway Systems. *IVHS Journal*, in press.
3. K. L. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Internodal Transit Times. *Journal of Math. Anal. Appl.*, Vol. 14, 1966. pp. 492–498.
4. R. Bellman. On a Routing Problem. *Quart. Appl. Mathematics*, Vol. 16, 1958, pp. 87–90.
5. S. E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, Vol. 17, 1969, pp. 395–412.
6. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Mathematics*, Vol. 1, 1959, pp. 269–271.
7. J. Halpern and I. Priess. Shortest Path with Time Constraints on Movement and Parking. *Networks*, Vol.4, 1974, pp. 241–253.
8. C. Malandraki. *Time Dependent Vehicle Routing Problems: Formulations, Solution Algorithms and Computational Experiments*. Ph.D. dissertation, Northwestern University, Evanston, Ill., 1989.
9. A. Orda and R. Rom. Shortest-Path and Minimum-Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, Vol. 37, 1990, pp. 607–625.
10. A. K. Ziliaskopoulos and H. S. Mahmassani. Design and Implementation of a Shortest Path Algorithm with Time-Dependent Arc Costs. *Proc., of 5th Advanced Technology Conference*, Washington, D.C., 1992, pp. 1072–1093.
11. R. B. Dial, F. Glover, D. Karney, and D. Klingman. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks*, Vol. 9, 1979, pp. 215–248.
12. U. Pape. Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem. *Mathematical Programming*, Vol. 7, 1974, pp. 212–222.
13. F. Glover, R. Glover, and D. Klingman. The Threshold Shortest Path Algorithm. *Networks*, Vol. 14, 1986, pp. 132–145.
14. S. Pallotino. Shortest-Path Methods: Complexity, Interrelations, and New Propositions. *Networks*, Vol. 14, 1984, pp. 257–267.
15. *Parallel Processing Guide. CF77*, Vol. 4. Cray Research Inc., Mendota Heights, Minn., 1991.

Publication of this paper sponsored by Committee on Transportation Supply Analysis.