CHAPTER 2

LITERATURE REVIEW

In this Chapter, we review several variations of shortest path problems and algorithms that vary by complexity and constraints. Mainly, we classify the shortest problems according to three different scenario-based types: (i) Time-Independent Shortest Path Problem, (ii) Label-Constrained Shortest Path Problem, and (iii) Time-Dependent Shortest Path Problem. For the first scenario involving the Time-Independent Shortest Path Problem, often referred to simply as the Shortest Path Problem, we provide a linear programming formulation and show how the dual problem solves this problem more efficiently. The concepts of label setting and label correcting algorithms are also introduced. Moreover, we discuss Dijkstra's algorithm and its variants used to solve problems when networks have *non*negative, and *mixed*-sign costs. For the second variant, the Label-Constrained Shortest Path Problem, we refer to a study of Barrett, Jacob, and Marathe [1998] where edges/arcs in the network are not only weighted but also labeled. The problem then becomes one of finding a shortest path in the network complying with the additional constraint that the unique *word/string of labels* given by concatenating the labels in the network along the path is an **admissible** *word/string*. Finally, for the third type, the Time-Dependent Shortest Path Problem, we show that the problem under consideration can be formulated in terms of as an equivalent linear program defined over an expanded time-space network. Accordingly, we present an extension of Dijkstra's algorithm that can be used to solve this problem. This Chapter ends with a basic description of TRANSIMS.

2.1 Time-Independent Shortest Path Problem (TISP)

Consider a network $G = (N, A)$ defined by a set N of $|N|$ nodes and a set A of $|A|$ arcs. Arc (i, j) provides a connection from node i to node j in the network G. Each arc $(i, j) \in A$ has an associated cost d_{ij}. These costs can be negative, but the network is assumed to contain no negative circuits. Given this data, we wish to find a path of minimum cost from a specified source or origin node O to another specified sink or destination node D. Alternatively, we might view the problem as sending a unit of flow as inexpensively as possible from node O to node D. This conceptualization results in the following (integer) programming formulation of the basic shortest path problem.

$$\text{Minimize} \quad \sum_{(i,j)\in A} d_{ij}x_{ij} \tag{2.1a}$$

$$\text{subject to} \quad \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} 1 & \text{if} \quad i = O \\ 0 & \text{if} \quad i \neq O \text{ or } D \\ -1 & \text{if} \quad i = D \end{cases} \tag{2.1b}$$

$$x_{ij} = 0 \text{ or } 1 \qquad \forall\, (i, j) \in A. \tag{2.1c}$$

The binary restrictions on the variables x_{ij} indicate that each corresponding arc is either selected to be in the path or not. Constraint (2.1b) ensures the conservation of flow at each node, as the unit of flow traverses a path from node O to node D. Note that the column associated with each variable x_{ij} in the constraint matrix corresponding to (2.1b) contains a "+1" in row i, a "-1" in row j, and zeros elsewhere. Due to this feature, the constraint matrix enjoys the total unimodularity property (see Bazaraa, Jarvis, and Sherali [1990]), which implies that at any extreme point optimal solution to the continuous relaxation to the problem in which (2.1c) is replaced simply by $x_{ij} \geq 0 \ \forall (i, j) \in A$, the variable x_{ij} would automatically take only integer values. Hence, the "$x_{ij} = 0$ or 1" requirement can be equivalently replaced by the constraint $x_{ij} \geq 0$. Thus, we may solve the integer program as the following linear program.

$$\text{Minimize} \quad \sum_{(i,j)\in A} d_{ij} x_{ij} \tag{2.2a}$$

$$\text{subject to} \quad \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} 1 & \text{if} \quad i = O \\ 0 & \text{if} \quad i \neq O \text{ or } D \\ -1 & \text{if} \quad i = D \end{cases} \tag{2.2b}$$

$$x_{ij} \geq 0 \qquad \forall (i, j) \in A. \tag{2.2c}$$

Let us write the dual problem for (2.2), which as we discuss below, provides a mechanism for developing a more efficient solution method. Denoting the dual variable corresponding to the constraint for node i in (2.2b) by w_i, the dual formulation can be written as shown below.

$$\text{Maximize} \quad w_O - w_D \tag{2.3a}$$

$$\text{subject to} \quad w_i - w_j \leq d_{ij} \qquad \text{for all } (i, j) \in A \tag{2.3b}$$

$$w_i \quad \text{unrestricted} \qquad \forall\, i. \tag{2.3c}$$

For convenience, denote $v_i \equiv -w_i$. As shown in Bazaraa et al. [1990], for example, we can set $v_O = 0$, and then solve the dual, where at optimality, v_i would represent the shortest path value from the origin node O to node i. Hence, this formulation (2.3) yields the shortest path from node O to all the nodes in the network. While the shortest path problem can be solved as a min-cost network flow problem, specialized labeling algorithms have been developed to solve the problem more efficiently. These algorithms utilize a label for each node that corresponds to the tentative shortest path length v_i to that node. The algorithm proceeds in a way such that these labels are improved until the shortest path to a destination node is found. There are two types of labeling algorithms: **label setting and label correcting** (see Bazaraa et al. [1990]). The label-setting algorithm designates one label as permanent (optimal) at each iteration, thus determining the shortest path to some new node at each step. On the other hand, the label-correcting algorithm considers all labels as temporary, correcting the labels to revise the shortest path value estimates, until at the final step, all labels become permanent. The label-correcting algorithms are more general and apply to all classes of problems, including those having negative arc costs.

Dial, Glover, Karney, and Klingman [1979] describe the salient features of these two labeling algorithms as follows:

The problem of finding the shortest paths from a given node O to all other nodes in a network $G = (N, A)$ may be stated as that of finding a minimum tree $T(N_T, A_T)$ of G rooted at node O, when N_T and A_T respectively represent the nodes and arcs of T. Both label setting and label correcting algorithms typically start with a tree $T(N_T, A_T)$ such that $N_T = \{O\}$ and $A_T = \emptyset$.

A label setting algorithm then augments N_T and A_T respectively, by one node $q \in N$ and one arc $(p, q) \in A$ at each iteration in such a manner that $p \in N_T$, $q \notin N_T$, and the unique path from O to any node in T is a shortest path. A label setting algorithm terminates when all arcs in A which have their starting endpoints in N_T also have their ending endpoints in N_T.

A label correcting algorithm, on the other hand, always exchanges, augments, or updates arcs in A_T in a manner that replaces or shortens the unique path from O to q in T, but does not guarantee that the new path is a shortest path (until termination occurs).

2.1.1 Shortest Path Algorithm for a Network having Nonnegative Costs

Dijkstra's algorithm is a label-setting algorithm, which is a very simple and efficient procedure to determine the shortest paths when all $d_{ij} \geq 0$. Moreover, this algorithm also yields shortest paths from node O to all of the other nodes in a network (see Bazaraa et al. [1990]).

Mathematical Terminology and Definition

O is the origin node.

i is any node in the network.

$\boldsymbol{v_i}$ is a shortest path value (estimate) corresponding to node *i*.

$\boldsymbol{d_{ij}}$ is a cost associated with arc $(i, j) \in A$.

DOWN (ᵡ) label: if (i, j) is an arc included in the current estimate of the shortest path, we set DOWN $(j) = i$. Correspondingly, we say that the **predecessor** of node *j* is node *i*.

X is a set that contains “node *O* (the origin node)” and “any other nodes in the network *G* for which the shortest path has currently been determined, but not node *D* (the destination node)”.

N is a set of all nodes in the network *G*.

$\boldsymbol{\overline{X}} = N - X$, is a set that contains the nodes in *N*, but not in *X*.

(*i, j*) is an arc between node *i* and node *j*.

$\boldsymbol{(X, \overline{X})} = \{(i, j)\text{: } i \in X, j \in \overline{X}\}$, is the set of possible arcs from the nodes in *X* to the nodes in $\overline{X}$. (This is called a *cut-set*.) Note that there exists only a single arc between any pair of nodes *i* and *j*.

DOWN ( ) label: if (i, j) is an arc included in the current estimate of the shortest path, we set DOWN $(j) = i$.

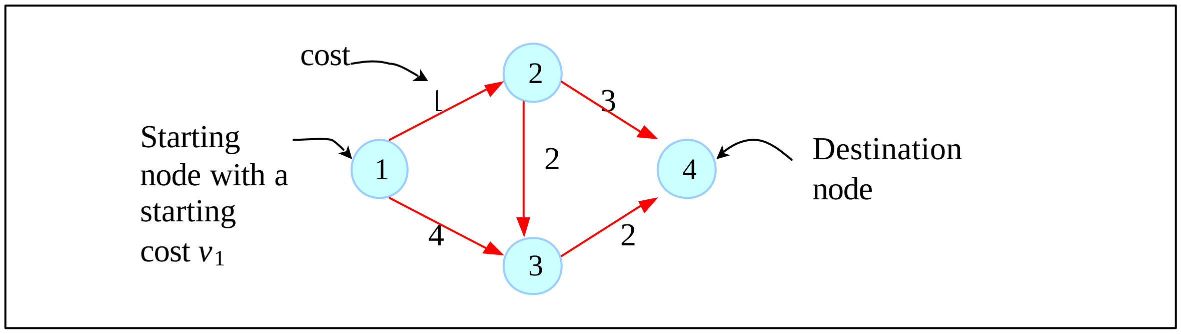


Figure 2: An example of a simple network.

From Figure 2, there are four nodes in the network. Node 1 is the starting node. Node 4 is the destination node. Each arc has a time-independent nonnegative cost. The set of all nodes is N = {1, 2, 3, 4}. There are four possible sets X that contain "node 1" and "any other nodes in the network (nodes 2 and 3) but not node 4". The four possible sets of this type are "X = {1}, X = {1, 2}, X = {1, 3}, and X = {1, 2, 3}". The following table shows the possible sets X, their corresponding complement sets $\bar{X}$, and the cut-sets $(X, \bar{X})$. (Note that Dijkstra's algorithm *does not* require this enumeration, and we only display this here for the sake of illustration.)

Finally, in the (shortest) path $\begin{tikzcd} 1 \arrow[r] & 2 \arrow[r] & 4 \end{tikzcd}$ from node 1 to 4, we have DOWN (4) =2, and DOWN (2) =1.

The possible sets X, their corresponding complements $\bar{X}$ and the cut-sets $(X, \bar{X})$.

X	$\bar{X}$	$(X, \bar{X})$
X = {1}	{2, 3, 4}	{(1, 2), (1, 3)} *Note that arc (1, 4) is not included because a direct-single arc between node 1 and node 4 does not exist in this network.*
X = {1, 2}	{3, 4}	{(1, 3), (2, 3), (2, 4)}
X = {1, 3}	{2, 4}	{(1, 2), (3, 4)}
X = {1, 2, 3}	{4}	{(2, 4), (3, 4)}

INITIALIZATION STEP

1. Let the set N contain all nodes in the network.
2. Set a starting cost v_O for the origin node (node O) as desired. Note that generally, $v_O = 0$.
3. Let X initially contain only the origin node, $X = \{O\}$.

MAIN STEP

1. Let $\bar{X} = N - X$, the set that contains nodes in N but not in X.
2. Find the arcs in the cut-set $(X, \bar{X}) = \{(i, j): i \in X, j \in \bar{X}\}$, the set of possible arcs that connect between nodes in X and nodes in $\bar{X}$.
3. Select the arc (i, j) in the cut-set $(X, \bar{X})$ that has a minimum starting cost plus the cost associated with the arc, $(v_i + d_{ij})$, and let (p, q) be such an arc (i, j). Hence, $v_p + d_{pq} = \underset{(i,j)\in(X,\bar{X})}{\text{minimum}} \{v_i + d_{ij}\}$.
4. Set $v_q = v_p + d_{pq}$ and let DOWN $(q) = p$.
5. If q equals the destination node D, then stop; the shortest path to node D is of total cost v_D, and can be traced by following the DOWN $(\cdot)$ labels backwards. Else, place node q in X and repeat the Main Step.

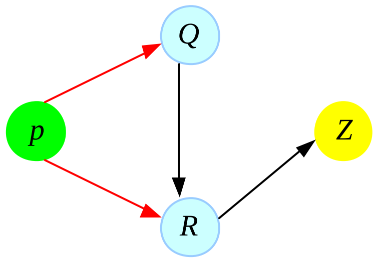
2.1.2 Shortest Path Algorithm for a Network having Mixed Sign Costs

Dijkstra's algorithm described earlier does not generalize to the case when the network includes negative cost arcs. In a more efficient actual implementation in general, that can be adopted even for the case of nonnegative costs, we introduce the concept of a *scan-eligible* set SE, which is the cornerstone of label-correcting algorithms.

Mathematical Terminology and Definition

SE is a *scan-eligible* set that contains nodes *adjacent* to the nodes in X. Note that there exists at most a *single arc* between any pair of nodes in X and SE.

Forward star of node *p* is the set of nodes that are adjacent to the node p and for which there exists an arc from p to each of the nodes in this set. For example, as shown in the network below, nodes Q and R belong to the forward star of the node p but the node Z is *not* in the forward star of the node p.



The following is a description of a basic label-correcting algorithm.

INITIALIZATION STEP

1. Set a starting cost v_O for the origin node (node O) as desired. Note that generally, $v_O = 0$.
2. Let SE initially contain only the origin node, $SE = \{O\}$.
3. Label node O with its starting cost v_O, and label all the other nodes as infinity (∞).

MAIN STEP

1. If SE is empty, then stop; the destination node is unreachable from the starting node. Otherwise, pick the node p from SE that has the smallest label v_p (break ties arbitrarily, but in favor of the destination node).

2. If p equals the destination node D, then stop; the shortest path to node D is of total cost v_D, and can be traced by following the DOWN ( ) labels backwards.
3. Else, scan the forward star of p. For each node q in this forward star of p, if $v_p + d_{pq}$ is less than the current label v_q of node q, then reset $v_q = v_p + d_{pq}$, let DOWN (q) = p, and let $SE = SE \cup \{q\}$.
4. Remove node p from SE and repeat the Main Step.

2.2 Label-Constrained Shortest Path Problem

In this type of shortest path problems, edges/arcs/links in the network are not only weighted but also labeled. The problem then becomes one of finding a shortest path in the network complying with the additional constraint that the unique *word/string of labels* given by concatenating the labels in the network along the path is an **admissible** *word/string*. Barrett, Jacob, and Marathe [1998] introduce this class of problems in the context of formal language constrained path problems, where certain patterns of traversing edge or vertex labels in the labeled graph are permitted, while others are disallowed. Thus, the feasibility of a path is determined by its connection pattern as well as its associated *label sequence*. The acceptable label patterns can be specified as a *formal language.* For illustration, in transportation systems with *travel mode options* for a traveler to go from an origin to a destination, the sequence of permitted *travel modes* constitutes the admissible sequence of labels, and this can be specified by a *formal language.* Such the problems of finding label constrained paths arise in other application areas as well, such as in production distribution networks and database queries.

The following example illustrates a prototypical problem arising in this context. We are given a directed labeled, weighted graph G. The graph here represents a *transportation* network with the labels on edges/arcs representing various modal attributes, e.g. a label c might represent a *car* mode. Suppose, we wish to find a shortest route from O to D for a traveler. This is the practical shortest path problem we regularly find in every day life. But now, we are also told that the traveler wants to go from O to D using the following *modal choices*: either he goes all the way from O to D in his car, or would like to drive his car to a transit station, and then take the transit transportation to D. Furthermore, these different modal choices are linked to each other by a *walk* mode, including the start from the origin and the final link toward the destination. Using w to represent walking, c to represent car, and t to represent transit transportation, the travelers mode choice can be specified as *wcw* or *wcwtw*.

2.2.1 Problem Formulation

Let $G(V, E)$ be a directed graph comprised of a set of vertices, V and edges, E. Each edge/arc $e \in E$ has two attributes: $l(e)$ and $w(e)$. The attribute $l(e)$ denotes the label of edge e. In this context, the label is drawn from an fixed/finite alphabet $\mathbf{S}$. The attribute $w(e)$ denotes the weight of an edge. Here, we assume that the weights are **nonnegative** values. (Note that the next scheme, Section 2.3, will consider the case when the weights are time-dependent functions.) A path p having k edges from u to v in G is a sequence of edges $(e_1, e_2, \ldots, e_k)$, such that $e_1 = (u, v_1)$, $e_k = (v_{k-1}, v)$ and $e_i = (v_{i-1}, v_i)$ for $1 < i < k$. Given a path $p = (e_1, e_2, \ldots, e_k)$, the weight of the path is given by $\sum_{1\le i\le k} w(e_i)$ and the label of p is defined as $l(e_1)\cdot l(e_2)\cdot\cdot\cdot l(e_k)$. In other words the label of a path is obtained by concatenating the labels of the edges on the path in their natural order. Let $l(p)$ and $w(p)$ denote the label and the weight of the path p respectively. We wish to find a shortest path p in G such that $l(p) \in L$, where L denotes a formal language.

In general we consider the input for this problem to consist of a description of the graph, including labels and weights, together with the description of the formal language as a *grammar*. By restricting the topology of the graph and/or syntactic structure of the grammar, Barrett et al. [1998] derive various modifications of the problems.

Note that in *unlabeled* networks with nonnegative edge weights, a shortest path between O and D is necessarily *simple*. This need *not* be true in the case of *label-constrained shortest paths*. As a simple example, consider the graph $G(V, E)$ that is a *simple* cycle on two nodes x and y. Let each edge has weight 1 and label a. The shortest path from x to y consists of a single edge between them; in contrast a shortest path with label *aaa* consists of a cycle from x back to x plus an *additional* edge (x, y).

2.2.2 Algorithm

The algorithm proposed by Barrett et al. [1998] applies to any regular label-constrained shortest path problem. The basic idea behind finding the shortest paths is to construct an auxiliary graph (the product graph, G^*) by combining the **nondeterministic finite automata (NFA)** denoting the regular expression and the underlying graph. The following are notation and terminology pertaining to this algorithm.

Definition 1. A nondeterministic finite automata (NFA), M is a five tuple

(S, Σ, d, s_0, F), where

1. $G(V, E)$ is the directed graph comprised of a set of vertices, V and edges, E,
2. Σ is the input alphabet (a finite nonempty set of letters),
3. Σ^* is a collection of all possible finite strings of alphabets,
4. e is an edge/arc, each edge/arc $e \in E$ has two attributes: $l(e)$ and $w(e)$,
5. $l(e)$ denotes the label of edge e (in this context, the label is drawn from an fixed/finite alphabet $\mathbf{S}$),
6. $w(e)$ denotes the weight of an edge,
7. L denotes a formal language, which is a collection of "words" or strings from Σ^* that are acceptable according to some criteria or rules,
8. Q is a set of finite states for the system,
9. S is a finite nonempty set of states,

10. $d : Q \times \Sigma \rightarrow Q$ is the state transition function that takes a (state, alphabet) combination, and accordingly, transforms to some other (perhaps the same) state,
11. $s_0 \in S$ is the initial state, and
12. $F \subseteq S$ is the set of accepting states.

Definition 2. Given a labeled directed graph G, an origin O and a destination D, define the NFA $M(G) = (S, \Sigma, d, s_0, F)$ as follows:

1. $S = V$, $s_0 = O$, $F = \{D\}$,
2. Σ is the set of all labels that are used to label the edges in G, and
3. $j \in d(i, a)$ if and only if there is an edge (i, j) with label a.

Note that this definition can as well be used to interpret an NFA as a *label*-graph.

Definition 3. Let $M_1 = (S_1, \Sigma, d_1, p_0, F_1)$, and $M_2 = (S_2, \Sigma, d_2, q_0, F_2)$, be two NFAs. The *product* NFA is defined as $M_1 \times M_2 = (S_1 \times S_2, \Sigma, d, (p_0, q_0), F_1 \times F_2)$, where $\forall a \in \Sigma$, $(p_2, q_2) \in d((p_1, q_2), a)$ if and only if $p_2 \in d_1(p_1, a)$ and $q_2 \in d_2(q_1, a)$.

The algorithm can be described as follows:

Input: A regular expression R, a directed-labeled-weighted graph G, an origin O, and a destination D.

1. Construct an NFA $M(R) = (S, \Sigma, d, s_0, F)$ from R.
2. Construct the NFA $M(G)$ of G.
3. Construct $M(G) \times M(R)$. The length of the edges in the product graph are equal to the corresponding edges in G.

4. Starting from the state (s_0, O), find shortest paths to all of the vertices (f, D), where $f \in F$. Denote these paths by p_i, $1 \le i \le w$. Also denote the cost of p_i by $w(p_i)$.
5. Let $C^* := \min_{p_i} w(p_i) = w(p^*)$, say.

   (If p^* is not uniquely determined, choose an arbitrary one.)

Output: The path p^* in G from O to D of minimum length subject to the constraint that $l(p) \in L(R)$.

Barrett et al. [1998] show that the algorithm has a polynomial-time complexity. The basic techniques extend quite easily (with appropriate time performance bounds) to solve other regular expression constrained variants of shortest path problems. Two notable examples that frequently arise in transportation science and can be solved are multiple cost shortest paths and time-dependent shortest paths, which will be described in more detail in the next section.

2.3 Time-Dependent Shortest Path Problem

In this type of shortest path problems, the delays/weights/costs of the links/arcs/edges in the network possibly change with time according to arbitrary functions. Bellman [1957] was the first to define a principle of optimality for dynamic programming as applied to solve such time-independent shortest path problems. Then, Cook and Halsey [1966] extended Bellman's work to refine the application of dynamic programming. Specifically, let $d_{ij}(t)$ denote the time-dependent delay function on link (i, j) between nodes i and j at time t. We wish to find a shortest path from node O to node D starting at a time $t = t_0$. Cook and Halsey let $d_{ij}(t)$ be any positive integer function, and considered a discrete time set $S = \{t_0, t_0+1, t_0+2, \ldots, t_0+T\}$, where T is an upper bound determined via some path from O to D. They defined $E_i^{(k)}(t)$ for $t \in S$, to be the set of all paths having at most k links leaving i at time t and reaching D at or before $t_0 + T$, and iteratively determined $f_i^{(k)}(t)$ as the shortest path value from node i to node D, starting at time t, and involving at most k links.

Dreyfus [1969] was the first who suggested to solve the time-dependent shortest path problems using Dijkstra's labeling algorithm. Instead of using discrete time intervals in the traditional Dijkstra's algorithm, he suggested that Dijkstra's algorithm could be directly applied to an expanded static time-space network in which the link travel times are effectively time-invariant.

Sherali, Ozbay, and Subramanian [1998] proved that the time-dependent shortest path problem, and many variations of it, are NP-hard (even if only *one* link in a network has a time-dependent delay). They also studied time-dependent shortest pair of path problems, developing strong 0-1 linear programming models to solve this problem. The model can accommodate various degrees of disjointedness of a pair of paths, from complete to partial with respect to specific links.

Most of the algorithms proposed for the solution of the shortest path problem with time-dependent delays are valid only under the assumption that parking/waiting at the nodes is *un*limited and any desirable delay in departure time from a given node is permitted. Halpern [1977] considered the case where such as assumption is *not* acceptable, and presented an efficient algorithm for the solution of the time-dependent and *parking regulation* shortest path problem. Various situations may be described by this class of problems:

- Fluctuations in the time of travel between two intersections or regions. For example, streets that are opened for traffic in *one way* only, say from a suburb to downtown during morning rush hours, and in the opposite direction in the late afternoon.

- Parking regulations. Parking may be *prohibited* during certain times of the day or restricted to a *limited* amount of time.

Halpern considered the case where parking or waiting at the nodes is *limited* by a specified extent, so that any desirable delay in departure time from a given node is *not permitted.* He presented an efficient algorithm for the solution of the shortest path problem in networks having time-dependent delays on the edges and *parking regulations* at the nodes. Orda and Rom [1990] studied the time-dependent shortest path problem with FIFO and non-FIFO links.

Similarly to Halpern, Orda and Rom [1990] also proposed algorithms for various waiting constraints. Their motivation was that Halpern's algorithm *cannot* be bounded by the network topology (i.e., the number of operations cannot be bounded as a function of the number of nodes or edges), nor are the properties of the resulting path investigated, for example, whether it is a simple path (a path having non-repeating nodes). Unlike Halpern's algorithm that avoided the treatment of functions by addressing the problem for a *single* instant of time, and not for ranges of time, Orda and Rom presented algorithms for finding time-dependent shortest paths for *all* instances of time, and investigated properties of the derived paths. Moreover, they did not restrict their study to **FIFO** links only. For example, the FIFO assumption is invalid for a traveler standing on a platform at a railway station wondering whether to take the first regular train stopping in front of him *or* to wait for the express one. Because of the possible non-FIFO characteristics of such a situation, a traveler might prefer to *wait* for a certain amount of time before embarking on to further links. However, all nodes may not permit such waiting. Orda and Rom considered three different types of waiting constraints:

1) *Unrestricted Waiting*. Unlimited waiting is allowed for every node in a network.
2) *Forbidden Waiting.* Waiting is *dis*allowed for every node in the network.

3) *Source Waiting.* Unlimited waiting is allowed *only at the source node*; waiting is disallowed for every node *except* the origin.

It is interesting to note that for the Forbidden Waiting case, it is possible that the shortest path may neither be simple nor concatenate. An example is shown in Figure 3.

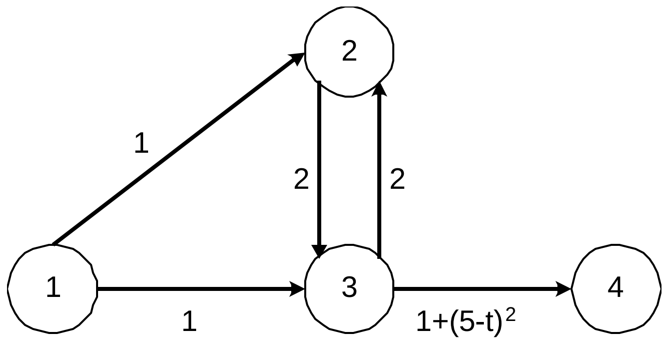


Figure 3: Nonsimple, nonconcatenated, and non-FIFO time-dependent network

Note that the travel delay on link (3, 4) is non-FIFO and monotonically decreases for $t_3 \leq 5$. Here, the shortest path from node 1 to node 4 is {1, 3, 2, 3, 4} with a total delay of 6. It is the only shortest path solution and contains a loop (it is non-simple). The shortest path repeats node 3, while the shortest path from node 1 to node 2 is {1, 2}, and not {1, 3, 2}. Thus, the shortest path from node 1 to node 4 is neither simple nor concatenated, in the foregoing respective cases.

The above example suggests that if the shortest paths are not concatenated, then *partial results* cannot be used. The determination of a shortest path between two nodes must consider *all* possible paths between them. This problem can be shown to be NP-hard. Moreover, it is possible to have infinite links in a shortest path via infinite loops, and yet have finite delay (i.e. the finite series of accumulated delays has a finite value). Orda and

Rom concluded that Halpern's algorithm cannot perform its task in all case unless the restrictions on delay function are tightened (using FIFO and positive delay functions).

Orda and Rom also stated two types of Source Waiting Constraints, given respectively by a single starting time and multiple starting times. They proved that for the case of a *single starting* time with continuous (and/or piecewise continuous) delay functions and negative discontinuities, the shortest path of the Unrestricted Waiting Constraint case is equivalent to the Source Waiting Constraint case. Hence, the procedure used to obtain the equivalent Source Waiting Constrained path is to first find the shortest path using the Unrestricted Waiting case, and then, starting from the *destination* node, calculate the time of departure from the previous node and so on, back to the origin node.

For the case of multiple starting times, Orda and Rom again first use the Unrestricted Waiting Constraint case to find a shortest path. Next, they compute the source waiting time for any appropriate starting time. Unlike the previous algorithm, this algorithm is not guaranteed to work for piecewise continuous functions. Orda and Rom introduced a relaxation of the Source Waiting problem to allow limited waiting (in addition to the unrestricted waiting at the source) at nodes whose incoming links have discontinuous delay functions.

Generally, their work shows that the time-dependent shortest path problem can be solved efficiently when no waiting constraints are imposed at the nodes. However, the computational complexity for the general time-dependent shortest path problem (where delays may be non-FIFO) is NP-Hard, *almost* surely cannot be solved in *polynomial-time.* Furthermore, Sherali et al. [1998] show that this NP-Hard result holds true, even there is *only* one link in the network that has a time-dependent delay.

2.4 Description of TRANSIMS

TRANSIMS (Transportation Analysis and Simulation System) is an integrated system of travel forecasting models designed to provide transportation planners with accurate, complete information on traffic impacts, congestion, and pollution. It is one part of the multi-track Travel Model Improvement Program sponsored by the U.S. Department of Transportation, the Environmental Protection Agency, and the Department of Energy. The Los Alamos National Laboratory is developing new, integrated transportation and air quality forecasting procedures necessary to satisfy the Intermodal Surface Transportation Efficiency Act, and the Clean Air Act, and its amendments. This organization released the first version of the TRANSIMS software (C++ on Linux base) in May 1999 (see Los Alamos National Laboratory [1999]). Recently, they have released the latest version in July 2000 (see Los Alamos National Laboratory [2000]).

TRANSIMS departs from the traditional four-step process that is commonly used in transportation planning for demand forecasting and impact analysis. The new technical approaches in TRANSIMS respond to issues derived from legislation such as the Intermodal Surface Transportation Efficiency Act and the Clean Air Act. The various transportation issues that require new technical approaches include (1) congestion pricing, (2) alternative development patterns, (3) transportation control measures, and (4) motor vehicle emissions. In addition, major initiatives such as the Intelligent Transportation System program require new analytical approaches for conducting substantive evaluations of their effectiveness. A *major* TRANSIMS technical feature is that the identity of *individual* synthetic travelers is maintained throughout the entire simulation (over a 24-hour day) and the architecture of the analysis.

2.4.1 Major Data Inputs

TRANSIMS requires the following major inputs:

- Census Data. This is used as a source to create a base of synthetic population,
- Traveler Activity Survey. This is used along with the census data to generate activities for each individual population in the synthetic households
- Land-Use Data. This is used to select an appropriate zone for each of the synthetic household member's activities.
- Transportation Network. This is a very important input used to find specific activity locations within a zone for each activity, to construct a network for solving the shortest path for each trip, and to run a traffic simulation. It provides information regarding streets, intersections, signals, parking, activity locations, and transit modes.

2.4.2 TRANSIMS' Modules

Currently, there are six different modules in TRANSIMS. Figure 1 details the inputs and outputs of these modules, along with their interactions with the other relevant modules. The six modules are generally described as follows (for more details, see http://transims.tsasa.lanl.gov).

1) **Population Synthesizer Module.** This module generates synthetic households, which represents every *individual* in the metropolitan region under study from the census data at the block group level or the census tract level. It develops the associated demographic characteristics (e.g., age, gender, income, etc.) for each synthetic household. Each synthetic household is located on a link in the transportation network via the land-use data. The assignment of vehicles to each

household, including information regarding the vehicle emission type and the initial vehicle location are also generated in this module.

2) **Activity Generator Module.** This module takes as a major input the households in the synthetic population, local area surveys, transportation networks, and land-use data. The Activity Generator Module *produces a list of activities for each traveler* in the system. Activity patterns and mode choice preferences are derived from surveys. This derivation depends on demographic information contained in the synthetic households. The Classification and Regression Tree (CART) method is used in this module to produce an accurate classification of households based on an assumption that households having the same demographic characteristics should have the same travel/activity behavior. The locations of the activities for each traveler are currently chosen by using a method derived from gravity models.

3) **Route Planner Module.** This module attempts to produce plans for every individual traveler in the activity lists. The Route Planner Module computes a shortest or least-cost path for each traveler. The methods used range from Dijkstra's algorithm to sophisticated time-dependent label-constrained shortest path methods (Nagal et al. [1998]). If mode preferences are recorded for the traveler, the Route Planner Module ensures that these are met and that the plan contains the required modal links. The Route Planner estimates the time that it takes to make a trip based on link traversal time estimates contained in the transportation networks or in the simulation output.

4) **Traffic Microsimulator Module.** This module simulates the travel of individual vehicles and travelers in accordance with the plans provided by the Route Planner Module. Each plan has a specified start-time, which begins the execution of movement for that traveler. Plans that overlap in time are executed *simultaneously* by the Traffic Microsimulator Module. The interactions of travelers and vehicles on the network over time create traffic flow dynamics.

5) **Emissions Estimator Module.** This module uses results from the Traffic Microsimulator Module to predict tailpipe emissions for light- and heavy-duty vehicles. Any pollution accruing from spilled fuel evaporation is also estimated. These emissions are aggregated to provide input to other systems that are specifically designed to produce overall regional air quality estimates.

6) **Selector Module.** This module is the primary mechanism used to achieve internal consistency (i.e., to achieve a reasonable agreement among the travel demands expressed in the activities list, the travel plans to meet these demands, and the execution of the plans in the simulation), among the various computational modules. It selectively feeds back information from one module to another. In effect, this information is used to modify some designated subset of activities and/or plans to achieve *realistic* overall traffic results.

Nagel et al. [1999] indicate some disadvantages of TRANSIMS, including the following. (i) *Size of the problem*: Metropolitan regions typically consist of several millions of travelers. Executing a second-by-second traffic microsimulation on a problem of this size within reasonable computing time is only possible with the use of advanced statistical and computational techniques. (ii) *Behavioral foundation*: We are far from understanding human behavior. For this reason alone, we are unable to predict the accurate behavior of any *individual* traveler. However, there seems to be a realistic chance that the *macroscopic* behavior that is generated by thousands or tens of thousands of interacting individuals is considerably more robust than the behavior of an *individual* agent. This would be similar to Statistical Physics, where the trajectory of a single particle is unpredictable, yet, useful *macroscopic* properties of gases such as equations of state can still be derived. (iii) *Consistency problem*: The approach outlined above is not as straightforward as it sounds because the plans depend on *expectations* about traffic conditions during execution. For example, if a person expects congestion, he or she may make *different* plans than if no congestion is expected. Yet, congestion occurs only when plans *interact* during their simultaneous execution. In short, plans depend on congestion, but congestion also depends on plans. This logical deadlock is not unknown in economic theory and is traditionally overcome by the assumption of rational agents. Both with and without such an assumption of rationality, this problem of consistency between plans and microsimulation makes the computational challenge even bigger. (iv) *Robustness*: Any approach to a problem needs to have reproducibility of the results under a wide range of changes, or otherwise the results are useless for practical purposes.

In conclusion, although it is now possible to run TRANSIMS, the agent-based simulation approach to transportation on enormously large problems, the computing aspects are still challenging and demand a knowledgeable use of the available technology.