# Link State Relationships under Incident Conditions: Using a CTM-based Dynamic Traffic Assignment Model

**Weihao Yin**

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

**Master of Science**

In

Civil and Environmental Engineering

Pamela Murray-Tuite, Chair
Kathleen Hancock, Member
Konstantinos Triantis, Member

August 11, 2010

Falls Church, VA

**Keyword:** Transportation Engineering, Link State relationship, Dynamic Traffic Assignment, Cell Transmission Model

# Link State Relationships under Incident Conditions:

# Using a CTM-based Dynamic Traffic Assignment Model

**Weihao Yin**

## Abstract

Urban transportation networks are vulnerable to various incidents. In order to combat the negative effects due to incident-related congestion, various mitigation strategies have been proposed and implemented. The effectiveness of these congestion mitigation strategies for incident conditions largely depends on the accuracy of information regarding network conditions. Therefore, an efficient and accurate procedure to determine the link states, reflected by flows and density over time, is essential to incident management.

This thesis presents a user equilibrium Dynamic Traffic Assignment (DTA) model that incorporates the Cell Transmission Model (CTM) to evaluate the temporal variation of flow and density over links, which reflect the link states of a transportation network. Encapsulation of the CTM equips the model with the capability of accepting inputs of incidents like duration and capacity reduction. Moreover, the proposed model is capable of handling multiple origin-destination (OD) pairs. By using this model, the temporal variation of flows over links can be readily evaluated.

The visualized prediction of link density variations is used to investigate the link state relationships. By isolating the effects of an incident, the parallel routes of a specific OD pair display the relationship of substituting for each other, which is consistent with the general expectation regarding such parallel routes. A closer examination of the density variations confirms the existence of a substitution relationship between the unshared links of the two parallel routes. This information regarding link state relationship can be used as general guidance for incident management purposes.

## Acknowledgement

I would like to express my sincere gratitude and thanks to my advisor and chair of my committee, Dr. Pamela Murray-Tuite, for her guidance and constant support during my study at Virginia Tech. I would also like to thank my committee members, Dr. Konstantinos Triantis and Dr. Kathleen Hancock for their valuable comments and help.

Very special thanks are to my colleagues and friends Zhuojin Wang and Sirui Liu, who provided a lot of assistance when I started my master studies. I would like to express my deep appreciation to my family for their unconditional support and long time encouragement throughout my studies.

Table of Contents

## List of Tables

## List of Figures

# Chapter 1 Introduction

Urban transportation networks are vulnerable to various events ranging from natural disasters, such as hurricanes, to common traffic incidents, like accidents. A typical consequence of these events is congestion which causes high travel time variability. Congestion due to incidents constitutes at least 25% of total congestion (Cambridge Systematics, 2005). In addition, incidents account for approximately 60% of the vehicle hours lost to congestion (Robinson and Nowak, 1993). Congestion effects introduced by incidents are accentuated during peak hours when traffic flow approaches road capacity; the queue caused by lane closure possibly remains until the peak hours end (Helman, 2004).

In order to combat the negative effects due to congestion, various mitigation strategies have been proposed and implemented in the United States and worldwide. Strategies commonly used include ramp metering as well as hard shoulder operation which aim to deal with recurrent congestion. For non-recurrent congestion mainly due to incidents, mitigation strategies adopted are variable speed limits, dynamic high-occupancy-vehicle (HOV) designations, and route diversion through variable message signs (Liu and Murray-Tuite, 2008). The effectiveness of implementing these strategies for incident conditions largely depends on the accuracy of information regarding network conditions. Particularly for the route diversion strategy, it is important to predict the amount of traffic diverted in order to ensure the effectiveness of the strategy; operations of the diverted routes may need to be adjusted accordingly to handle additional traffic due to the diversion. Therefore, an efficient and accurate procedure to determine the link states is essential to incident management. Moreover, understanding of the link state relationships can also be useful to provide general guidance for incident mitigation efforts.

The remainder of this chapter is divided into four sections. The first section presents background to the study methodology by discussing the nature of link states and dynamic traffic assignment. The objectives of the study are presented in the second section, and then the major contributions are discussed in the third section. The last section outlines the organization of the thesis.

## 1.1. Methodology of the Study

The necessity of using Dynamic Traffic Assignment (DTA) is illustrated by a discussion of the nature of link states, which focuses on comparing and contrasting the difference between links of

transportation network and other networks. Then, the characteristics of DTA output are discussed to show the appropriateness of the methodology.

### 1.1.1. Nature of Link States

The states of the links that constitute a traffic network cannot be described exclusively by binary indicators for detailed analyses. Such dichotomous descriptions like "connected" and "disconnected" are suitable for utility networks because the connection status of links are of most interest under usual circumstances. However, under most occasions, links of a transportation network exhibit intermediate states rather than extreme states such as "disconnected." Furthermore, the link states are dynamic. Therefore, continuous measurements that have a time dimension are needed to accurately describe the link states of a traffic network. Fortunately, the traffic conditions on a specific link can be described using measures from well-established traffic theory such as flow, density and speed, which are dynamic and continuous.

In addition to the difficulty of state description, the human factor adds to the complexity of determining the link states. When incidents occur, drivers tend to choose a route different from the one that is usually chosen under normal conditions especially when information about the traffic conditions is available. This is very different from traditional communication networks in which signal traffic, under normal conditions, would not divert from the pre-determined route spontaneously. However, people spontaneously switch routes under similar conditions within a transportation network. In other words, the route choice of network traffic possesses a dynamic nature. Therefore, in order to predict the link states, this dynamic routing behavior needs to be captured.

The existence of intermediate link states, combined with the dynamic nature of drivers' route choice and multiple origin-destination pairs, leads to complicated link state relationships. Specifically, link state relationships may change over time and cannot be excessively synthesized into simple descriptions. Therefore, a complete evaluation of the dynamic variations of link states is necessary to facilitate the understanding of the link state relationships.

### 1.1.2. Dynamic Traffic Assignment

The static traffic assignment problem is defined as determining the flows for each link of a transportation network based on known demand (origin-destination matrix) and link performance functions (Sheffi, 1985). Dynamic Traffic Assignment (DTA) departs from this definition by dealing with time-dependent flows. In addition, DTA is inherently characterized by the need to adequately represent traffic realism and human behaviors, which are, to some extent, reflected by the assignment principles like user-equilibrium (UE). The UE principle assumes that network users, or drivers, can choose their paths freely at any time. Therefore, the output of a DTA procedure based on UE principle provides all the necessary elements to describe link states and subsequently understand link state relationships. User-Equilibrium Dynamic Traffic Assignment (UE-DTA) is applied for this study since it gives both the required flow measurements and captures the dynamic routing behavior.

### 1.1.3. Link State Relationships

Due to the dynamic nature of the link states as discussed in the previous paragraphs, the link state relationships also possesses this dynamic feature. As a result, link state relationships can only be revealed through constant monitoring of link states over time. In this study, the link state relationships are examined by using link state maps, which document traffic density variations over time. By comparing the link state maps under different conditions, the link state relationships can be unraveled. The details of constructing the maps and comparison are provided in Chapter 5.

## 1.2. Objectives of the Study

The objectives of this study can be described as follows:

- Construct a model that is capable of providing the temporal variation of flows and densities over links within a traffic network;
- Transform the predicted flows into temporal density variations to evaluate link states of a traffic network and
- Compare density variations under different scenarios to identify link state relationships for the traffic network.

## 1.3. Contributions of the Thesis

This thesis constructs a new DTA model to predict link states and subsequently derive insights about link state relationships. The new model is able to handle networks with multiple origin-

destination pairs and model incidents. These are major improvements to the original framework by Carey (1999; 2009). Based on the visualized link states, link state relationship can be revealed. In addition, the model, with proper adaption, can serve as a useful component for incident mitigation and management in near real time.

## 1.4. Organization of the Thesis

The rest of the thesis is organized as follows. The next chapter reviews research efforts in the DTA field and focuses on various analytical formulations. Chapter 3 lays out the formulation of the model, discusses some properties of the proposed model and presents the proof of equivalence to dynamic user equilibrium. Numerical examples are provided in Chapter 4. Based on the results for the two sample networks, basic insights into link state relationships are provided in Chapter 5. Conclusions and future directions are provided in the last chapter.

# Chapter 2 Literature Review

Since dynamic traffic assignment is applied to identify the link states, only literature relevant to DTA is reviewed here. To the author's best knowledge, no studies have addressed link state relationships under dynamic incident conditions. Existing studies only target the link relationships for path routing algorithms (Bhosle and Gonzalez, 2003) and investigate link relationships under static traffic assignment using sensitivity of link capacities (Jiang Qian and Miyagi, 2001).

## 2.1. Introduction

DTA has received a lot of attention due to its significance in predicting traffic patterns within a transportation network for controlling and managing the network. According to different assignment principles, DTA problems fall into two general categories, namely, system optimum and user equilibrium. If the time-dependent origin-destination matrices are assumed to be known, in Dynamic User Equilibrium (DUE), users choose paths whose travel costs (time) are no higher than those on other available paths. The complete definition of DUE implies users cannot shorten their travel time by unilaterally changing paths, which is similar to the definition of static user equilibrium presented by Sheffi (1985). A more formal definition of DUE is given by Janson (1991a; 1991b) who called DUE "a temporal generalization of static user equilibrium assignment with additional constraints to insure temporal continuous trip paths". The two conditions presented in his work are as follows:

- "All paths between a given zone pair used by trips arriving at the destination within a given time interval must have equal travel impedance and
- All paths between a given zone pair not used by trips arriving at the destination within a given time interval cannot have lower travel impedance."

The system optimum principle distinguishes itself from user equilibrium by requiring users to make route decisions for the sake of the network-wide benefit, or more specifically, minimization of the total travel costs of all the users.

There are numerous research papers regarding the DTA problem. Solution methods for DTA can be synthesized into two major approaches: computer simulation and analytical models. Based on the mathematical techniques applied, analytical models can be classified into three categories: mathematical programming, optimal control theory, and variational inequality (Peeta and

Ziliaskopoulos, 2001). The analytical formulations, developed in the past two decades, focus on user equilibrium (UE) and system optimum (SO), or variants of them such as incorporation of the stochastic nature of travel time.

It should be noted that real-time deployment of DTA models and realistic operational performance are among the objectives of DTA models. Regardless of the mathematical techniques the analytical models apply, conflict exists between tractability and modeling details, especially when traffic flow behaviors are considered within links of a traffic network. It is relatively difficult to incorporate traffic behaviors into mathematical programs or other analytical frameworks compared to discrete simulation. Aiming to overcome this difficulty, numerous research efforts have been devoted to incorporating traffic flow models into DTA solving mechanisms. A natural idea is to absorb the traffic flow model into the network assignment procedure. Some models are hard to solve due to non-linearity of the travel time function or travel time calculation procedures they adopt. These DTA formulations, incorporating various traffic flow models, will be discussed in the next section in greater detail.

In addition to the requirement of realistic representation of traffic dynamics, it is important to model various external disruptions, such as traffic incidents, within the DTA modeling frameworks. With the availability of information about the incident occurrences, travelers have the option to change their routes accordingly. This dynamic routing behavior may significantly influence the traffic pattern of a certain network especially within the DTA context. Hence, the capability of modeling transient incidents should be considered critical for DTA models.

The remainder of the chapter reviews the analytical DTA models first and then the simulation-based DTA models. The last section compares the two methodologies and distinguishes the proposed model to its analytical counterparts.

## 2.2. Review of Analytical DTA models

Analytical DTA models mainly apply three categories of mathematical techniques, namely mathematical programming, optimal control theory and variational inequality. The review follows this order.

### 2.2.1. Mathematical programming formulations

The applications of mathematical programming techniques to DTA were pioneered by Merchant and Nemhauser (1978a; 1978b). Their formulation (M-N model) deals with a simple deterministic problem of a fixed demand, single-destination network within the system optimum context. A typical problem with this SO formulation as well as many other SO models is First-In-First-Out (FIFO) violation. FIFO conditions make a lot of formulations unsolvable analytically due to the fact that it invites non-convexity to mathematical programs (Peeta and Ziliaskopoulos, 2001). Compared to analytical models, FIFO compliance is not a problem for simulation models since it is easy to track each vehicle and thus maintain the FIFO condition. Another problem associated with some SO formulations like the M-N model relates to "holding-back" vehicles on links. In other words, certain traffic streams are intentionally favored over others to minimize the system delays (Carey and Subrahmanian, 2000).

Studies by Janson (1991a; 1991b) are the first formulations attempting to model DTA under the UE principle. Non-linear mixed integer constraints were applied for the purpose of ensuring temporal continuity of OD flows. It is noted by other authors such as Peeta and Ziliaskopoulos (2001), that this model led to unrealistic traffic behaviors and relied on static link travel time functions. In order to realistically capture traffic behaviors, several authors attempted to accommodate traffic stream models into mathematical programming DTA models. Bi-level mathematical programs that encapsulate Greenshields' traffic flow model (Jayakrishnan, Tsai et al., 1995; Jayakrishnan, Chen et al., 1999) represent those early attempts.

As noted by Peeta and Ziliaskopoulos (2001), mathematical programming approaches have their limitations in strict adherence to dynamic optimality conditions and retaining the FIFO property as well as realistic traffic dynamics for general networks.

### 2.2.2. Optimal control formulations

In an optimal control modeling framework, both OD demands and traffic flows are considered as continuous functions of time, different from mathematical programming formulations. Friesz, Luque et al. (1989) proposed a link-based optimal control formulation for both SO and UE objectives for the single-destination case. The central assumption was continuous modifications to routing decisions based on changing network conditions. They also generalized Beckmann's

equivalent optimization problem for static UE traffic assignment as an optimal control problem, which lacks an efficient solution algorithm. By defining link inflows and outflows as control variables, Ran and Boyce (1994) transformed the DUE problem into a convex model using the Frank-Wolfe algorithm, which is easy to solve relative to the non-convex model.

The limitations of optimal control formulations lie in their lack of explicit constraints to preclude FIFO violations and sound procedures to maintain traffic realism and more importantly a solution procedure for general networks (Peeta and Ziliaskopoulos, 2001).

### 2.2.3. Variational inequality formulations

The strengths of variational inequality (VI) derive from its unified mechanism to address equilibrium and equivalent optimization problems (Peeta and Ziliaskopoulos, 2001). Moreover it can handle more realistic traffic scenarios. The study by Dafermos (1980) serves as the pioneer to use the VI approach for the traffic equilibrium problem. The model presented by Huey-Kuo and Che-Fu (1998) demonstrated the feasibility of the VI approach within the UE-DTA context by relating travel time of a link exclusively with link inflow.

A more influential study (Lo and Szeto, 2002) developed a variational inequality model that uses CTM as the underlying traffic flow model. This model successfully meets the FIFO condition and is capable of capturing traffic dynamics. It can mimic the queue accumulation and dissipation under capacity reduction scenarios, which is substantial progress for DTA models using the VI approach. This formulation successfully circumvented traffic realism issues that were raised towards the VI approach by Peeta and Ziliaskopoulos (2001) and essentially made it possible for the VI approach to achieve both computational tractability and traffic realism.

## 2.3. DTA Models Based on the Cell Transmission Model

The Cell Transmission Model (Daganzo, 1994; Daganzo, 1995) provides a set of linear equations that are a numerical approximation of the Lighthill-Whitham-Roberts model (Lighthill and Whitham, 1955; Richards, 1956), or LWR model. Ever since its inception, the CTM received great attention from researchers who focus on dynamic traffic assignment due to its mathematical simplicity for encapsulation in an analytical framework. Ziliaskopoulos (2000), in a pioneering work, applied the CTM to a dynamic traffic assignment problem. The single-destination system optimum

dynamic traffic assignment problem was formulated as an LP model based on CTM. Since there was only one origin-destination pair and system optimum was in effect, FIFO did not pose a threat to the analytical formulation. Though the model does not have much operational value for actual applications (Peeta and Ziliaskopoulos, 2001), it did provide insights into the DTA problem by raising the concept of marginal travel time and more importantly, explored the possibility of linear formulation of DTA based on the CTM.

Inspired by the aforementioned work, various extensions to the LP-DTA model of Ziliaskopoulos (2000) have been made. The single-destination assumption was relaxed by Li and Ziliaskopoulos (1999). They designated fixed arrival windows for all vehicles of the network and set the objective function to minimize the total travel time experienced for all users within the network. While the two papers by Ziliaskopoulos and his colleagues formulated the system optimum DTA (SO-DTA) problems for single and multiple destinations, another significant contribution to the linear programming model for SO-DTA problem was made by Waller and Ziliaskopoulos (2006b). It presented a stochastic extension to Ziliaskopoulos' deterministic LP model. The paper proposed a chance-constrained based formulation which provided a robust SO solution when the level of reliability is specified by users.

Though Ziliaskopoulos and his colleagues did not propose analytical formulations for user-equilibrium DTA (UE-DTA), they developed heuristic algorithms for UE-DTA problems. Waller and Ziliaskopoulos (2006a) developed a combinatorial algorithm for the single-destination UE-DTA problem based on CTM. The conceptual framework is straightforward. Vehicles are always assigned to the time-dependent shortest paths, which are calculated at the beginning of each iteration. Successful attempts by Golani and Waller (2004) were made to extend the algorithm to the multi-destination UE-DTA problem though vehicles were assumed to take fixed routes.

It can be seen that LP DTA models proposed by Ziliaskopoulos and his colleagues are able to obtain robust solutions for the single-destination system optimum DTA problem. Under certain assumptions, the model can be extended to incorporate multiple origin-destination pairs. Unfortunately, their models are not capable of analytically dealing with the UE-DTA problem though heuristics for UE-DTA are developed.

Aiming to tackle the UE-DTA problem within a linear programming framework, Carey and his colleagues proposed an LP framework for the single-destination UE-DTA problem. Essentially, Carey's DTA framework (Carey, 1999; Carey and Subrahmanian, 2000; Carey, 2009) is a two-module mathematical program in which the first module is formulated as an LP-based DTA problem and the other module is referred to as the link sub-model. The link sub-model predicts traffic flow over each link at each time interval. These link flow predictions are then fed back to the LP program and serve as link capacity at each time interval. By iterating between the two modules, a UE-DTA solution is obtained when convergence is reached.

## 2.4. Review of Simulation-Based Models

Compared to analytical DTA models, simulation-based DTA models rely on a traffic simulator to replicate the complex traffic flow dynamics and additionally, use it to search for the optimal solution (Sisiopiku and Xuping, 2006). Although analytical formulations provide critical insights into DTA problems and applications, as noted by previous works (Peeta and Ziliaskopoulos, 2001; Sisiopiku and Xuping, 2006), limitations such as traffic realism and computational tractability still raise questions regarding their applicability for realistic applications due to the fact that the network size is typically very large. Therefore, when considering realistically large networks, simulation-based DTA models enjoy advantages for practical implementation over analytical approaches. In addition, notions of convergence and uniqueness of the solution may not be considered important from a practical viewpoint (Peeta and Ziliaskopoulos, 2001), so simulation-based models have gained greater acceptability and popularity in the context of real-world problems (Ben-Akiva, Bierlaire et al., 1998).

A clarification needs to be made for the definition of the simulation-based DTA models. The term "simulation-based" primarily concerns the solution methodology rather than the problem formulation which is typically a mathematical programming model (Peeta and Ziliaskopoulos, 2001). In other words, the traffic flow propagation and critical constraints such as flow conservation and vehicular movements are achieved through simulation. As a result, various simulation-based DTA models are distinguished from each other primarily by the traffic simulation procedures.

This section focuses on two major simulation-based DTA models, namely DYNASMART and DynaMIT. The underlying routing logic and traffic propagation mechanism for both models will be

discussed and compared. Various additions proposed by other researchers to these two models are also reviewed. Besides, the DTA modules of two widely-used commercial packages specifically, INTEGRATION and VISSIM will also be discussed.

### 2.4.1. DYNASMART

DYNASMART, short for **DY**namic **N**etwork **A**ssignment **S**imulation **M**odel for **A**dvanced **R**oad **T**elematics, is initially developed specifically for studying the information-supplying and information-control system for urban traffic networks with ATIS (Advanced Traveler Information System) and/or ATMS (Advanced Traffic Management Systems) by Mahmassani et al. (1992; 1994).

This original version of DYNASMART combines three key elements of dynamic traffic simulation in a modular manner. These three modules are: traffic flow module, driver behavior module, and path processing module (Jayakrishnan, McNally et al., 1993b). The traffic flow module is responsible for the generation and movement of individual vehicles and updating link traffic conditions. It moves vehicles according to prevailing local speeds and keeps track of their positions. The speed is determined by using a modified version of the Greenshield's equation for the speed-density relationship (Jayakrishnan, Mahmassani et al., 1994). The driver behavior module performs the functionality of route decision based on bounded rational behavior where drivers are indifferent to route changes which offer minimal travel time benefits. For an information-technology enabled vehicle, route choice is made when it approaches the end of a link. The travel time to the destination on the vehicle's currently assigned path is calculated and compared to the corresponding time on the minimum of the K-shortest paths. If the improvement of travel time by changing paths exceeds a defined threshold, a path change occurs. Vehicles unequipped with traffic information devices stay on the initially assigned path. This simple path choice logic governs the driver behavior within the traffic network (Jayakrishnan, McNally et al., 1993a). Since the path choice logic is based on K-shortest path, the path processing module finds K-shortest paths based on an efficient label-correcting algorithm on updated link and arc travel times from every node to every destination (Jayakrishnan, Mahmassani et al., 1991).

Based on the algorithmic framework aforementioned, DYNASMART is able to model route choice of drivers with and without access to ATIS information and possesses the ability to keep track of

locations of all drivers and is capable of predicting the time-dependent travel time according to assignment results (Jayakrishnan, Mahmassani et al., 1994). DYNASMART is also able to model real-time traffic control strategies such as signal settings and ramp metering.

DYNASMART has two different versions namely DYNASMART-X and DYNASMART-P. DYNASMART-X, developed in 1998 several years after DYNASMART was pioneered (Mahmassani, Hu et al., 1998), is an online traffic estimation and prediction system (Mahmassani, Dong et al., 2009), which predicts traffic conditions over the near future (usually less than an hour). It is used in conjunction with traffic management activities such as dissemination of traffic information via Variable Message Sign (VMS), traffic controlling practice like ramp metering and incident mitigation such as traffic diversion. This online version of DYNASMART interacts continuously with multiple source of real-time information such as loop detectors and vehicle probes and provides (1) estimates of current network traffic conditions; (2) predictions of network flow patterns over the near and medium terms, in response to various contemplated traffic control measures and information dissemination strategies; and (3) anticipatory traveler and routing information to guide trip-makers in their travel activities(Dong, Mahmassani et al., 2006; Sisiopiku and Xuping, 2006; Mahmassani, Dong et al., 2009). The additions made to the original DYNASMART algorithmic framework are Origin-Destination (OD) estimation and prediction modules. The OD estimation module (ODE) is responsible for estimating the coefficients of a time varying polynomial function that describes the OD demand in the current stage. The OD prediction module (ODP) utilizes these to calculate the demand that is generated from each origin to each destination at each departure time interval of the current and future stages(Mahmassani, Hu et al., 1998; Mahmassani, Dong et al., 2009).

On the other hand, DYNASMART-P is intended for offline planning and evaluation applications such as assessing impacts of ITS strategies on the transportation network, work zone planning, evaluation of HOV and HOT lanes and evaluation of congestion pricing schemes, etc (Mctrans, 2010). The difference between DYNSMART-P and original DYNASMART lies in the functionality that the planning version possesses of simulating activity/trip chains while the basic theory and core capabilities remain unchanged (FHWA, 2007). Specifically, vehicles are permitted to exit the transportation network at intermediate destination(s) along their travel path to perform a particular activity for a time that is equal to the activity duration. Vehicles, outside the traffic network for

activities, have no impact over the network. Upon completion of an activity, the trip-maker resumes their trip again from this destination, to complete the trip according to their pre-specified travel pattern. Once the vehicle reaches its final destination, it exits the network. This trip/activity chain logic distinguish the planning version from the online version as well as the original DYNASMART (FHWA, 2007; Mahmassani, Dong et al., 2009).

The limitations that DYNSMART-P has are that it cannot model detailed traffic maneuvers such as car-following, lane-changing and weaving operations and it has a limited capability of modeling transit and intermodal operations (FHWA, 2009). These limitations are overcome by some commercially available simulation software such as INTEGRATION and VISSIM that will be discussed in the following sections.

### 2.4.2. VISTA

VISTA, for Visual Interactive System for Transport Algorithms, was developed by Ziliaskopoulos and Waller (2000) based on DYNASMART. The main difference between the two models is the application of the traffic simulator named RouteSim (Ziliaskopoulos, Waller et al., 2004) that is different from the traffic simulator used in original DYNASMART proposed by Jayakrishnan and Mahmassani et al. (1994). RouteSim is based on the Cell Transmission Model (Daganzo, 1994; Daganzo, 1995). In addition to the replacement of traffic simulator, the path assignment module and time-dependent shortest path module were developed into a more efficient module that can handle large data sets, intersection movement delays, and link travel times. Finally, database support was added to handle critical data communication issues involving the link flows, travel times, and path storage, based on a key/data pairing database system (Ziliaskopoulos, Waller et al., 2004).

### 2.4.3. DynaMIT

DynaMIT is a real-time computer system designed to support the operations of ATIS and ATMS. Similar to DYNASMART-X, through effective integration of information databases with real-time inputs from field installations, DynaMIT efficiently achieves: (1) estimates of network conditions (2) predictions of network conditions in response to various traffic control measures and information dissemination strategies and (3) generation of traveler information to guide drivers towards optimal decisions (ITS-Program, 2010).

The main architect of DynaMIT is Ben-Akiva, whose pioneering work (Ben-Akiva, Koutsopoulos et al., 1996; Ben-Akiva, Bierlaire et al., 1997) laid the foundation for DynaMIT as a dynamic traffic assignment system to estimate and predict real-time current and future traffic patterns (Peeta and Ziliaskopoulos, 2001). It tackles the DTA problem from both the supply and demand aspects. Though no underlying mathematical program is formulated (Peeta and Ziliaskopoulos, 2001), the demand simulator estimates and predicts OD demand by applying a Kalman Filtering approach, which takes into consideration both the historical information and driver response to information (Ben-Akiva, Bierlaire et al., 1998; Sisiopiku and Xuping, 2006). The supply simulator is used to determine the flow pattern and it is a mesoscopic traffic simulator, where vehicles are moved in packets and links are divided into segments that include a moving part and a queuing part to model traffic flow (Peeta and Ziliaskopoulos, 2001).

### 2.4.4. INTEGRATION

INTEGRATION was initially developed by Van Aerde and Rakha (Rakha, Van Aerde et al., 1989; Van Aerde and Rakha, 1989). The strengths of the INTEGRATION model lie in a number of aspects. First, the model combines traffic assignment with microscopic simulation that captures car-following and lane-changing behaviors. Second, the model captures vehicle dynamics in terms of its acceleration behavior. Third, the model includes detailed microscopic energy, emission, and safety models (Rakha, Hellinga et al., 1996).

The traffic simulator module of the INTEGRATION model ensures traffic realism by incorporating the specific car-following logic that considers both speed-flow-density relationship and vehicle dynamics. The route selection logic that its traffic assignment module applies is a link-based routing logic proposed by Rilett and Van Aerde (1991a; 1991b). When INTEGRATION executes the time-dependent DTA routing logic, it computes the minimum path for every scheduled vehicle departure, in view of the link travel times anticipated in the network at the time the vehicle will reach these specific links. The anticipated travel time for each link is estimated based on anticipated link traffic volumes and queue sizes (Van Aerde & Assoc., 2005a; 2005b). The outcome of this routing logic is eventually conveyed to the simulated vehicle using a look-up table format.

This routing look-up table format provides, for each vehicle class, an indication of the next link to be taken towards a particular destination (Rakha, Hellinga et al., 1996).

### 2.4.5. VISSIM

VISSIM is a microscopic, time step and behavior based simulation model developed to model urban traffic and public transit operations (PTV, 2005). The traffic simulator of VISSIM is a microscopic traffic flow simulation model including car following and lane change logic. More specifically, the car-following model is the one proposed by Wiedemann in 1974 (PTV, 2005). The fundamental rule of this car-following logic is drivers of a faster moving vehicle starts to decelerate as he reaches his individual perception threshold to a slower moving vehicle. Since the speed of the slower vehicle is unknown to the driver of the faster vehicle, his speed will fall below that vehicle's speed until he starts to slightly accelerate again after reaching another perception threshold (PTV, 2005). It is obvious that this process involves iterative acceleration and deceleration.

VISSIM assumes that not everybody uses the best route but that less attractive routes are used as well, although by a minor part of the drivers (PTV, 2005). However, the K-shortest paths algorithm is not used to generate a set of best routes available for a certain OD pair as DYNASMART does. VISSIM searches for the best route for each O-D-pair in each iteration of the Dynamic Assignment. Different "best" routes in different iterations are stored and considered known to later iterations (PTV, 2005). The criterion for the route search is the combination of various costs such as travel time, financial cost and distance. Hence strictly speaking, the DTA VISSIM performs is not either UE or SO DTA.

## 2.5. Comparison of Simulation-based DTA Models

The comparison results are organized into a tabular format, as shown in Table 1. The comparison between DYNASMART, DynaMIT and VISTA is borrowed directly from (Sisiopiku and Xuping, 2006).

**Table 1 Comparison of Simulation-based DTA Models**

| Models | DYNASMART | DynaMIT | VISTA | INTEGRATION | VISSIM |
|---|---|---|---|---|---|
| **Approach** | • Heuristic<br>• UE & SO<br>• Mesoscopic, Moving queuing segments<br>• Modified Greenshield's speed-density relationship | • Heuristic<br>• UE<br>• Moving queuing segments<br>• Kalman filtering | • Exact and Heuristic<br>• UE & SO<br>• Cell Transmission Model | • Exact and Heuristic<br>• UE & SO<br>• Special car-following logic | • Heuristic<br>• Not UE nor SO<br>• Wiedemann car-following logic |
| **Evaluation Functionality** | • Short-term, long-term infrastructure and operational changes<br>• Both online and offline management tasks | • Optimizing the operation of TMCs through the provision of real-time predictions.<br>• Efficient operation of Variable Message Signs (VMS).<br>• Real-time incident management and control.<br>• Evaluation of alternative traffic signals and ramp meters operation strategies. | • Short-term, long-term infrastructure and operational changes<br>• Both online and offline management tasks | • Combines traffic assignment with microscopic simulation.<br>• Capture vehicle dynamics in terms of its acceleration behavior.<br>• Energy, emission, and safety models.<br>• Flexibility in modeling numerous Intelligent Transportation System (ITS) applications. | • Development and evaluation of signal priority logic<br>• Evaluate the feasibility and impact of integrating light rail into urban street networks.<br>• Analysis of slow speed weaving and merging areas<br>• Evaluate transit system |
| **Input Requirement** | • Demand profile<br>• Traffic control<br>• Incident information<br>• Network Geometry<br>• Information control | • Demand profile (departure-time based)<br>• Network Geometry | • Demand profile<br>• Traffic control<br>• Incident information | • Demand profile<br>• Traffic control<br>• Incident information<br>• Network Geometry<br>• Vehicle properties | • Demand profile<br>• Traffic control<br>• Incident information<br>• Network Geometry |

| Table 1 continued | | | | | |
|---|---|---|---|---|---|
| **Models** | **DYNASMART** | **DynaMIT** | **VISTA** | **INTEGRATION** | **VISSIM** |
| **Input Requirement** | | | • Network Geometry (created through VISTA or PSQL) | | • Transit settings |
| **Major Output** | • Link flows<br>• Link density<br>• Link average speed<br>• Vehicle trajectory and etc. | • Individual vehicle trajectories | • Cell occupancies<br>• Vehicle path and travel time | • Link flows<br>• Link density<br>• Vehicle travel time<br>• Vehicle positions, speeds and accelerations, etc. | • Link flows<br>• Link density<br>• Vehicle travel time<br>• Queue length and etc. |
| **Quality of Graphics** | Medium | Low | Low | Medium | High |

## 2.6. Summary

Analytical dynamic traffic assignment models mainly apply three categories of mathematical techniques, namely mathematical programming, optimal control theory and the variational inequality approach. A persistent issue is the need to balance mathematical tractability and traffic realism. Reasons for this problem include difficulty in incorporating traffic flow models and non-convexity invited by the FIFO condition. Research efforts are still needed to overcome the aforementioned difficulties and realize large-scale applications at reasonable expenses.

Simulation-based DTA models enjoy their advantage in handling issues that are troublesome in analytical formulations. The adoption of simulation and incorporation of well-established theoretical aspects of traffic speed-density-flow relationships circumvents the limitations of analytical forms of link performance functions. In addition, some traffic simulators are capable of capturing vehicle interaction and vehicle dynamics. All these factors, together with the ability to keep track of individual vehicles' paths in information-technology enabled context, represent the strengths of simulation-based DTA models (Peeta and Ziliaskopoulos, 2001).

Extending the works by Carey (2009) and Carey and Subrahmanian (2000), the proposed model deals with multiple origin-destination pairs and is capable of modeling incidents under the UE-DTA framework.

# Chapter 3 Model Formulation

The model presented here is an extension of the dynamic traffic assignment framework proposed by Carey et al. (1999; 2000; 2009). The following notation is used throughout the thesis.

**Table 2 Summary of Notation**

| Notation | Interpretation |
|---|---|
| $T$ | Total number of time intervals |
| s | OD pair s |
| $x^s_{tj\tau k}$ | Flow entering node $j$ at time$t$ and arriving at node $k$ at time $\tau$ for OD pair $s$ |
| $\bar{x}^s_{tj\tau k}$ | Capacity for OD pair $s$ for time space link $tj\tau k$ |
| $d^s_t$ | Demand for OD pair $s$ generated at time $t$ |
| $y_{jk}(t)$ | The output flow for a cell $j$ to its successor cell $k$ at time $t$ |
| $z^s_{wj}(t)$ | Size of the vehicle packet leaving cell $j$ at time $t$ whose entry time is $w$ and bounded for OD pair $s$ |
| $n_{wj}(t)$ | Size of a sub-packet whose entry time to cell $j$ is $w$ at time $t$ |
| $n^s_{wj}(t)$ | Size of a sub-packet whose entry time to cell $j$ is $w$ and belongs to OD pair $s$ at time $t$ |
| $D^s_j(t)$ | Number of vehicles generated for OD pair $s$ at time $t$ in cell $j$ |
| $I(tjk)$ | The set of exit time intervals for the flow entering spatial link $jk$ at time $t$ |
| $I^+(tjk)$ | Expanded format of $I(tjk)$, with a time-space link with unbounded capacity |
| $O(ltj)$ | The set of entering time intervals for the flow exiting spatial link $lj$ at time $t$ |
| $O^+(ltj)$ | Expanded format of $O(ltj)$, with a time-space link with unbounded capacity |
| $c_{tj\tau k}$ | Cost factor equal to the node k entering time and node j leaving time |
| $p_{tj}$ | the traversed path connecting node $j$ to the destination node at time $t$ |
| **CTM-related Notation** | |
| $\delta$ | Wave propagation constant |
| $E(j)$ | The immediate downstream nodes of node $j$ |
| $F(j)$ | The immediate upstream nodes of node $j$ |
| $A(j)$ | The immediate predecessor cells of cell $j$ |
| $B(j)$ | The immediate successor cells of cell $j$ |
| $C_R$ | The set of source cells |
| $C_O$ | The set of ordinary cells |
| $C_S$ | The set of sink cells |

## 3.1. Basic Assumptions

Consider a traffic network $G(N, A)$, with a set of nodes $N = \{1, \dots, j, k, \dots, |N|\}$ joined by a set $A$ of directed spatial links such as link $jk$. Each node $j$ is duplicated $T$ times if the whole planning horizon is divided into $T$ small time periods. This network with duplicate nodes is called a time-expanded network and the links between nodes are "time-space" links (Carey, 1999; Carey and Subrahmanian, 2000; Carey, 2009).

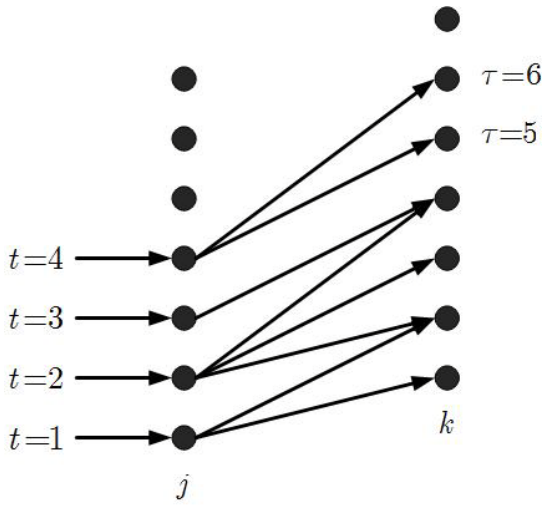


**Figure 1 Time-Expanded Network**

In this time expanded network, illustrated in Figure 1, inflow to a specific spatial node, say $j$, can exit the link $jk$ during multiple time intervals. For example, inflow at $t = 4$ exits in two time intervals which are $\tau = 5$ and $\tau = 6$. The expanded format of the spatial network $G(N, A)$ is denoted as $G^*(N, A)$.

This model only deals with freeway network. In other words, signalized networks are not considered. But the model can be adapted to model signalized surface streets by using a dynamic capacity term in the link flow model. See the example provided in the work by (Lo and Szeto (2002).

## 3.2. DTA Routing Model

Carey's DTA framework can be extended to incorporate multiple origin-destination pairs as follows. In the DTA routing model which is a linear programming model, the objective function minimizes total travel time of all the flows along all time-space links in a time-expanded network $G^*(N, A)$.

$$f(x) = Min \sum_{s} \sum_{t=1}^{T} \sum_{j \in N} \sum_{k \in E(j)} \sum_{\tau \in I(tjk)} c_{tj\tau k} x_{tj\tau k}^{s} \qquad (1)$$

where $c_{tj\tau k}$ is the cost factor equal to the difference between the node $k$ entering time and node $j$ leaving time $(\tau - t)$.

For any $j \in N$ that is not a destination associated with OD pair $s$, any OD pair $s$ and $t = 1,2,3, \dots, T$, we have the flow conservation constraint in eq. (2):

$$\sum_{l \in F(j)} \sum_{w \in O^+(ltj)} x^s_{wltj} + d^s_t = \sum_{k \in E(j)} \sum_{\tau \in I^+(tjk)} x^s_{tj\tau k} \quad (2)$$

where $O^+(ltj)$ and $I^+(tjk)$ are the expanded exit-flow time interval set and inflow time interval set respectively.

In addition, it should be noted that for destination nodes, there are no outflows and the total demand headed for this node should be equal to the total pertinent inflow throughout the whole planning horizon. Mathematically, it is expressed as:

$$\sum_{l \in F(j)} \sum_{w \in O^+(ltj)} \sum_{t=2}^{T} x^s_{wltj} = \sum_{t=1}^{T} d^s_t \quad (3)$$

where $j$ is the destination node of OD pair $s$.

The capacity constraint confines the maximum flows along each time-space link, which is expressed by eq. (4):

$$x^s_{tj\tau k} \leq \bar{x}^s_{tj\tau k}, \tau \in I(tjk) \quad (4)$$

Note that flow conservation constraints (eq. (2) and eq. (3)) use the expanded format of the set of link exit time intervals. The reasons are discussed as follows. Essentially, the capacity constraint (eq. (4)) excessively confines the flows over time-space links by not allowing flows from upstream links to enter the current link, which could lead to infeasibility of the whole program. Hence, an additional unbounded time-space link is introduced. Specifically, if the time-space link with highest cost is link $tj\bar{\tau}k$, then this extra time-space link is $tj(\bar{\tau}+1)k$ with unbounded capacity. Introduction of this unbounded time-space link expands the exit time set $I(tjk)$ to $I^+(tjk)$. For example, when vehicles entering link $jk$ at time $t$ try to leave the link, say if only two links with finite capacities are available and the total capacity of these two time-space links is less than the number of vehicles that try to leave, then not all the vehicles can leave the links. This will violate the flow conservation and

thus make the linear program infeasible. Therefore, this extra time-space link ensures users' free choice of spatial paths and feasibility of the program (Carey, 2009).

All the flows should be non-negative, which is expressed by:

$$x_{tj\tau k}^{s} \geq 0, \forall s, t, j \in F(k), \tau > t \tag{5}$$

The objective function in eq.(1), together with the flow conservation constraints (eq.(2), (3)), capacity constraint (eq.(4)) and non-negativity constraints (eq.(5)) completes the DTA routing model.

### 3.3. Link Flow Model

The link flow model adopts the Cell Transmission Model, which is capable of maintaining the temporal FIFO principle among multiple OD pairs. Only individual links are considered here. In other words, the CTM recipe is implemented within each spatial link separately. However, interaction between a link and its downstream links can still be captured. This will be explained after the CTM model is presented.

Since only individual links are taken into consideration, only ordinary cells, source cells and sink cells are needed here. The three types of cells are illustrated by Figure 2.

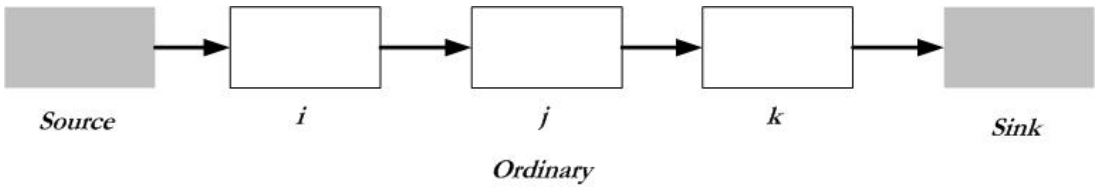


**Figure 2 Cell Partition within a Link**

The source cell is the place where demand from the DTA routing model is input. Cells $i$, $j$ and $k$ are ordinary cells which have downstream and upstream cells. The sink cell is the place where all the flows arrive eventually. A unique feature of this link flow model, each spatial link has its own dummy source cell and sink cell. In addition, vehicles arriving at the sink cell of a specific link do not necessarily successfully enter the source cell of a downstream link. The vehicles in the sink cell of an upstream link advance to the source cell of the downstream link depending on space availability,

consistent with the FIFO rule. The vehicles that fail to enter the downstream cells remain in the dummy sink cells until they can advance.

This procedure propagates multi-OD traffic based on the CTM recipe while maintaining the FIFO condition. The procedure can be conceptually described as follows. Each vehicle packet is made up of several sub-packets which are distinguished by entry time interval and OD pair. At each time interval throughout the whole planning horizon, the sub-packets that can leave are determined based on their respective entry time interval. In order to maintain temporal FIFO, the sub-packet that enters earlier should always leave earlier compared to those entering later. The specifics are illustrated in the coming paragraphs. The planning horizon starts from time interval $t = 1$ and it is assumed that initial conditions of all the cells are known. Specifically, all the occupancies and flows are set equal to zero at the very beginning.

The connection between the DTA routing model and the link flow model is the capacity of each time-space link $\bar{x}^s_{tj\tau k}$. Note that the capacity constraint is not applicable to the additional unbounded time-space link aforementioned. The exit time interval $\tau$ belongs to the set $I(tjk)$. After the solution to the DTA routing model is obtained, the flow variables $x^s_{tj\tau k}$ are summed as in eq. (6):

$$x^s_{tjk} = \sum_{\tau \in I(tjk)} x^s_{tj\tau k} \tag{6}$$

where $x^s_{tjk}$ signifies the total flow of OD pair $s$ entering the spatial link $jk$ at time $t$. This total flow will be input into the link flow model as demand generated in the dummy source cell for the spatial link $jk$. In other words, eq.(4) and (6) are the connection between the DTA routing model and link flow model. In addition, the flow conservation constraint (eq. (4)) precludes the possibility of vehicle disappearance. Since the inflows to each spatial link are summed and fed into the link flow model using eq. (6), there is no discontinuity between the spatial and time-space links.

The first step of the link flow model finds the aggregate flow according to the CTM flow propagation equations (Daganzo, 1995) which are presented as follows. At any time $t$, the output flow for cell $j$ to its successor cell $k$ can be calculated by the following equation:

$$y_{jk}(t) = min\{n_j(t), Q_j(t), Q_k(t), \delta[N_k(t) - n_k(t)]\}, k \in B(j) \tag{7}$$

It can be seen that the outflow is determined by cell capacities, denoted by $Q_j(t)$ and $Q_k(t)$, current occupancy $n_j(t)$ and the shockwave effect represented by the term $\delta[N_k(t) - n_k(t)]$. Incidents are

modeled through the cell capacity parameter. Specifically, incident severity is defined as the percentage of capacity lost due to a certain incident. This capacity drop is introduced to the cell where the incident occurs during the incident duration. For example, if an incident removing $\theta$ percent of the original capacity occurs at cell $j$ at time $t$, the resulting capacity of the cell can be mathematically expressed as:

$$Q_j'(t) = (1-\theta)Q_j \tag{8}$$

With the outflow, the cell occupancies can be updated by using eq.(9) as follows.

$$n_j(t+1) = n_j(t) - y_{jk}(t) + y_{ij}(t) + \sum_s D_j^s(t), k \in B(j), i \in A(j) \tag{9}$$

Note that if cell $j$ is a source cell there is no inflow to it, or equivalently $y_{ij}(t) = 0, \forall t, i \in A(j)$. If cell $j$ is a sink cell then no demand would be generated in this cell and there is no output flow at any time. In other words, two conditions $y_{jk}(t) = 0$ and $D_j^s(t) = 0$ hold at any time $t$. Ordinary cells can have inflow and outflow but no demand. Hence, eq.(9) can be used to update cell occupancy for any category of cell at any time.

In order to determine which sub-packets can leave at a specific time period, it is necessary to identify the sub-packets that are present in the current time interval. It has to be noted that the size of some vehicle sub-packets may be zero since it is possible that no vehicles enter this cell at a certain time. For cell $j$ at time interval $t$, the sub-packets may enter this cell at any time interval prior to time interval $t$. If the number of different OD pairs is $m$, then totally the maximum number of sub-packets within a cell at any time $t$ is $m \times t$. For example, at $t = 2$, the maximum number of sub-packets within cell $j$ is 4 if there are two OD-pairs.

Since sub-packets with early entry time should always leave earlier regardless of their destinations, it is essential to know occupancy disaggregated only by entry time interval at time $t$, which is computed by:

$$n_{wj}(t) = \sum_s n_{wj}^s(t) \tag{10}$$

There should always exist an entry time interval $\overline{w}$ such that for any time interval $t$,

$$\sum_{w \le \overline{w}} n_{wj}(t) \le y_{jk}(t) \le \sum_{w \le \overline{w}+1} n_{wj}(t), \overline{w} \le t, k \in B(j) \tag{11}$$

In eq. (11), $y_{jk}(t)$ is the outflow for cell $j$ to its successor cell $k$ at time $t$, which is known from eq. (7). Eq. (11) indicates that the packets with an arrival time interval earlier than $\bar{w}$ should advance to the downstream cell in their entirety while the ones arriving later than this specific time interval $\bar{w}$ should be split. In addition, we can identify the packets leaving as a whole for OD pair $s$ by the summation $\sum_{w \le \bar{w}} n^s_{wj}(t)$. Eq. (11) is used to identify the packets that should leave in their entirety.

The temporal FIFO principle requires that vehicle sub-packets $n_{wj}(t)$ that enter before and at $\bar{w}$ should leave while only part of the sub-packet $n_{(\bar{w}+1)j}(t)$ can leave. Since we have multi-OD flows, we need to determine the proportions of this sub-packet allocated to each OD pair. The procedures presented in the work by Ge and Carey (2004) are used to determine these proportions. The remaining part in the outflow packet after excluding packets leaving in their entirety is calculated as:

$$z_{(\bar{w}+1)j}(t) = y_{jk}(t) - \sum_{w \le \bar{w}} n_{wj}(t)\,, k \in B(j) \tag{12}$$

In eq. (12), $\sum_{w \le \bar{w}} n_{wj}(t)$ calculates the packets leaving in their entirety after the interval $\bar{w}$ has been determined by eq.(11). The outflow allocated to each OD-pair is proportional to the size of the sub-packet of this OD-pair. The mathematical expression takes the form:

$$z^s_{(\bar{w}+1)j}(t) = \frac{n^s_{(\bar{w}+1)j}(t)}{n_{(\bar{w}+1)j}(t)} \times z_{(\bar{w}+1)j}(t), \forall s \tag{13}$$

By using eq. (10) to (13), we can determine the output flow disaggregated only by OD-pair,

$$y^s_{jk}(t) = \sum_{w \le \bar{w}} n^s_{wj}(t) + z^s_{(\bar{w}+1)j}(t), \forall s, k \in B(j) \tag{14}$$

So far we have determined the packets that can leave cell $j$ at time interval $t$; additional manipulations are needed to update the vehicle packet lists for each cell other than the sink cell of a spatial link. More specifically, the outflow disaggregated by both entry time interval and OD-pair are calculated as follows. For notational clarity purposes, a single variable $f^s_{wj}(t)$ is introduced to denote the flow for OD pair $s$ leaving cell $j$ at time $t$ whose entry time is $w$. It can be used to update the packet list for each cell.

$$f^s_{wj} = \begin{cases} n^s_{wj}(t), w \le \bar{w} \\ z^s_{wj}(t), w = \bar{w} + 1 \\ 0, w > \bar{w} + 1 \end{cases} \tag{15}$$

Eq. (15) summarizes the flow calculation results presented in eq. (10) to eq. (14).

- If the entry time $w$ of a packet is earlier than $\bar{w}$ (determined in eq. (11)), it is supposed to leave the current cell in its entirety. Hence, we have $f_{wj}^{s}(t) = n_{wj}^{s}(t)$.
- If the entry time $w$ of a packet is equal to $\bar{w} + 1$, this packet cannot leave as a whole. Only part of it can leave cell $j$. Thus we have $f_{wj}^{s}(t) = z_{wj}^{s}(t)$ and
- If the entry time $w$ of a packet is later than $\bar{w} + 1$, it will not leave cell $j$.

In order to continue to track disaggregated flows and occupancy, we need to update the occupancy for the next time step. Flows leave cell $j$ so we have:

$$n_{wj}^{s}(t+1) = n_{wj}^{s}(t) - f_{wj}^{s}(t) \tag{16}$$

Once a packet is allowed to leave, this output flow leaving at time $t$ from cell $j$ arrives at cell $k$ after one time interval. As a result, we have:

$$n_{(t+1)k}^{s}(t+1) = y_{jk}^{s}(t), \forall s, j \notin C_s, k \in B(j) \tag{17}$$

In addition, if we consider the external demand as inflow to the source cell, in a similar fashion to eq.(17), we have:

$$n_{(t+1)j}^{s}(t+1) = D_j^s(t), j \in C_R, \forall s \tag{18}$$

Using eq. (16), (17) and (18), we calculate the disaggregated occupancies for the next time period.

Cell FIFO ensures path FIFO as well as OD FIFO, which was proved by Lo and Szeto (2002). Note that here FIFO refers to temporal FIFO. In other words, spatial FIFO is not preserved since vehicles' physical locations are not tracked. Hence, the cumulative arrival disaggregated by OD-pair in the sink cell can be used to determine the travel time for the whole link. The procedures used are explained with a concrete example.

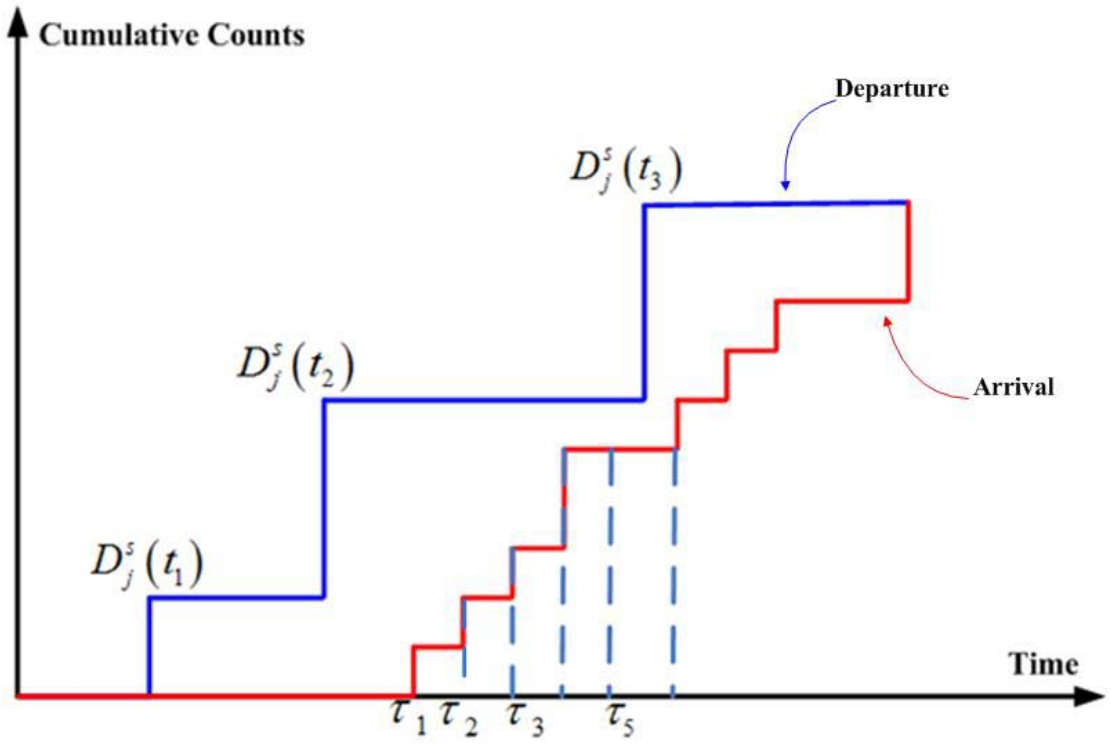


**Figure 3 Cumulative Counts of Vehicles of a Link**

If a spatial link $jk$ is considered, in Figure 3, demand for OD pair $s$ is generated at three distinct time intervals, namely $t_1$, $t_2$ and $t_3$. We can know how many vehicles arrive at every single time interval. It can be seen from Figure 3 that all the vehicles generated at $t_1$ arrive by time $\tau_2$. Hence, we can know the flow along time-space links $x^s_{t_1 j \tau_1 k}$and $x^s_{t_1 j \tau_2 k}$ . Additionally, the travel time for vehicles is $(\tau_2 - t_1)$ on the second time-space link. Note that there may be no flow along certain time-space links. For example, at time $\tau_5$, there are no vehicles arriving. Hence, the flow $x^s_{t_2 j \tau_5 k} = 0$.

## 3.4. Properties of the Proposed Model

It can be seen that the outflows obtained from the Cell Transmission Model, which is the link flow model, are disaggregated by OD pairs. In other words, the disaggregated flows maintain temporal FIFO conditions between multiple OD pairs. It is true that flows bound for different destinations share the same time-space link and thus it is argued that a single aggregate capacity constraint should be exerted on the time-space links. However, in order to maintain temporal FIFO, disaggregated capacity constraints are used for time-space links in the DTA routing model.

The reason why queuing effects can still be captured by the proposed model is explained below. The flow conservation constraint, expressed in eq. (2), indicates that flow entering at time $t$ equals the flow that leaves at the same time. Graphically, it is depicted as Figure 4 shows. Recall that here the sub-script $t$ refers to the entering time of the flow packet. The inflow to spatial node $j$ at time interval $t$ equals the outflow from the node $j$ in two time intervals.

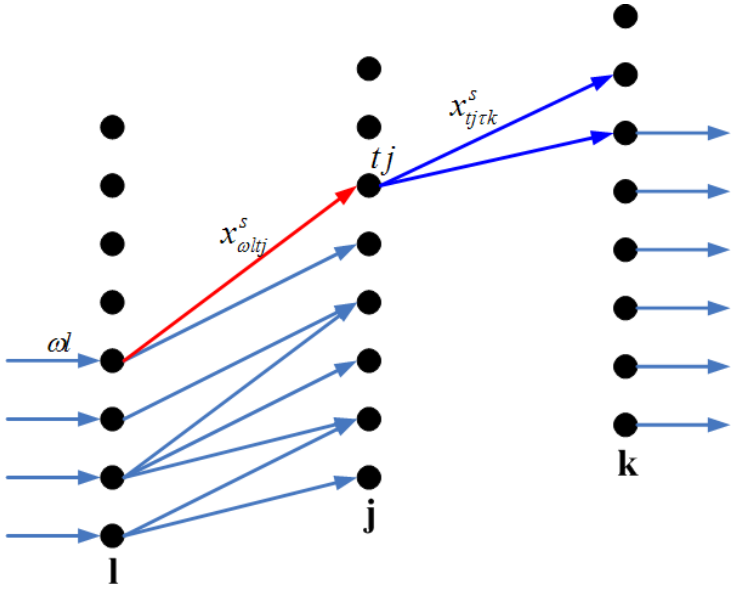


**Figure 4 Flow Conservation between Time-space Links**

As mentioned in the previous section, each link is associated with a dummy source cell and sink cell. The inflow to a specific time-expanded node, say $tj$ , is treated as demand generated at time $t$ in the dummy source cell. Due to flow conservation, this inflow is effectively equal to the outflow at time-expanded node $tj$. Moreover, this "dummy demand" is loaded in a strict chronological order from the beginning of the modeling horizon to the end. Hence, even if all the flows can move into the sink cells, this does not necessarily mean that they have entered the downstream link, which may not have adequate available space for additional incoming flows. Recall that each link has its own sink cell and source cell and the sink cell of a link is not the source cell of its downstream link. Therefore, entering the sink cell means the flows have left the link that contains this sink cell however it does not necessarily mean that these flows have entered the downstream link. Since demand is loaded sequentially, it is possible that the first cell of the downstream link cannot receive a vehicle packet at a certain time $t$ . Vehicles have to wait for a certain period of time. In this way, this model is able to capture the temporal effect of spillback. The inflow at time $t$ may not advance immediately due to congestion in downstream cells. As a result, the temporal effect of spillback, which is longer waiting

time is captured between links. It should be noted that within each link, the spillback effects can be fully captured by the CTM recipe.

### 3.5. Proof of Equivalence to DUE

According to Carey (1999), three categories of paths can be identified:

i. Fully utilized paths are paths on which one or more links constituting the path carry flow to full capacity.
ii. Partially utilized paths are paths on which every link carries flow, but no links have flow to their full capacity.
iii. Available unutilized paths are paths on which there is at least one link carrying no flow while other links carry flow under the capacity limit.

The definition for dynamic user equilibrium in a time-expanded network is as follows (Carey, 1999; Carey, 2009):

**DUE Definition**: "Links have fixed capacities. A traffic assignment is a UE if for each OD pair, FIFO is satisfied and the trip time on all partially utilized paths is not higher than the trip time on any available unutilized path, and is not lower than the trip time on any fully utilized path" (Carey, 1999; Carey, 2009). Note that all paths in this definition refer to time-space paths.

The DTA routing model is constituted by eqs. (1)~(4). For notational simplicity, eq.(2) can be rewritten as:

$$g_{tj}(\boldsymbol{x}) = \sum_{l \in F(j)} \sum_{\omega \in O(tlj)} x^s_{\omega ltj} + d^s_t - \sum_{k \in E(j)} \sum_{\tau \in I(tjk)} x^s_{tj\tau k}, \{\lambda_{tj}\} \qquad (19)$$

$\lambda_{tj}$ and $\alpha^s_{tj\tau k}$ are non-negative dual variables associated with their respective constraints and $\alpha^s_{tj\tau k}$ corresponds to the capacity constraint (eq.(4)). The proof uses Kuhn-Tucker optimality conditions which are necessary and sufficient for a linear program since a linear program is always a convex optimization problem (Bertsekas, 1999). For a specific OD pair $s$, the Lagrangian of the linear program for the constraints other than the non-negativity constraint can be formulated as follows. Note that the OD-pair super-script is dropped for simplicity.

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_t \sum_j \lambda_{tj} g_{tj}(\boldsymbol{x}) + \sum_t \sum_j \sum_\tau \sum_k \alpha_{tj\tau k} \left( x_{tj\tau k} - \bar{x}_{tj\tau k} \right) \qquad (20)$$

The Karush-Kuhn-Tucker (KKT) condition is used. First, the stationarity condition, expressed in (21), should hold. Specifically, at the stationary point $x^*$, we have

$$\nabla L(x^*) = 0 \tag{21}$$

In addition to stationarity condition, the primal feasibility conditions have to hold:

$$\begin{cases} g_{tj}\left(x^*_{tj\tau k}\right) = 0 \\ x^*_{tj\tau k} \geq 0 \\ x^*_{tj\tau k} \leq \bar{x}^*_{tj\tau k} \end{cases} \text{, for } \forall t, j, \tau, k \tag{22}$$

Regarding the dual variable $\alpha_{tj\tau k}$ associated with the capacity constraint, we have dual feasibility condition and complementary slackness condition. The first inequality in eq. (23) is the dual feasibility condition and the equality is the complementary slackness condition.

$$\begin{cases} \alpha_{tj\tau k} \geq 0 \\ \alpha_{tj\tau k}\left(x_{tj\tau k} - \bar{x}_{tj\tau k}\right) = 0 \end{cases} \text{for } \forall t, j, \tau, k \tag{23}$$

In addition to eq. (23), we have the first-order condition for the non-negativity constraint as shown in eq. (24):

$$\begin{cases} x_{tj\tau k}\dfrac{\partial L}{\partial x_{tj\tau k}} = 0 \\ \dfrac{\partial L}{\partial x_{tj\tau k}} \geq 0 \end{cases} \text{for } \forall t, j, \tau, k \tag{24}$$

In expanded format, eq. (24) can be written as:

$$\begin{cases} x_{tj\tau k}\left(c_{tj\tau k} - \lambda_{tj} + \alpha_{tj\tau k}\right) = 0 \\ c_{tj\tau k} - \lambda_{tj} + \alpha_{tj\tau k} \geq 0 \end{cases} \tag{25}$$

By definition, the actual path travel time ($\mathrm{p.t.t}$) is the summation of travel times of all the links actually traversed along the path irrespective of the path category. Mathematically,

$$p.t.t = \sum_{(t'j'\tau'k') \in p_{tj}} c_{t'j'\tau'k'} \tag{26}$$

where $p_{tj}$ denotes a time-space path connecting node $j$ to the destination node at time $t$. $c_{t'j'\tau'k'}$ indicates the cost factor for the time-space link taken by the vehicles. From eq.(23), we know:

$$c_{tj\tau k} + \alpha_{tj\tau k} \geq \lambda_{tj} \tag{27}$$

We can write an equation similar to eq. (27) for each time-space node $t'j'$ along the path $p_{tj}$. By adding these equations, we have:

$$\sum_{t'j' \in p_{tj}} \lambda_{t'j'} \leq \sum_{(t'j'\tau'k') \in p_{tj}} c_{t'j'\tau'k'} + \sum_{(t'j'\tau'k') \in p_{tj}} \alpha_{t'j'\tau'k'} \tag{28}$$

- If path $p_{tj}$ is an available unutilized path, then flows on all the links are below capacity. As a result, by the complementary slackness condition in eq.(23) we know all the dual variables $\alpha_{tj\tau k}$ are equal to zero. Hence, eq. (28) becomes:

$$\sum_{t'j' \in p_{tj}} \lambda_{t'j'} \leq \sum_{(t'j'\tau'k') \in p_{tj}} c_{t'j'\tau'k'} = p.t.t \tag{29}$$

Note that the right-hand side of eq. (29) is the actual path travel time expressed in eq. (26).

- If path $p_{tj}$ is a fully utilized path, then flows on all the links are greater than zero. So by complementary slackness in eq. (23), eq. (28) hold as a strict equality as follows:

$$\sum_{t'j' \in p_{tj}} \lambda_{t'j'} = \sum_{(t'j'\tau'k') \in p_{tj}} c_{t'j'\tau'k'} + \sum_{(t'j'\tau'k') \in p_{tj}} \alpha_{t'j'\tau'k'} \geq p.t.t \tag{30}$$

- If path $p_{tj}$ is a partially utilized path, then all the flows are below capacity but greater than zero. Hence we have:

$$\sum_{t'j' \in p_{tj}} \lambda_{t'j'} = \sum_{(t'j'\tau'k') \in p_{tj}} c_{t'j'\tau'k'} = p.t.t \tag{31}$$

From eq. (29), (30) and (31) , we can know that at any time $t$, travel time along a partially utilized path is not higher than the trip time on any available unutilized path, and is not lower than the trip time on any fully utilized path. This is consistent with the dynamic user equilibrium definition presented at the beginning of this section.

# Chapter 4 Computational Experience of the Model

This chapter, first, presents the solution algorithm of the model and then numerical examples are provided.

## 4.1. Solving the Model

The model is solved by an iterative approach. First, the capacity of all the time-space links is set to be a sufficiently large number. Then we solve the DTA routing model. Then by using eq.(6), the demand for the link flow sub-model is calculated. The next step is to use the CTM flow model expressed by eq.(7) to eq. (18). The output flow at each time interval can be determined with the help of cumulative counts at the dummy sink cell. Then these output flows are used as capacity constraints in the routing module in the next iteration. The solution procedure keeps iterating between the dynamic routing module and the link flow module until convergence is reached. This solving procedure is depicted by the following flow chart.

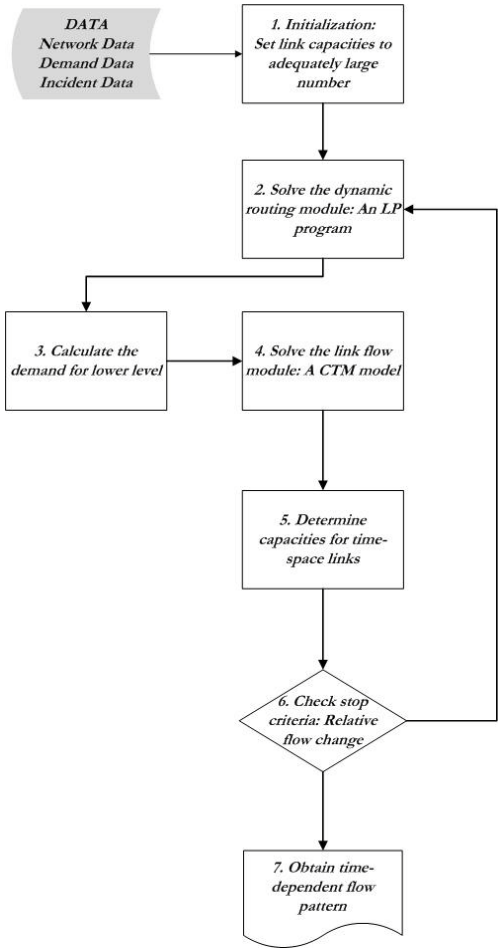


**Figure 5 Solution Procedure**

## 4.2. Numerical Examples

The DTA routing model is a network problem and it is programmed with LINGO, a commercial optimization package. The link flow model is coded with Visual C# for the sake of processing speed. The code is shown in the Appendix. In order to test both single OD pair and multiple OD pairs scenarios, two test networks are constructed and displayed in Figure 6. The cell networks are shown in Figure 7.

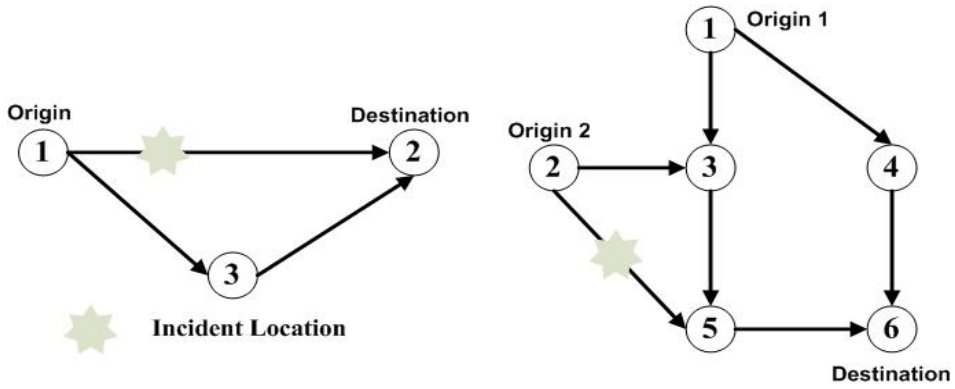


**Figure 6 Two Test Networks**

### 4.2.1. Single Origin Destination Pair

The basic network characteristics are shown in Table 3. The cell length is equal to the distance traveled by free-flowing traffic in one time interval (Daganzo, 1995). The length of the time interval adopted is 10 seconds. Hence, the cell lengths for respective links can be calculated.

**Table 3 Basic Characteristics of Single OD Pair Network**

| Link | Free Flow Speed (mph) | Jam Density (veh/mile) | Saturation Flow Rate (veh/hr/lane) | Number of Lanes | Length (miles) |
|---|---|---|---|---|---|
| Link (1,2) | 60 | 200 | 2200 | 3 | 0.4 |
| Link (1,3) | 45 | 200 | 1800 | 2 | 0.3 |
| Link (3,2) | 45 | 200 | 1800 | 2 | 0.3 |

The time-dependent demand (Table 4) is loaded onto the network and the program gives the traffic flows over the entire planning horizon as shown in Table 5.

**Table 4 Demand Profile for Single OD Network**

| Time Interval (sec) | 10 | | |
|---|---|---|---|
| **Generation Node** | **Generation Time (ith interval)** | **OD Pair** | **Number of Vehicles** |
| 1 | 1 | (1,2) | 30 |
| 1 | 2 | (1,2) | 40 |

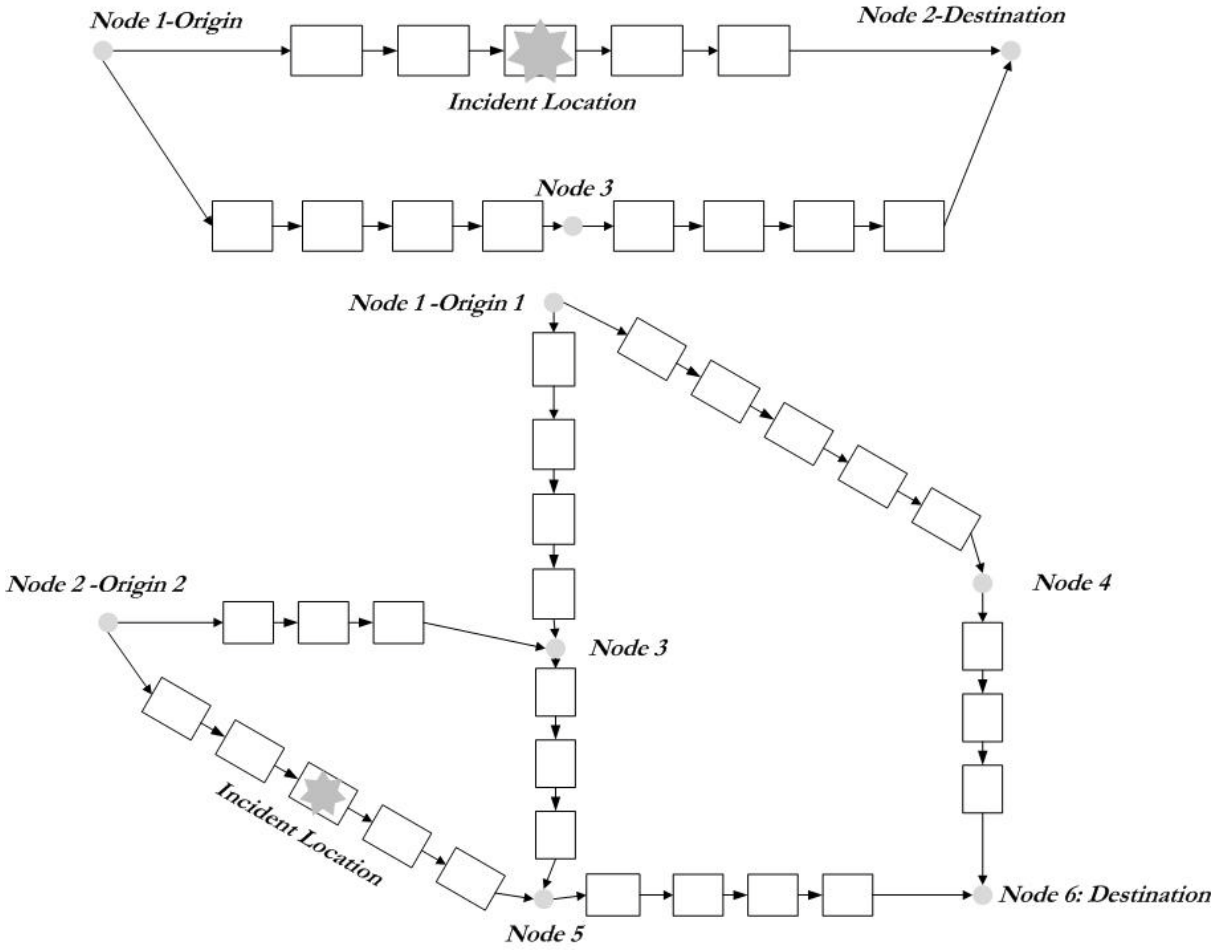


**Figure 7 Cell Networks**

**Table 5 Flow Pattern of Single OD Network (No Incident)**

| Program Execution Time (sec) | Planning Horizon (sec) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (veh*sec) |
|---|---|---|---|---|---|
| 19.965 | 200 | Link (1,2) | 0 (No Incident) | N/A | 421.67 |
| **Departure Time (ith interval)** | **Departure Node** | **Arrival Time (ith interval)** | **Arrival Node** | **OD Pair** | **Size (veh)** |
| 1 | 1 | 6 | 2 | 1 | 18.33 |
| 1 | 1 | 7 | 2 | 1 | 11.67 |
| **2** | **1** | **6** | **3** | **1** | **1.67** |
| 2 | 1 | 7 | 2 | 1 | 3.33 |
| 2 | 1 | 8 | 2 | 1 | 18.33 |
| 2 | 1 | 9 | 2 | 1 | 15.00 |
| **2** | **1** | **10** | **2** | **1** | **1.67** |
| **6** | **3** | **10** | **2** | **1** | **1.67** |

It can be seen that most vehicles take Link (1,2) which is obviously shorter compared to the other path. At the $10^{th}$ time interval, the first time-space path through spatial node 1 and 2 has the travel time of $10 - 2 = 8$ time intervals and the travel time of the other time-space path is $(6 - 2) + (10 - 6) = 8$ time intervals. Note that the first path is a fully-utilized path and the second path is a partially utilized path. This partially utilized path whose travel time is not lower than the fully capacitated path. This finding is consistent with the DUE definition. In the later explanation of consistency with DUE definition, the paths mentioned refer to time-space paths.

Two incident scenarios, distinguished by incident severity, are tested for the single OD network. As a matter of fact, dynamic incident severity can be easily modeled by introducing a time dimension to the severity index, which is written as $\theta$ in eq.(8). The dynamic incident severity index can be written as $\theta(t)$. In this way, the incident scenario could be simulated in a more realistic manner. The flow pattern for the two incident scenarios are displayed in Table 6 and Table 7.

**Table 6 Flow Pattern Single OD Network (Incident Severity = 0.6)**

| Program Execution Time (sec) | Planning Horizon (sec) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (veh*sec) |
|---|---|---|---|---|---|
| 17.77 | 200 | Link (1,2) | 0.6 | 4-8 | 501.67 |
| **Departure Time (ith interval)** | **Departure Node** | **Arrival Time (ith interval)** | **Arrival Node** | **OD Pair** | **Size (veh)** |
| 1 | 1 | 6 | 2 | 1 | 18.33 |
| 1 | 1 | 7 | 2 | 1 | 5.70 |
| 1 | 1 | 8 | 2 | 1 | 5.70 |
| 1 | 1 | 9 | 2 | 1 | 0.26 |
| 2 | 1 | 6 | 3 | 1 | 1.67 |
| 2 | 1 | 7 | 2 | 1 | 1.63 |
| 2 | 1 | 7 | 3 | 1 | 8.33 |
| 2 | 1 | 8 | 2 | 1 | 1.63 |
| 2 | 1 | 9 | 2 | 1 | 7.07 |
| 2 | 1 | 10 | 2 | 1 | 7.33 |
| 2 | 1 | 11 | 2 | 1 | 12.33 |
| 6 | 3 | 10 | 2 | 1 | 1.67 |
| 7 | 3 | 11 | 2 | 1 | 8.33 |

It can be seen clearly that flows still take the shorter path as expected. However, due to the occurrence of an incident, its capacity becomes small and is quickly fully occupied. This is evidenced by a decreasing number of vehicles exiting the shorter link. At first, the size of the exiting packet is 18.33 but later it drops continuously. At the 9$^{th}$ interval, the first path reaches its full capacity. The travel time for the other path (through Node 1, 3, 2) is $(6-2)+(10-6)=8$ time intervals while travel time on Link (1,2) is $9-1=8$ time intervals. Note that path 2 (Node 1,3,2) is a partially utilized path whose travel time is not lower than the fully capacitated path. This finding is consistent with the DUE definition.

At the 2$^{nd}$ departure time interval, the travel times on two time-space paths are equal. Specifically, the travel time for the first path is $11-2=9$ time intervals while the travel time for the other path is $(7-2)+(11-7)=9$ time intervals. The scenario in which a major incident with severity index equal to 1 occurs is also tested and the results are shown in Table 7.

**Table 7 Flow Pattern of Single OD Network (Incident Severity = 1)**

| Program Execution Time (sec) | Planning Horizon (sec) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (veh*sec) |
|---|---|---|---|---|---|
| 33.87 | 200 | Link (1,2) | 1 | 4-8 | 636.67 |
| **Departure Time (ith interval)** | **Departure Node** | **Arrival Time (ith interval)** | **Arrival Node** | **OD Pair** | **Size (veh)** |
| 1 | 1 | 6 | 2 | 1 | 18.33 |
| 1 | 1 | 11 | 2 | 1 | 1.67 |
| 2 | 1 | 6 | 3 | 1 | 10.00 |
| 2 | 1 | 7 | 3 | 1 | 1.67 |
| 2 | 1 | 8 | 3 | 1 | 10.00 |
| 2 | 1 | 9 | 3 | 1 | 6.67 |
| 2 | 1 | 11 | 2 | 1 | 13.33 |
| 2 | 1 | 12 | 2 | 1 | 8.33 |
| 6 | 3 | 12 | 2 | 1 | 10.00 |
| 7 | 3 | 13 | 2 | 1 | 1.67 |
| 8 | 3 | 14 | 2 | 1 | 10.00 |
| 9 | 3 | 15 | 2 | 1 | 6.67 |

It can be observed that vehicles take the shorter path (Link (1,2)) first. When an incident seriously undermines the link capacity, vehicles take the parallel path. When the shorter path is used at capacity, the travel time for this time-space path is $11 - 1 = 10$ time intervals. We can see the travel time of the parallel time-space path is equal to 10 time intervals. In other words, the shorter path, which is the fully utilized path, has the travel time which is not higher than that of the partially utilized path, which is the parallel path (Link (1,3), Link (3,2)). This is consistent with the DUE definition.

### 4.2.2. Multiple Origin Destination Pairs

Like the single OD pair network, three scenarios are tested for the multiple OD pairs network shown in Figure 6. The basic information and demand profile are provided in Table 8 and Table 9.

**Table 8 Basic Characteristics of the Multiple OD Pairs Network**

| Link | Free Flow Speed (mph) | Jam Density (veh/mile) | Saturation Flow Rate (veh/hr/lane) | Number of Lanes | Length (miles) |
|---|---|---|---|---|---|
| Link (1,3) | 45 | 200 | 1800 | 2 | 0.3 |
| Link (1,4) | 60 | 200 | 2200 | 3 | 0.4 |
| Link (2,3) | 45 | 200 | 1800 | 2 | 0.2 |
| Link (2,5) | 60 | 200 | 2200 | 3 | 0.4 |
| Link (3,5) | 45 | 200 | 1800 | 2 | 0.2 |
| Link (4,6) | 45 | 200 | 1800 | 2 | 0.2 |
| Link (5,6) | 45 | 200 | 1800 | 2 | 0.3 |

**Table 9 Demand Profile for Multiple OD Pairs Network**

| Time Interval (sec) | 10 | | |
|---|---|---|---|
| **Generation Node** | **Generation Time (ith interval)** | **OD Pair** | **Number of Vehicles** |
| 1 | 1 | (1,6) | 20 |
| 1 | 2 | (1,6) | 25 |
| 2 | 1 | (2,6) | 15 |
| 2 | 3 | (2,6) | 25 |

The assignment results for the three cases are shown in Table 10, Table 11 and Table 12 .

**Table 10 Flow Pattern for Multiple OD Pairs Network (No Incident)**

| Program Execution Time (seconds) | Planning Horizon (seconds) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (seconds) |
|---|---|---|---|---|---|
| 121.14 | 180 | / | 0 (No Incident) | N/A | 839.14 |
| **Departure Time** | **Departure Node** | **Arrival Time** | **Arrival Node** | **OD Pair** | **Size** |
| 1 | 1 | 6 | 4 | 1 | 18.33 |
| 1 | 1 | 7 | 4 | 1 | 1.67 |
| 1 | 2 | 6 | 5 | 2 | 15.00 |
| 2 | 1 | 7 | 4 | 1 | 13.33 |
| 2 | 1 | 8 | 4 | 1 | 1.67 |
| 2 | 1 | 9 | 4 | 1 | 10.00 |
| **3** | **2** | **6** | **3** | **2** | **1.81** |
| 3 | 2 | 8 | 5 | 2 | 18.33 |
| **3** | **2** | **10** | **5** | **2** | **4.85** |
| 6 | 4 | 9 | 6 | 1 | 10.00 |
| 6 | 4 | 10 | 6 | 1 | 8.33 |
| 6 | 5 | 10 | 6 | 2 | 10.00 |
| 6 | 5 | 11 | 6 | 2 | 5.00 |
| **6** | **3** | **9** | **5** | **2** | **1.81** |
| 7 | 4 | 11 | 6 | 1 | 1.67 |
| 7 | 4 | 12 | 6 | 1 | 13.33 |
| 8 | 4 | 12 | 6 | 1 | 1.00 |
| 8 | 4 | 13 | 6 | 1 | 0.67 |
| 8 | 5 | 12 | 6 | 2 | 10.00 |
| 8 | 5 | 13 | 6 | 2 | 8.33 |
| 9 | 4 | 14 | 6 | 1 | 10.00 |
| **9** | **5** | **14** | **6** | **2** | **1.81** |
| **10** | **5** | **14** | **6** | **2** | **4.85** |

For OD pair (1,6), all the flows take the path connecting nodes 1,4,6 while for the other OD pair both paths connecting the OD pair are chosen. At the 3rd time interval, vehicles started to take the alternative route (Link (2,3), Link (3,5) and Link (5,6)). The travel time for the first path is (10-3) + (14-10) = 11 time intervals, which is equal to that of the alternative path. In other words, the shorter path, which is the fully utilized path, has the travel time which is not higher than the cost of the partially utilized path.

**Table 11 Flow Pattern for Multiple OD Pair Network (Incident Severity = 0.6)**

| Program Execution Time (seconds) | Planning Horizon (seconds) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (seconds) |
|---|---|---|---|---|---|
| 122.94 | 180 | Link (2,5) | 0.6 | 4-10 | 870.63 |
| **Departure Time** | **Departure Node** | **Arrival Time** | **Arrival Node** | **OD Pair** | **Size** |
| 1 | 1 | 6 | 4 | 1 | 18.33 |
| 1 | 1 | 7 | 4 | 1 | 1.67 |
| 1 | 2 | 6 | 5 | 2 | 7.33 |
| 1 | 2 | 7 | 5 | 2 | 7.33 |
| 1 | 2 | 8 | 5 | 2 | 0.33 |
| 2 | 1 | 7 | 4 | 1 | 13.33 |
| 2 | 1 | 8 | 4 | 1 | 10 |
| 2 | 1 | 9 | 4 | 1 | 1.67 |
| **3** | **2** | **6** | **3** | **2** | **4.27** |
| 3 | 2 | 8 | 5 | 2 | 7 |
| 3 | 2 | 9 | 5 | 2 | 7.33 |
| **3** | **2** | **10** | **5** | **2** | **4.63** |
| 4 | 2 | 8 | 3 | 2 | 1.77 |
| **6** | **3** | **9** | **5** | **2** | **4.27** |
| 6 | 4 | 9 | 6 | 1 | 10 |
| 6 | 4 | 10 | 6 | 1 | 8.33 |
| 6 | 5 | 10 | 6 | 2 | 7.33 |
| 7 | 4 | 11 | 6 | 1 | 1.67 |
| 7 | 4 | 12 | 6 | 1 | 10 |
| 7 | 4 | 13 | 6 | 1 | 3.33 |
| 7 | 5 | 11 | 6 | 2 | 7.33 |
| 8 | 3 | 12 | 5 | 2 | 1.77 |
| 8 | 4 | 13 | 6 | 1 | 6.67 |
| 8 | 4 | 14 | 6 | 1 | 3.33 |
| 8 | 5 | 12 | 6 | 2 | 7 |
| 8 | 5 | 14 | 6 | 2 | 0.33 |
| 9 | 4 | 15 | 6 | 1 | 1.67 |
| 9 | 5 | 13 | 6 | 2 | 2.3 |
| **9** | **5** | **14** | **6** | **2** | **9.27** |
| **10** | **5** | **14** | **6** | **2** | **6.4** |
| **12** | **5** | **16** | **6** | **2** | **1.77** |

At the beginning, for OD pair 2, the shorter path was taken. At the 3rd time interval, vehicles started to take the alternative route (Link (2,3), Link (3,5) and Link (5,6)). The travel time for both paths is

11 time intervals. In other words, the shorter path, which is the fully utilized path, has the travel time which is not higher than that of the partially utilized path. It can be seen that flows took the alternative path for OD pair (2,6) after the incident occurred. It is expected that more flows will take this longer path when a major incident happens. The results shown in Table 12 confirm this inference.

**Table 12 Flow Pattern for Multiple OD Pairs Network (Incident Severity =1)**

| Program Execution Time (seconds) | Planning Horizon (seconds) | Incident Location | Incident Severity | Incident Duration (starting and End intervals) | Total Travel Time (seconds) |
|---|---|---|---|---|---|
| 121.20 | 180 | Link (2,5) | 1 | 4-10 | 920 |
| **Departure Time** | **Departure Node** | **Arrival Time** | **Arrival Node** | **OD Pair** | **Size** |
| 1 | 1 | 6 | 4 | 1 | 18.33 |
| 1 | 1 | 7 | 4 | 1 | 1.67 |
| 1 | 2 | 6 | 5 | 2 | 15.00 |
| 2 | 1 | 7 | 4 | 1 | 13.33 |
| 2 | 1 | 8 | 4 | 1 | 6.67 |
| 2 | 1 | 9 | 4 | 1 | 5.00 |
| **3** | **2** | **6** | **3** | **2** | **10.00** |
| 3 | 2 | 10 | 5 | 2 | 10.00 |
| **3** | **2** | **11** | **5** | **2** | **5.00** |
| 6 | 4 | 9 | 6 | 1 | 10.00 |
| 6 | 4 | 10 | 6 | 1 | 8.33 |
| 6 | 5 | 10 | 6 | 2 | 10.00 |
| 6 | 5 | 11 | 6 | 2 | 5.00 |
| **6** | **3** | **9** | **5** | **2** | **10.00** |
| 7 | 4 | 11 | 6 | 1 | 1.67 |
| 7 | 4 | 12 | 6 | 1 | 10.00 |
| 7 | 4 | 13 | 6 | 1 | 3.33 |
| 8 | 4 | 14 | 6 | 1 | 6.67 |
| 9 | 4 | 15 | 6 | 1 | 5.00 |
| **9** | **5** | **15** | **6** | **2** | **10.00** |
| 10 | 5 | 14 | 6 | 2 | 10.00 |
| **11** | **5** | **15** | **6** | **2** | **5.00** |

At the 1st time interval, vehicles took the shorter path for OD pair 2. At the 3rd time interval, there are two time-space paths whose travel times are equal. The travel time for the path (Link (2,5), Link (5,6)) is $(11-3)+(15-11)=12$ time intervals while the other path is $(6-3)+(9-6)+$

$(15 - 11) = 12$ time intervals. Therefore, the shorter path, which is the fully utilized path, has the travel time which is not higher than that of the partially utilized path.

By comparing the three scenarios of the multiple OD network, it is known that vehicles took alternative route when an incident occurred. Specifically in the multiple OD network, vehicles took the path that consists of Link (2,3), Link (3,5) and Link (5,6) when the capacity of the shorter path is undermined by an ongoing incident. Therefore, the flow on this longer path increased and thus its constituting links also exhibited flow increase. To take the idea further, Link (2,5) is substituted by Link (2,3) and (3,5) when incident occurred.

The next chapter examines the cases of different demand levels and the visualized results help to identify the link state relationships.

# Chapter 5 Examination of Link State Relationships

Link state relationships can be complicated especially when multiple origin-destination pairs exist due to interactions between links and paths under most circumstances. In this chapter, basic insights into link state relationships are provided based on density variations obtained from flow patterns of the test network with multiple OD pairs.

Two cases with different demand levels are tested. Density variation throughout the entire planning horizon is visualized by using a color-coded link map in which the commonly-adopted red-green color convention is applied to visualize the density. The link maps for incident and non-incident scenarios are compared to obtain information regarding link states under incident conditions while the link maps for different demand levels are also contrasted to acquire insights into the effect of OD pair interaction on link states.

The first section of this chapter presents the link maps for various demand levels as well as comparison results. The second section proposes possible application scenarios of link state relationships.

## 5.1. Density Variation under Different Demand Levels

First, the visualized density variation for the high-level demand under normal conditions is shown in Figure 7. Red indicates high density while low density is represented by green. For visual clarity purposes, the zero density is indicated by blue. Density can be visualized dynamically throughout the entire planning horizon but only four static frames of the temporal variation are shown for the analysis of link state relationships.

The link map of the no-incident scenario confirms that the vehicles traveling between OD pair 2, which is nodes 2 and 6, take both paths available for this OD pair. The path that consists of Link (2,3), Link (3,5) and Link (5,6) is taken due to congestion. The incident introduced for comparison occurs at time interval 4 with the duration of 6 time intervals. Its severity index is 0.6. The density variation for this incident scenario is presented in Figure 8. It is found that the identical path choice is exhibited in the incident scenario. Therefore, with this high-level demand, it is impossible to know whether there is path switching due to incident-related congestion.

In order to clearly show the impact of incident over drivers' route choice, the demand level for OD pair 2 is reduced. The demand profile is exhibited in Table 13. The incident introduced has the same properties as the previous scenario.

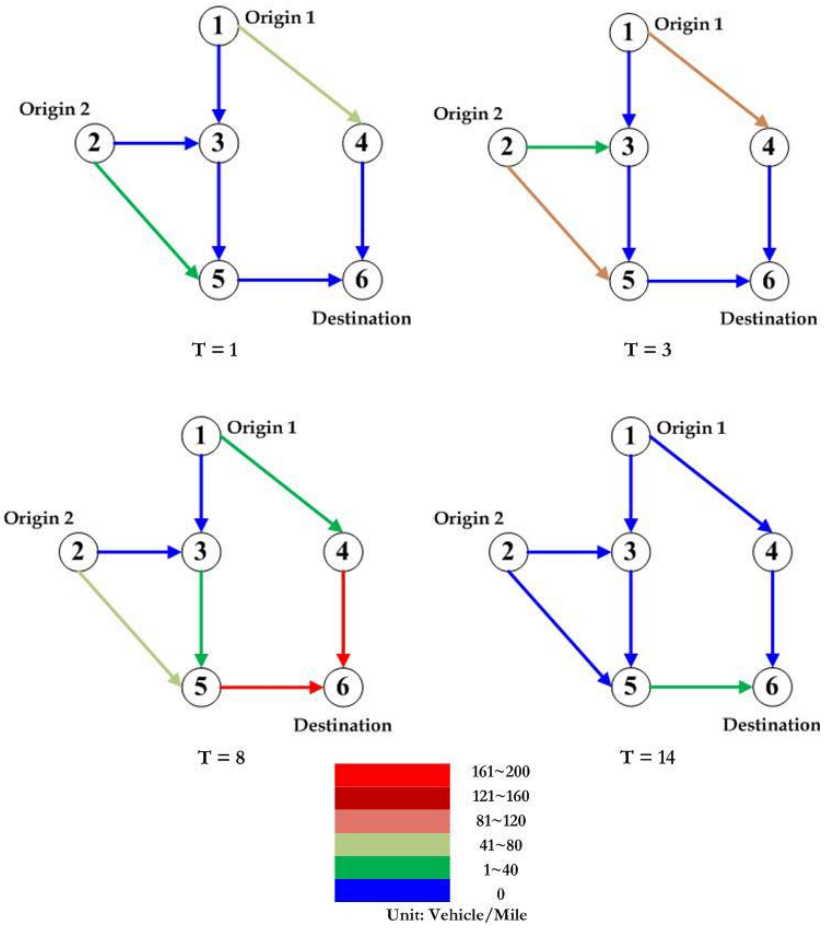


**Figure 8 Link Map for High-Level Demand of Non-Incident Scenario**

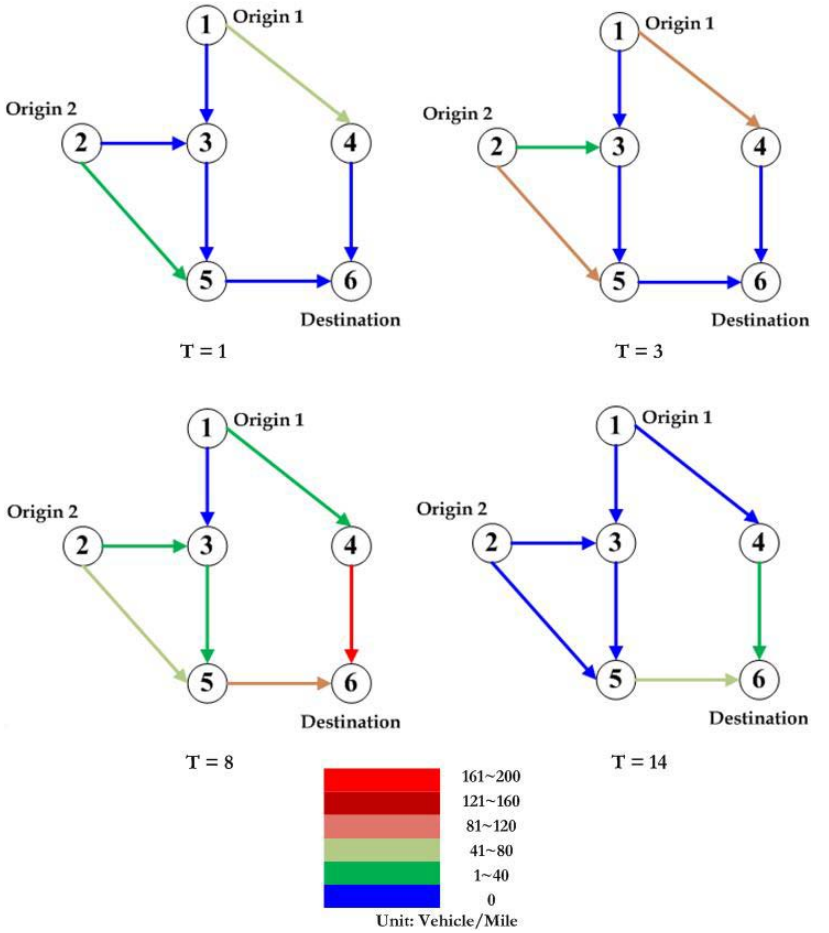


**Figure 9 Link Map for High-Level Demand of Incident Scenario**

**Table 13 Demand Profile for Low-Level Demand Scenario**

| Time Interval (sec) | 10 | | |
|---|---|---|---|
| **Generation Node** | **Generation Time (ith interval)** | **OD Pair** | **Number of Vehicles** |
| 1 | 1 | (1,6) | 20 |
| 1 | 2 | (1,6) | 25 |
| 2 | 1 | (2,6) | 8 |
| 2 | 3 | (2,6) | 10 |

The visualized density variation is depicted in Figure 9 and Figure 10. The test network has two routes for OD pair (2,6). The first route consists of links: (2,3), (3,5) and (5,6) and the second one consists of links: (2,5) and (5,6). It is expected that these two routes serve as substitutes to each other under incident conditions. In order to clearly examine the substitution relationship between the two routes, the possible maximum densities of all links were set to fall within the non-congested regime by reducing the demand for OD pair 2. In this way, we can isolate the effects of the incident from congestion effects. Note that the maximum density is less than 80 vehicles/mile, which indicates uncongested conditions. Under the no-incident scenario, all vehicles take the second route which consists of Link (2,5) and Link (5,6). When an incident occurred, the vehicles chose the parallel route. This different route choice behavior is consistent with the expectation of the relationship between the two routes.

For link state relationships, Link (2,3), Link (3,5) and Link (2,5) exhibit a substitution relationship under the incident conditions. More specifically, during the incident, Link (2,3) and Link (3,5) supply the capacity needed to handle the diverted traffic due to the capacity reduction of Link (2,5).

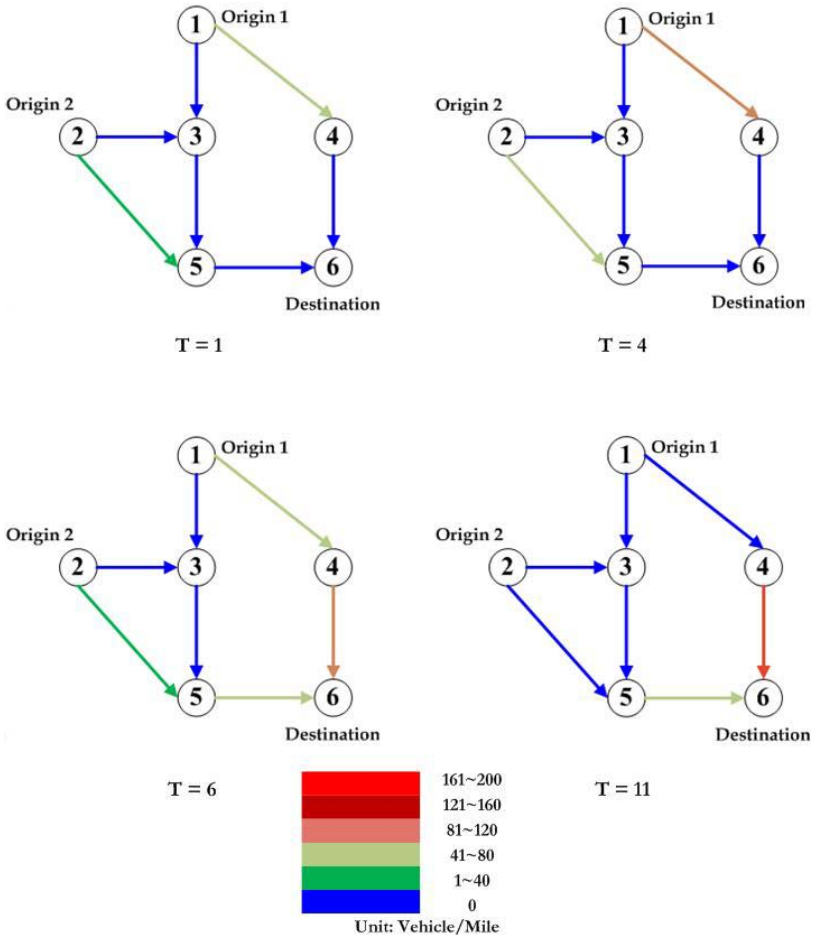


**Figure 10 Link Map for Low-Level Demand of Non-Incident Scenario**

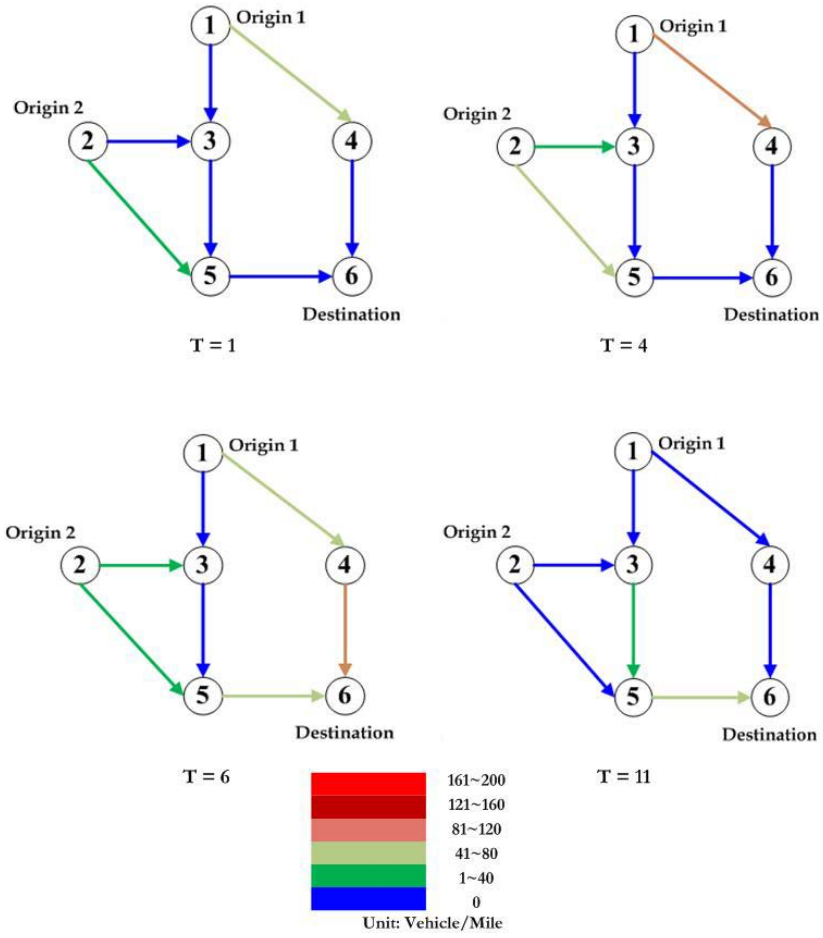


**Figure 11 Link Map for Low-Level Demand of Incident-Scenario**

## 5.2. Application of Link State Relationships

With the information regarding the link state relationships, effective incident management strategies can be determined accordingly. For example, the signal timing settings of the links that substitute for the link where incidents occur can be adapted. Moreover, the information of quantity and duration of the additional traffic can be used to determine whether it is necessary to exploit the dynamic HOV designation strategy and open the HOV lanes to all traffic to deal with the incident.

For the multiple-OD network in the real world, there are usually more than two paths for each OD pair and a few links are shared by many paths. Additionally, with the changing incident conditions and demand levels, the link states may not always exhibit identical relationships. However, the merits of predicting flow and density variations still remain and can be used for incident management applications. The visualized density and flow predictions under different incident scenarios can be aggregated and mined to extract a generalized pattern for the state relationships for certain links. These general patterns can then serve as guidance for incident management purposes for those links. Needless to say, the link state relationships for a sufficient number of incident scenarios need to be investigated and stored before operational suggestions for incident management can be made. For example, incidents of similar severity which occurred on the same link can be synthesized into a group. The visualized traffic variations for adjacent links can be analyzed to determine the impact of an incident of such kind. The spatial range and temporal duration of the impact can be investigated and this information is beneficial to incident management.

Besides the aggregate usage of the link state relationships aforementioned, flow and density predictions for individual links can be used as a reference for making informed incident management decisions for these specific links.

## 5.3. Recommendations for Near Real-time Implementation

The visualized density based on flow predictions can be used for online incident management purposes. Figure 12 depicts how they can be used.

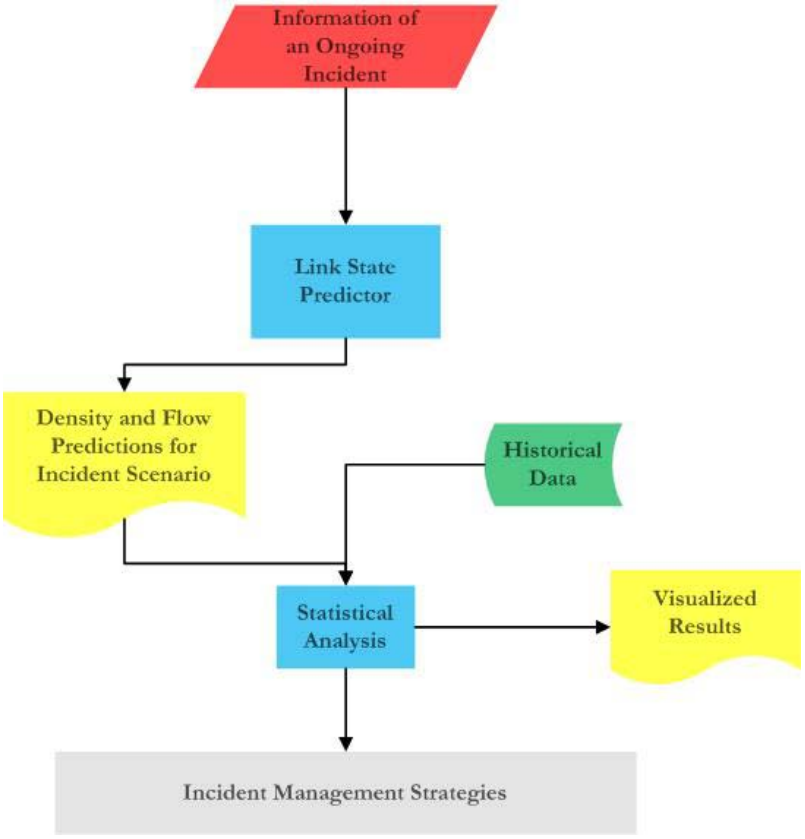


**Figure 12 The Process of Near Real-Time Applications**

The first two incident management activities are incident detection and verification (FHWA, 2000). When an incident occurs, the incident is reported to the agency or agencies responsible for maintaining traffic flow and safe operations on the facility in various ways such as mobile telephone calls, surveillance cameras, and police patrols (FHWA, 2000). Though, the accurate incident duration is hard to predict. The first responders arriving at the incident scene usually confirm that an incident has occurred, determine its exact location, and obtain as many relevant details about the incident as possible, which is called incident verification (FHWA, 2000). The information such as incident location and severity collected during the incident verification process can be used as inputs for the link state predictor.

With the incident information necessary, the link state predictor could be constructed and provide density and flow predictions. The historical flow and density data under similar OD demand should be used for the statistical comparison. The statistical analysis would identify the links that will experience significant additional traffic. The visualization of analysis results can be achieved through a procedure that is capable of real-time processing and visualization.

One candidate is ArcGIS Tracking Analyst, which is able to accept real-time data and more importantly project the visualized densities over the geographically referenced layer precisely (ESRI, 2009), which is crucial for visual clarity and easiness of interpretation. After the analysis is finished, corresponding incident management strategies can be applied.

# Chapter 6 Conclusions and Future Extensions

In this chapter, conclusions are drawn and directions for future research efforts are identified.

## 6.1. Conclusions

Effective implementation of incident congestion mitigation strategies requires accurate and efficient procedures to predict link states, which are described by traffic density and flow. Considering the dynamic routing behavior and the characteristics of link states, in this thesis, we applied UE-DTA to predict the link states and get insights regarding link state relationships. We developed a model that incorporates multiple origin destination pairs while possessing the capability of modeling transient incident phenomena. The model is based on the DTA framework proposed by Carey (1999, 2009) which allows user equilibrium by iterating between the routing module and the link flow module that constitute the whole model. The model's equivalence to DUE was proven by exploiting the Kuhn-Tucker condition, which is necessary and sufficient for a linear program.

Two cases, namely the single OD pair network and multiple OD network, were constructed and tested. The results showed that the flow pattern preserves the user equilibrium principle and satisfies the FIFO condition. The link-based encapsulation of Cell Transmission Model is able to temporally capture the spillback between links and fully mimics the spillback within links but not between links. This model strikes a balance between computational tractability, traffic realism and incident modeling capability.

The flow pattern resulting from the model can be easily transformed to maps of link density variations as shown in Section 5.1. The prediction of link density variations, combined with the flow pattern, was used to investigate the link state relationships. By isolating the effects of an incident, the parallel routes of a specific OD pair display the relationship of substituting for each other, which is consistent with the general expectation regarding such parallel routes. A closer examination of the density variations confirms the existence of a substitution relationship between the links of the two parallel routes.

Two levels of application of link state relationships were identified for real-world situations. Information about link states for different incident scenarios can be aggregated and mined to derive general patterns for the link state relationships. These patterns can be used as general guidance for

incident management purposes. A microscopic level of application involves usage of flow and density predictions for a specific incident. For example, they can be used to determine whether it is necessary to exploit the dynamic HOV designation strategy and open the HOV lane to all traffic to deal with the incident. Operational adjustments such as changing signal timing can also be made based on the information regarding link states.

The major contributions of this thesis are as follows. A new DTA model was proposed to investigate link state relationships. The model adopts a two-module scheme in which the routing module is a linear programming model and the link flow model applies the CTM approach. This new model is capable of handling multiple OD pairs and modeling capacity-reducing incidents, which is a major improvement to the DTA model by Carey (1999; 2009). Based on the output flow pattern, link density maps can be constructed to provide clear insights to dynamic routing behaviors and by comparing the link density maps, link state relationships can be revealed. In addition, the model can be adapted to near real-time applications and serve as a tool for incident mitigation and management.

## 6.2. Future Extensions

One of the most important extensions is to deal with spatial spillbacks between links. In the lower level CTM model, when diverge and merge cells are introduced, it is critical to develop a mechanism to determine the diverge coefficients while maintaining the FIFO condition. In addition, real-world situations including queuing at signalized intersections can be also incorporated into the model by adapting the lower level CTM model. More specifically, cell capacities can be set as time-dependent to simulate the traffic signals. With these extensions and adaption, the model can be applied to large-scale networks by refining the solution code or resorting to more advanced computing techniques. In addition, the resolution of visualization can be improved by visualizing cell densities instead of link densities.

Another avenue for future research is to tackle the challenges for real-time application of the proposed model. More specifically, it is important to explore how to integrate the information about link state relationships under different incident scenarios and thus derive the general pattern of link state relationships for statistical comparison. High computation speed for this derivation process is

desired since incident mitigation strategies need to be determined quickly. Therefore, research efforts are needed in designing efficient data mining algorithms for real-time deployment.

## Reference


Ben-Akiva, M., M. Bierlaire, et al. (1997). Development of a route guidance generation system for real-time application. Proceedings of the 8th IFAC/IFIP/IFORS. Transportation Systems 1997 (3 vol.), 16-18 June 1997, Oxford, UK, Pergamon.

Ben-Akiva, M. E., M. Bierlaire, et al. (1998). DynaMIT: A Simulation-based System for Traffic Prediction and Guidance Generation. The 3rd Triennial Symposium on Transportation Systems, San Juan, Puerto Rico.

Ben-Akiva, M. E., H. N. Koutsopoulos, et al. (1996). A simulation laboratory for testing traffic management systems. Proceedings of IASTED International Conference on Modelling and Simulation, 25-27 April 1996, Anaheim, CA, USA, IASTED-ACTA Press.

Bhosle, A. M. and T. F. Gonzalez (2003). Efficient Algorithms for Single Link Failure Recovery and Its Application to ATM Networks. Proceedings of the Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems, November 3, 2003 - November 5, 2003, Marina del Rey, CA, United states, Int. Assoc. of Science and Technology for Development.

Cambridge Systematics, I. (2005). "Traffic Congestion and Reliability: Trends and Advanced Strategies for Congestion Mitigation." Retrieved September 23, 2009, from http://ops.fhwa.dot.gov/congestion_report/.

Carey, M. (1999). A Framework for Dynamic Traffic Assignment. Northern Ireland, University of Ulster.

Carey, M. (2009). "A framework for user equilibrium dynamic traffic assignment." Journal of the Operational Research Society **60**: 395-410.

Carey, M. and E. Subrahmanian (2000). "An approach to modelling time-varying flows on congested networks." Transportation Research, Part B (Methodological) **34B**(3): 157-183.

Dafermos, S. (1980). "Traffic equilibrium and variational inequalities." Transportation Science **14**(Copyright 1980, IEE): 42-54.

Daganzo, C. F. (1994). "The cell transmission model: a dynamic representation of highway traffic consistent with the hydrodynamic theory." Transportation Research, Part B (Methodological) **28B**(4): 269-287.

Daganzo, C. F. (1995). "The cell transmission model. II. Network traffic." Transportation Research, Part B (Methodological) **29B**(2): 79-93.

Dong, J., H. S. Mahmassani, et al. (2006). "How Reliable Is This Route? Predictive Travel Time and Reliability for Anticipatory Traveler Information Systems." Transportation Research Record(1980): 117-125.

ESRI (2009). ArcGIS Desktop Help. Redlands, CA.

FHWA. (2000). "Traffic Incident Management Handbook." Retrieved July 23, 2010, 2010, from http://floridaapts.lctr.org/pdf/incident%20mgmt_handbook%20Nov00.pdf.

FHWA (2007). DYNASMART-P User's Guide v1.3.0. McLean, VA.

FHWA. (2009, August 24, 2009). "Priority, Market-Ready Technologies and Innovations List: DYNASMART-P." Retrieved June 10, 2010, 2010, from http://www.fhwa.dot.gov/crt/lifecycle/dynasmart.cfm.

Friesz, T. L., J. Luque, et al. (1989). "Dynamic network traffic assignment considered as a continuous time optimal control problem." Operations Research **37**(6): 893-901.

Ge, Y. E. and M. Carey (2004). Travel time computation of link and path flows and first-in-first-out, Dalian, China, Science Press.

Golani, H. and S. T. Waller (2004). "Combinatorial approach for multiple-destination user optimal dynamic traffic assignment." Transportation Research Record(1882): 70-78.

Helman, D. L. (2004). "Traffic Incident Management." Retrieved September 23, 2009, from http://www.tfhrc.gov/pubrds/04nov/03.htm.

Huey-Kuo, C. and H. Che-Fu (1998). "A model and an algorithm for the dynamic user-optimal route choice problem." Transportation Research, Part B (Methodological) **32B**(Copyright 1998, IEE): 219-234.

ITS-Program. (2010). "DynaMIT, How does it work?" Retrieved June 10, 2010, 2010, from http://mit.edu/its/dynamit.html.

Janson, B. N. (1991a). "Dynamic traffic assignment for urban road networks." Transportation Research, Part B (Methodological) **25B**: 143-161.

Janson, B. N. (1991b). "Convergent Algorithm for Dynamic Traffic Assignment." Transportation Research Record **1328**: 69-80.

Jayakrishnan, R., A. Chen, et al. (1999). "Freeway and arterial traffic flow simulation analytically embedded in dynamic assignment." Transportation Research Record(1678): 242-250.

Jayakrishnan, R., H. S. Mahmassani, et al. (1991). User-friendly Simulation Model for Traffic Networks with ATIS/ATMS. The 5th International Conference on Computing in Civil and Building Engineering. Anaheim, CA.

Jayakrishnan, R., H. S. Mahmassani, et al. (1994). "An evaluation tool for advanced traffic information and management systems in urban networks." Transportation Research Part C (Emerging Technologies) **2C**(Copyright 1994, IEE): 129-147.

Jayakrishnan, R., M. G. McNally, et al. (1993a). Simulation of ATIS strategies to mitigate special event congestion. Proceedings of the 5th International Conference on Computing in Civil and Building Engineering - V-ICCCBE, June 7, 1993 - June 9, 1993, Anaheim, CA, USA, Publ by ASCE.

Jayakrishnan, R., M. G. McNally, et al. (1993b). Simulation of Advanced Traveler Information Systems (ATIS) Strategies to Reduce Non-Recurring Congestion from Special Events. Irvine, CA, University of California, Irvine.

Jayakrishnan, R., W. K. Tsai, et al. (1995). "A dynamic traffic assignment model with traffic-flow relationships." Transportation Research Part C (Emerging Technologies) **3C**(1): 51-72.

Jiang Qian, Y. and T. Miyagi (2001). "Sensitivity analysis for stochastic user equilibrium network flows-a dual approach." Transportation Science **35**(Copyright 2001, IEE): 124-133.

Li, Y., A. K. Ziliaskopoulos, et al. (1999). "Linear programming formulations for system optimum dynamic traffic assignment with arrival time-based and departure time-based demands." Transportation Research Record(1667): 52-59.

Lighthill, M. J. and G. B. Whitham (1955). "On kinematic waves, Theory of traffic flow on long crowded roads." Proceedings of Royal Society of London **229**(1178): 317-345.

Liu, S. and P. Murray-Tuite (2008). Evaluation of Stategies to Increase Transportation Resilience to Congestion Caused by Incidents. Falls Church, Virginia, Virginia Polytechnic and State University.

Lo, H. K. and W. Y. Szeto (2002). "A cell-based dynamic traffic assignment model: Formulation and properties." Mathematical and Computer Modelling **35**(7-8): 849-865.

Mahmassani, H. S., J. Dong, et al. (2009). Incorporating Weather Impacts in Traffic Estimation and Prediction Systems. McLean, VA, SAIC**:** 108.

Mahmassani, H. S., T.-Y. Hu, et al. (1998). DYNASMART-X Volume II: Analytical and Algorithmic Aspects. Austin, TX, Center for Transportation Research, University of Texas at Austin. **2**.

Mahmassani, H. S., T.-Y. Hu, et al. (1992). DYNASMART: Dynamic Network Assignment-Simulation Model for Advanced Road Telematics. Austin, Texas, Center for Transportation Research, University of Texas at Austin.

Mctrans. (2010). "DYNASMART-P version 1.3.0." Retrieved June 10, , 2010, from http://mctrans.ce.ufl.edu/featured/dynasmart/.

Merchant, D. K. and G. L. Nemhauser (1978a). "A Model and an Algorithm for the Dynamic Traffic Assignment Problems." Transportation Science **12**(3): 183-199.

Merchant, D. K. and G. L. Nemhauser (1978b). "Optimality Conditions for a Dynamic Traffic Assignment Model." Transportation Science **12**(3): 200-207.

Peeta, S. and A. K. Ziliaskopoulos (2001). "Foundations of Dynamic Traffic Assignment: The past, the present and the future." Networks and Spatial Economics **1**(3-4): 233-265.

PTV (2005). VISSIM 4.10 User Manual. Karlsruhe, Germany.

Rakha, H., M. Hellinga, et al. (1996). INTEGRATION: An Overview of Current Simulation Features. the 75th Annual Meeting of Transportation Research Board Washington DC.

Rakha, H., M. Van Aerde, et al. (1989). Evaluating the Benefits and Interactions of Route Guidance and Traffic Control Strategies Using Simulation. First Vehicle Navigation and Information Systems Conference (VNIS). Piscataway NJ**:** 296-303.

Ran Bin and D. Boyce (1994). Dynamic Urban Transportation Network Models: Theory and Implications for Intelligent Vehicle Highway Systems. Berlin, Springer-Verlag.

Richards, P. I. (1956). "Shock waves on highway." Operations Research Society of America -- Journal **4**(1): 42-51.

Rilett, L. R. and M. Van Aerde (1991a). Modeling Distributed Real-Time Route Guidance Strategies in a Traffic Network that Exhibits the Braess Paradox. the 1991 Vehicle Navigation and Information Systems (VNIS) Conference, Dearborn, MI, Society of Automotive Engineers.

Rilett, L. R. and M. Van Aerde (1991b). "Routing Based On Anticipated Travel Times." Applications of Advanced Technologies in Transportation Engineering, Proceedings of the Second International Conference: 183-187.

Robinson, M. D. and P. M. Nowak (1993). "An Overview of Freeway Incident Management in the United States."

Sheffi, Y. (1985). Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods. Englewood Cliffs, New Jersey, Prentice-Hall.

Sisiopiku, V. P. and L. Xuping (2006). Overview of dynamic traffic assignment options. 2006 Spring Simulation Multiconference (SpringSim'06), 2-6 April 2006, San Diego, CA, USA, Society for Modeling and Simulation International.

Van Aerde & Assoc. (2005a). INTEGRATION RELEASE 2.30 FOR WINDOWS User's Guide-Volume I: Fundamental Model Features Kingston, Ontario.

Van Aerde & Assoc. (2005b). INTEGRATION RELEASE 2.30 FOR WINDOWS User's Guide-Volume II: Advanced Model Features Kingston, Ontario.

Van Aerde, M. and H. Rakha (1989). Development and Potential of System Optimized Route Guidance Strategies. First Vehicle Navigation and Information Systems Conference (VNIS). Piscataway NJ**:** 304-309.

Waller, S. T. and A. K. Ziliaskopoulos (2006a). "A combinatorial user optimal dynamic traffic assignment algorithm." Annals of Operations Research **144**: 249-261.

Waller, S. T. and A. K. Ziliaskopoulos (2006b). "A chance-constrained based stochastic dynamic traffic assignment model: Analysis, formulation and solution algorithms." Transportation Research Part C (Emerging Technologies) **14**(6): 418-427.

Ziliaskopoulos, A. K. (2000). "A linear programming model for the single destination system optimum dynamic traffic assignment problem." Transportation Science **34**(1): 37-49.

Ziliaskopoulos, A. K. and S. T. Waller (2000). "An Internet-based geographic information system that integrates data, models and users for transportation applications." Transportation Research Part C (Emerging Technologies) **8C**(Copyright 2000, IEE): 427-444.

Ziliaskopoulos, A. K., S. T. Waller, et al. (2004). "Large-scale dynamic traffic assignment: Implementation issues and computational analysis." Journal of Transportation Engineering **130**(Compendex): 585-593.

## Appendix

The solution code is provided here. There are 5 different modules, namely class definition, function definition, main program, link flow model and output module.

Class Definition

```
using System;
using System.Collections.Generic;
using System.Text;

namespace LP_DTA_MOD
{
    public class Network
    {
        public int numlinks;
        public int NumNodes;
        public Link[] linklist;
        public int[,] AdjacencyMatrix;
        public Network()
        {
            numlinks = 1; //default number of links is 1
            NumNodes = 2;
            linklist = new Link[numlinks];
            AdjacencyMatrix = new int[NumNodes, NumNodes];
        }
        public Network(int _numlinks, int _NumNodes)
        {
            numlinks = _numlinks;
            NumNodes = _NumNodes;
            linklist = new Link[numlinks];
            AdjacencyMatrix = new int[NumNodes, NumNodes];
        }
        public void ConstructAdjMatrix()
        {
            for (int i = 0; i < NumNodes; i++)
            {
                for (int j = 0; j < NumNodes; j++)
                {
                    AdjacencyMatrix[i, j] = 0;
                }
            }
            for (int k = 0; k < numlinks; k++)
            {
                int RowIndex = (int)linklist[k].upnode - 1;
                int ColumnIndex = (int)linklist[k].downnode - 1;
                AdjacencyMatrix[RowIndex, ColumnIndex] = 1;
            }
        }
    }
    public class Link : ICloneable
```

```
    {
        public double linkno;
        public double upnode;
        public double downnode;
        public double frespd;
        public double jamden;
        public double maxflow;
        public double nolanes;
        public double linklength;
        public int numcell; //calculated
        public Cell[] celllist;
        public Link()
        {
            linkno = 0;
            upnode = 0;
            downnode = 0;
            frespd = 0;
            jamden = 0;
            maxflow = 0;
            linklength = 0;
        }
        public Link(double[] link_data)
        {
            linkno = link_data[0];
            upnode = link_data[1];
            downnode = link_data[2];
            frespd = link_data[3];
            jamden = link_data[4];
            maxflow = link_data[5];
            nolanes = link_data[6];
            linklength = link_data[7];
        }
        public object Clone()
        {
            return this.MemberwiseClone();
        }
        public void ini_celllist(double clock_interval) //initialize
cell list for each link
        {
            double cell_length = frespd * clock_interval / 3600;
            double cell_capacity = maxflow * clock_interval / 3600 *
nolanes;
            double max_occupancy = cell_length * jamden;
            numcell = (int)System.Math.Ceiling(linklength /
cell_length) + 2;//add a dummy source and dummy sink
            celllist = new Cell[numcell];
            for (int i = 0; i < numcell; i++)
            {
                celllist[i] = new Cell(cell_capacity, max_occupancy);
            }
            for (int i = 0; i < celllist.Length; i++) //Assign cell
type to each cell, cell is a source by default
```

```
            {
                if (i == celllist.Length - 1)   //The last cell is a
sink
                {
                    celllist[i].type = 3;
                }
                else
                    if (i != 0)
                    {
                        {
                            celllist[i].type = 2;
                        }
                    }
                    else
                    {
                        celllist[i].type = 1;
                    }
            }
        }

    }
    public class Cell
    {
        public double occ;
        public double outflow;  //outflow for the current time
interval
        public int type;
        public double cap;
        public double default_cap; //capacity under normal
circumstances
        public double max_occ;
        public List<double> occ_record;
        public List<double> flow_record;
        public List<Packet> packetlist;
        public Cell()
        {
            packetlist = new List<Packet>();
            occ_record = new List<double>();
            flow_record = new List<double>();
            occ = 0;
            outflow = 0;
            type = 1;
            cap = 0;
            default_cap = 0;
            max_occ = 0;
            occ_record.Add(0); //occupancy and flow for all cells at
time 0 is 0
        }
        public Cell(double _cap, double _max_occ)
        {
            default_cap = _cap;
            cap = _cap;
```

```
            max_occ = _max_occ;
            packetlist = new List<Packet>();
            occ_record = new List<double>();
            flow_record = new List<double>();
            occ_record.Add(0);
            occ = 0;
            outflow = 0;
        }
    }
    public class NodePair : ICloneable, IComparable
    {
        public double upnode_no;
        public double downnode_no;
        public NodePair() { }
        public NodePair(double _unodeno, double _dnodeno)
        {
            upnode_no = _unodeno;
            downnode_no = _dnodeno;
        }
        public object Clone()
        {
            return MemberwiseClone();
        }
        public int CompareTo(object rightobject)
        {
            NodePair leftpair = this;
            NodePair rightpair = (NodePair)rightobject;
            if (leftpair.upnode_no < rightpair.upnode_no)
            {
                return -1;
            }
            if (leftpair.upnode_no > rightpair.upnode_no)
            {
                return 1;
            }
            if (leftpair.upnode_no == rightpair.upnode_no)
            {
                if (leftpair.downnode_no < rightpair.downnode_no)
                {
                    return -1;
                }
                if (leftpair.downnode_no > rightpair.downnode_no)
                {
                    return 1;
                }
                return 0;
            }
            return 0;
        }
    }
    public class Packet : ICloneable
    {
```

```
        public double size = 0;
        public double dest = 1;
        public double entry_time; //record this packet's time of entry
to a cell
        public double gen_time; //time of generation in the source
cell
        public Packet copy_packet(Packet packet_to_copy)
        {
            Packet temp_packet = new Packet();
            temp_packet.dest = packet_to_copy.dest;
            temp_packet.entry_time = packet_to_copy.entry_time;
            temp_packet.size = packet_to_copy.size;
            temp_packet.gen_time = packet_to_copy.gen_time;
            return temp_packet;
        }
        public Packet(Demand _current_demand) //generate a packet
according current demand
        {
            gen_time = _current_demand.demand_gentime;
            dest = _current_demand.demand_dest;
            size = _current_demand.demand_size;
            entry_time = _current_demand.demand_gentime + 1; //demand
is loaded one time interval later
        }
        public Packet() //default constructor
        {
            size = 0;
            dest = 1;
            entry_time = 0;
            gen_time = 0;
        }
        public object Clone()
        {
            return this.MemberwiseClone();
        }
    }
    public class Demand
    {
        public double demand_genlinkno;
        public double demand_gentime;
        public double demand_dest;
        public double demand_size;

        public Demand(double[] demand_data)
        {
            demand_genlinkno = demand_data[0];
            demand_gentime = demand_data[1];
            demand_dest = demand_data[2];
            demand_size = demand_data[3];
        }  //create demand by using external input
        public Demand()
        {
```

```
            demand_genlinkno = 0;
            demand_gentime = 0;
            demand_dest = 0;
            demand_size = 0;
        }
        public Demand(Flow flow_to_demand)
        {
            demand_genlinkno = 0;
            demand_gentime = flow_to_demand.in_time;
            demand_dest = flow_to_demand.dest;
            demand_size = flow_to_demand.size;
        }
        public Demand(double _genlinkno, double _gentime, double _dest,
double _size)
        {
            demand_genlinkno = _genlinkno;
            demand_gentime = _gentime;
            demand_dest = _dest;
            demand_size = _size;
        }
    }
    public class Flow : IComparable, ICloneable
    {
        public double in_time;
        public double up_node = 1;
        public double out_time;
        public double down_node = 2;
        public double dest;
        public double size;
        public Flow(double _in_time, double _up_node, double _out_time,
double _down_node, double _dest, double _size)
        {
            in_time = _in_time;
            up_node = _up_node;
            out_time = _out_time;
            down_node = _down_node;
            dest = _dest;
            size = _size;
        }
        public Flow()
        {
        }
        public Flow(Link _link, Packet _packet)
        {
            in_time = _packet.gen_time;
            up_node = _link.upnode;
            out_time = _packet.entry_time;
            down_node = _link.downnode;
            dest = _packet.dest;
            size = _packet.size;
        }
        public int CompareTo(object rightObject)
```

```
{
    Flow leftflow = this;
    Flow rightflow = (Flow)rightObject;
    if (rightflow.in_time < leftflow.in_time)
    {
        return 1;
    }
    if (rightflow.in_time > leftflow.in_time)
    {
        return -1;
    }
    if (rightflow.in_time == leftflow.in_time)
    {
        if (rightflow.up_node < leftflow.up_node)
        {
            return 1;
        }
        if (rightflow.up_node > leftflow.up_node)
        {
            return -1;
        }
        if (rightflow.up_node == leftflow.up_node)
        {
            if (rightflow.out_time < leftflow.out_time)
            {
                return 1;
            }
            if (rightflow.out_time > leftflow.out_time)
            {
                return -1;
            }
            if (rightflow.out_time == leftflow.out_time)
            {
                if (rightflow.down_node < leftflow.down_node)
                {
                    return 1;
                }
                if (rightflow.down_node > leftflow.down_node)
                {
                    return -1;
                }
                if (rightflow.down_node == leftflow.down_node)
                {
                    if (rightflow.dest < leftflow.dest)
                    {
                        return 1;
                    }
                    if (rightflow.dest > leftflow.dest)
                    {
                        return -1;
                    }
                }
```

```
                    return 0;
                }
                return 0;
            }
            return 0;
        }
        return 0;
    }
    public object Clone()
    {
        return this.MemberwiseClone();
    }

}
public class LingoDemand : ICloneable
{
    public double Ldemandno;
    public double Ldemand_gentime;
    public double Ldemand_snode;
    public double Ldemand_enode;
    public double Ldemand_dest;
    public double Ldemand_size;
    public LingoDemand() { }
    public LingoDemand(double[] LingoDemandData)
    {
        Ldemandno = LingoDemandData[0];
        Ldemand_gentime = LingoDemandData[1];
        Ldemand_snode = LingoDemandData[2];
        Ldemand_enode = LingoDemandData[3];
        Ldemand_dest = 0;
        Ldemand_size = LingoDemandData[4];
    }
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
public class Incident : ICloneable
{
    public double incidentno;
    public double linkno;
    public double cellno;
    public double starttime;
    public double endtime;
    public double severity;
    public Incident() { }
    public Incident(double[] IncidentData)
    {
        incidentno = IncidentData[0];
        linkno = IncidentData[1];
        cellno = IncidentData[2];
        starttime = IncidentData[3];
```

```
            endtime = IncidentData[4];
            severity = IncidentData[5];
        }
        public object Clone()
        {
            return this.MemberwiseClone();
        }

    }
}
```

Function Definition

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace LP_DTA_MOD
{
    class FunctionDef
    {
        public struct TNodePair : ICloneable
        {
            public double upnode;
            public double downnode;
            public double intime;
            public object Clone()
            {
                return MemberwiseClone();
            }
        }
        ////////////////////////////////Functions for Initialization
Procedure//////////////////////////////
        //Function to initialize the link list for the whole network
        public static void GetNetworkDim(string linkdatafile, ref int
numlinks, ref int numnodes)
        {
            StreamReader sr = new StreamReader(linkdatafile);

            string[] string_link = new string[8];
            String line;
            char[] charSeparators = new char[] { ',' };
            List<double> NodeStorage = new List<double>();

            while ((line = sr.ReadLine()) != null)
            {
                numlinks++;
                string_link = line.Split(charSeparators, 8,
StringSplitOptions.None);
```

```
                double[] temp_linkdata = new double[8];
                for (int j = 0; j < 8; j++)
                {
                    temp_linkdata[j] = Double.Parse(string_link[j]);
                }
                NodeStorage.Add(temp_linkdata[1]);
                NodeStorage.Add(temp_linkdata[2]);
            }
            NodeStorage.Sort();
            numnodes =
System.Convert.ToInt32(NodeStorage[NodeStorage.Count - 1]);
        }

        public static Link[] ini_linklist(string linkdatafile, Link[]
_linklist)
        {
            StreamReader sr = new StreamReader(linkdatafile);

            String line;

            string[] string_link = new string[8]; //store the demand
data in string format

            char[] charSeparators = new char[] { ',' };
            int k = 0;
            while ((line = sr.ReadLine()) != null)
            {
                string_link = line.Split(charSeparators, 8,
StringSplitOptions.None);
                double[] temp_linkdata = new double[8];
                for (int j = 0; j < 8; j++)
                {
                    temp_linkdata[j] = Double.Parse(string_link[j]);

                }
                Link temp_link = new Link(temp_linkdata);
                _linklist[k] = (Link)temp_link.Clone();
                k++;
            }
            return _linklist;
        }

        //create one-dimension node predecessor and successor array
        public static void CreateNodeAdjArrays(double[] NodePred,
double[] NodeSucc, int[,] AdjMatrix)
        {
            for (int k = 0; k < NodePred.Length; k++)
            {
                NodePred[k] = -1;
                NodeSucc[k] = -1;
            }
            int MatrixDimension = AdjMatrix.GetLength(0);
```

```
            for (int i = 0; i < MatrixDimension; i++)
            {
                for (int j = 0; j < MatrixDimension; j++)
                {
                    if (AdjMatrix[i, j] == 1)
                    {
                        if (NodeSucc[i] == -1)
                        {
                            NodeSucc[i] = j + 1;
                        }
                        if (NodePred[j] == -1)
                        {
                            NodePred[j] = i + 1;
                        }
                    }
                }
            }

            for (int k = 0; k < NodePred.Length; k++)
            {
                if (NodePred[k] == -1)
                {
                    NodePred[k] = 0;
                }
                if (NodeSucc[k] == -1)
                {
                    NodeSucc[k] = 0;
                }
            }

        }

        //Create Node pair list to look up for link no
        public static List<NodePair>
ConvertAdjMatrixNodePairList(int[,] AdjMatrix)
        {
            List<NodePair> NodeList = new List<NodePair>();
            for (int i = 0; i < AdjMatrix.GetLength(0); i++)
            {
                for (int j = 0; j < AdjMatrix.GetLength(1); j++)
                {
                    if (AdjMatrix[i, j] == 1)
                    {
                        NodePair np = new NodePair(i + 1, j + 1);
                        NodeList.Add(np);
                    }
                }
            }
            return NodeList;
        }
```

```
        //Create Lingo Demand List and convert it to one-dimensional
array
        public static List<LingoDemand> CreateLingoDemandList(string
lingodemandfile)
        {
            StreamReader sr = new StreamReader(lingodemandfile);
            String line;
            string[] string_link = new string[5]; //store the demand
data in string format
            char[] charSeparators = new char[] { ',' };
            int k = 0;
            List<LingoDemand> LingoDemandList = new
List<LingoDemand>();
            while ((line = sr.ReadLine()) != null)
            {
                string_link = line.Split(charSeparators, 5,
StringSplitOptions.None);
                double[] temp_Ldemanddata = new double[5];
                for (int j = 0; j < 5; j++)
                {
                    temp_Ldemanddata[j] = Double.Parse(string_link[j]);

                }
                LingoDemand temp_Ldemand = new
LingoDemand(temp_Ldemanddata);
                LingoDemand temp_Ldemand_2 =
(LingoDemand)temp_Ldemand.Clone();
                LingoDemandList.Add(temp_Ldemand_2);
                k++;
            }

            //Update dest field for the LingoDemandList
            List<NodePair> DestNodeList = new List<NodePair>();
            NodePair dnp1 = new
NodePair(LingoDemandList[0].Ldemand_snode,
LingoDemandList[0].Ldemand_enode);
            DestNodeList.Add(dnp1);
            for (int i = 1; i < LingoDemandList.Count; i++)
            {
                int flag = 1;
                for (int j = 0; j < DestNodeList.Count; j++)
                {
                    if (LingoDemandList[i].Ldemand_snode ==
DestNodeList[j].upnode_no &&
LingoDemandList[i].Ldemand_enode==DestNodeList[j].downnode_no)
                    {
                        flag = flag * 0;
                        break;
                    }
                }
                if (flag == 1)
                {
```

```
                    NodePair dnp2 = new
NodePair(LingoDemandList[i].Ldemand_snode,
LingoDemandList[i].Ldemand_enode);
                    DestNodeList.Add((NodePair)dnp2.Clone());
                }
            }
            DestNodeList.Sort();
            for (int i = 0; i < LingoDemandList.Count; i++)
            {
                for (int j = 0; j < DestNodeList.Count; j++)
                {
                    if
(LingoDemandList[i].Ldemand_snode==DestNodeList[j].upnode_no &&
LingoDemandList[i].Ldemand_enode==DestNodeList[j].downnode_no)
                    {
                        LingoDemandList[i].Ldemand_dest =
System.Convert.ToDouble(j + 1);
                    }
                }
            }
            return LingoDemandList;
        }
        public static double[]
CreateArrayLingoDemandSize(List<LingoDemand> LingoDemandList, ref
double demand_horizon, ref double no_dest, int numnodes)
        {
            int LDemandTotalEntry = LingoDemandList.Count;
            for (int i = 0; i < LingoDemandList.Count; i++)
            {
                if (LingoDemandList[i].Ldemand_gentime >
demand_horizon)
                {
                    demand_horizon =
LingoDemandList[i].Ldemand_gentime;
                }
            }
            List<double> TempDestList = new List<double>();
            TempDestList.Add(LingoDemandList[0].Ldemand_dest);
            for (int i = 1; i < LingoDemandList.Count; i++)
            {
                int flag = 1;
                for (int j = 0; j < TempDestList.Count; j++)
                {
                    if (LingoDemandList[i].Ldemand_dest ==
TempDestList[j])
                    {
                        flag = flag * 0;
                        break;
                    }
                }
                if (flag != 0)
                {
```

```
                    TempDestList.Add(LingoDemandList[i].Ldemand_dest);
                }
            }
            no_dest = System.Convert.ToDouble(TempDestList.Count);
            double[] LingoDemandSizeArray = new double[numnodes *
System.Convert.ToInt32(demand_horizon) * TempDestList.Count];
            for (int i = 0; i < LDemandTotalEntry; i++)
            {
                int index_1 =
System.Convert.ToInt32(LingoDemandList[i].Ldemand_gentime);
                int index_2 =
System.Convert.ToInt32(LingoDemandList[i].Ldemand_snode);
                int index_3 =
System.Convert.ToInt32(LingoDemandList[i].Ldemand_dest);
                LingoDemandSizeArray[(index_1 - 1) * (numnodes *
TempDestList.Count) + (index_2 - 1) * TempDestList.Count + index_3 - 1]
= LingoDemandList[i].Ldemand_size;
            }
            return LingoDemandSizeArray;
        }
        public static void AnalyzeLingoDemand(List<LingoDemand>
LingoDemandList, ref int demand_horizon, ref int no_dest)
        {
            double Tdemand_horizon = 0;
            Tdemand_horizon = LingoDemandList[0].Ldemand_gentime;
            for (int i = 1; i < LingoDemandList.Count; i++)
            {

                if (LingoDemandList[i].Ldemand_gentime >
demand_horizon)
                {
                    Tdemand_horizon =
LingoDemandList[i].Ldemand_gentime;
                }
            }
            demand_horizon = System.Convert.ToInt32(Tdemand_horizon);
            List<double> destList = new List<double>();
            destList.Add(LingoDemandList[0].Ldemand_dest);
            for (int i = 1; i < LingoDemandList.Count; i++)
            {
                int flag = 1;
                for (int j = 0; j < destList.Count; j++)
                {
                    if (LingoDemandList[i].Ldemand_dest == destList[j])
                    {
                        flag = flag * 0;
                        break;
                    }
                }
                if (flag == 1)
                {
                    destList.Add(LingoDemandList[i].Ldemand_dest);
```

```
                }
            }
            no_dest = destList.Count;
        }
        public static void GetLingoNodeArrays(List<LingoDemand>
LingoDemandList, double[] LStartNode, double[] LEndNode)
        {
            List<NodePair> DestNodeList = new List<NodePair>();
            NodePair dnp1 = new
NodePair(LingoDemandList[0].Ldemand_snode,
LingoDemandList[0].Ldemand_enode);
            DestNodeList.Add(dnp1);
            for (int i = 1; i < LingoDemandList.Count; i++)
            {
                int flag = 1;
                for (int j = 0; j < DestNodeList.Count; j++)
                {
                    if (LingoDemandList[i].Ldemand_snode ==
DestNodeList[j].upnode_no && LingoDemandList[i].Ldemand_enode ==
DestNodeList[j].downnode_no)
                    {
                        flag = flag * 0;
                        break;
                    }
                }
                if (flag == 1)
                {
                    NodePair dnp2 = new
NodePair(LingoDemandList[i].Ldemand_snode,
LingoDemandList[i].Ldemand_enode);
                    DestNodeList.Add((NodePair)dnp2.Clone());
                }
            }
            DestNodeList.Sort();

            //Update two array for LingoDemandList
            for (int i = 0; i < DestNodeList.Count; i++)
            {
                LStartNode[i] = DestNodeList[i].upnode_no;
                LEndNode[i] = DestNodeList[i].downnode_no;
            }
        }

        //Create Incident List
        public static void CreateIncidentList(string incidentfile,
List<Incident> IncidentList)
        {
            if (File.Exists(incidentfile))
            {
                StreamReader sr = new StreamReader(incidentfile);
                String line;
```

```
                string[] string_link = new string[6]; //store the
demand data in string format
                char[] charSeparators = new char[] { ',' };
                int k = 0;
                while ((line = sr.ReadLine()) != null)
                {
                    string_link = line.Split(charSeparators, 6,
StringSplitOptions.None);
                    double[] temp_Idemanddata = new double[6];
                    for (int j = 0; j < 6; j++)
                    {
                        temp_Idemanddata[j] =
Double.Parse(string_link[j]);
                    }
                    Incident temp_incident = new
Incident(temp_Idemanddata);
                    Incident temp_incident_2 =
(Incident)temp_incident.Clone();
                    IncidentList.Add(temp_incident_2);
                    k++;
                }
            }
        }

        //Determine whether to use the reduced capacity for the
current time interval
        public static bool DetermineCellIncident(int CurrentTime,
double linkno, double cellno, List<Incident> IncidentList)
        {
            bool boolReturnValue = true;
            for (int i = 0; i < IncidentList.Count; i++)
            {
                if (CurrentTime >= IncidentList[i].starttime &&
CurrentTime <= IncidentList[i].endtime && linkno ==
IncidentList[i].linkno && cellno == IncidentList[i].cellno)
                {
                    boolReturnValue = true;
                    break;
                }
                else
                    boolReturnValue = false;
            }
            return boolReturnValue;
        }
        public static double DetermineSeverity(int CurrentTime, double
linkno, double cellno, List<Incident> IncidentList)
        {
            int IncidentListIndex = 0;
            for (int i = 0; i < IncidentList.Count; i++)
            {
                if (DetermineCellIncident(CurrentTime, linkno, cellno,
IncidentList))
```

```
                {
                    IncidentListIndex = i;
                    break;
                }
            }
            return IncidentList[IncidentListIndex].severity;
        }
        public static double[] DetermineAllCellCapacity(Link link, int
CurrentTime, double linkno, List<Incident> IncidentList)
        {
            double[] CellCapArray = new double[link.celllist.Length];
            if (IncidentList.Count == 0)
            {
                for (int i = 0; i < link.celllist.Length; i++)
                {
                    CellCapArray[i] = link.celllist[i].default_cap;
                }
            }
            else
            {
                for (int i = 0; i < link.celllist.Length; i++)
                {
                    double severity = 0;
                    if (DetermineCellIncident(CurrentTime, linkno, i +
1, IncidentList))
                    {
                        severity = DetermineSeverity(CurrentTime,
linkno, i + 1, IncidentList);
                        CellCapArray[i] = (1 - severity) *
link.celllist[i].default_cap;
                    }
                    else
                    {
                        CellCapArray[i] = link.celllist[i].default_cap;
                    }
                }
            }
            return CellCapArray;
        }



        ////////////////////////////Functions for Link Sub
Model//////////////////////////////////
        public static List<Demand> GetLinkDemand(List<Demand>
SubModelDemandList, double linkno)
        {
            List<Demand> _demandlist = new List<Demand>();
            for (int i = 0; i < SubModelDemandList.Count; i++)
//identify the demand for a specific link
            {
                if (SubModelDemandList[i].demand_genlinkno == linkno)
```

```
                {
                    _demandlist.Add(SubModelDemandList[i]);
                }
            }
            return _demandlist;
        }

        public static List<int> Get_currentdemand_index(int
current_time, List<Demand> demand_list)
        {
            List<int> demand_index_currentinterval = new List<int>();
            for (int i = 0; i < demand_list.Count; i++)
            {
                if (demand_list[i].demand_gentime == current_time)
                {
                    demand_index_currentinterval.Add(i);
                }

            }
            return demand_index_currentinterval;
        }

        public static Packet[] ListToArray(List<Packet> _packetlist)
        {
            int array_dim = _packetlist.Count;
            Packet[] packet_array = new Packet[array_dim];
            for (int i = 0; i < array_dim; i++)
            {
                packet_array[i] = (Packet)_packetlist[i].Clone();
            }
            return packet_array;
        }


        public static List<Packet> ListManipulate(List<Packet>
_packetlist)
        {
            int packet_no = _packetlist.Count;
            int[] packetlist_flag = new int[packet_no];
            Packet[] current_packets = new Packet[packet_no];
            current_packets = ListToArray(_packetlist);
            for (int j = 0; j < current_packets.Length; j++) //first
iteration finds the packets that should be deleted
            {
                double temp_entrytime = current_packets[j].entry_time;
                double temp_dest = current_packets[j].dest;
                double temp_gentime = current_packets[j].gen_time;
                for (int k = j + 1; k < current_packets.Length; k++)
                {
                    if (current_packets[k].dest == temp_dest &&
current_packets[k].entry_time == temp_entrytime &&
current_packets[k].gen_time == temp_gentime)
```

```
                    {
                        current_packets[j].size +=
current_packets[k].size;
                        packetlist_flag[k] = 1;
                    }
                }
            }

            _packetlist.Clear();

            for (int i = 0; i < current_packets.Length; i++)
            {
                if (packetlist_flag[i] != 1)
                {
                    _packetlist.Add(current_packets[i]);
                }

            }

            return _packetlist;
        }

        public static double[] GetAllEntryTime(List<Packet>
_packetlist)  //find all the entry time; checked for reference passing
        {
            List<double> all_entrytime = new List<double>();
            int packet_index = 0;
            double temp_entrytime =
_packetlist[packet_index].entry_time;
            all_entrytime.Add(temp_entrytime);
            while (packet_index != _packetlist.Count - 1)
            {
                packet_index++;
                if (temp_entrytime !=
_packetlist[packet_index].entry_time)
                {
                    temp_entrytime =
_packetlist[packet_index].entry_time;
                    all_entrytime.Add(temp_entrytime);
                }
            }
            int array_dimension = all_entrytime.Count;
            double[] set_entrytime = new double[array_dimension];
            all_entrytime.CopyTo(set_entrytime);
            return set_entrytime;
        }

        public static double[] AggPacketSize(List<Packet> _packetlist)
        {
            double[] all_entry = GetAllEntryTime(_packetlist);
//identify all the entry times in current packetlist
            int agg_packet_dim = all_entry.Length;
```

```
            int dis_packetlist_dim = _packetlist.Count;
            double[] agg_packet = new double[agg_packet_dim];
            for (int i = 0; i < agg_packet_dim; i++)
            {
                for (int j = 0; j < dis_packetlist_dim; j++)
                {
                    if (all_entry[i] == _packetlist[j].entry_time)
                    {
                        agg_packet[i] += _packetlist[j].size;
                    }
                }
            }
            return agg_packet;
        } //aggregate packet list based on entry time

        public static int GetNumPacketsLeave(double latest_entry_time,
Packet[] current_packets)
        {
            int num_packets_leave = 1;
            int packet_iter = 0;
            while (packet_iter != current_packets.Length - 1)
            {
                packet_iter++;
                if (current_packets[packet_iter].entry_time <=
latest_entry_time)
                {
                    num_packets_leave++;
                }
            }
            return num_packets_leave;
        }

        public static int get_numODpairs(Packet[] _packetarray) //Used
in another function
        {
            List<double> ODpairs = new List<double>();
            ODpairs.Add(_packetarray[0].dest);
            for (int i = 1; i < _packetarray.Length; i++)
            {
                if (!ODpairs.Contains(_packetarray[i].dest))
                {
                    ODpairs.Add(_packetarray[i].dest);
                }
            }
            return ODpairs.Count;
        }

        public static double[] GetPartialPacketSize(double
partial_outflow, Packet[] _packetarray)
        {
            int OD_no = get_numODpairs(_packetarray);
            int GenTime_no = 0;
```

```
            List<double> GenTimes = new List<double>();
            GenTimes.Add(_packetarray[0].gen_time);
            for (int i = 1; i < _packetarray.Length; i++)
            {
                if (!GenTimes.Contains(_packetarray[i].gen_time))
                {
                    GenTimes.Add(_packetarray[i].gen_time);
                }
            }
            GenTime_no = GenTimes.Count;

            int size_partialpacketDim = 0;
            if (OD_no == _packetarray.Length)
            {
                size_partialpacketDim = OD_no;
            }
            else
            {
                size_partialpacketDim = GenTime_no;
            }
            double total_size = 0;
            double[] size_partialpacket = new
double[size_partialpacketDim];
            for (int i = 0; i < _packetarray.Length; i++)
            {
                total_size += _packetarray[i].size;
            }
            for (int i = 0; i < size_partialpacketDim; i++)
            {
                size_partialpacket[i] = _packetarray[i].size *
partial_outflow / total_size;
            }
            return size_partialpacket;

        }



        ////////////////////////////Functions for Main Calculation
Routine/////////////////////////
        public static bool StopRuleCheck(List<double[]> ListdFlowArray,
int CurrentIteration, double StopThreshold)
        {
            if (CurrentIteration <= 2)
            {
                return false;                    //false indicates
iteration should go on
            }
            else
            {
```

```
                double[] CurrentFlowArray = new
double[ListdFlowArray[CurrentIteration - 2].Length];  //At least 2
iterations
                double[] PreviousFlowArray = new
double[ListdFlowArray[CurrentIteration - 3].Length];

                for (int i = 0; i < CurrentFlowArray.Length; i++)
                {
                    CurrentFlowArray[i] =
ListdFlowArray[CurrentIteration - 2][i];
                    PreviousFlowArray[i] =
ListdFlowArray[CurrentIteration - 3][i];
                }
                double PercentChange = 0;
                double SumCurrent = 0;
                double SumPrevious = 0;
                for (int i = 0; i < CurrentFlowArray.Length; i++)
                {
                    SumCurrent += CurrentFlowArray[i];
                    SumPrevious += PreviousFlowArray[i];
                }
                PercentChange = System.Math.Abs(SumCurrent -
SumPrevious) / SumCurrent * 100;
                if (PercentChange < StopThreshold)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }

        public static bool NewStopRule(List<double[]> ListdFlowArray,
int CurrentIteration, double StopThreshold)
        {
            if (CurrentIteration <= 2)
            {
                return false;                    //false indicates
iteration should go on
            }
            else
            {
                double[] CurrentFlowArray = new
double[ListdFlowArray[CurrentIteration - 2].Length];  //At least 2
iterations
                double[] PreviousFlowArray = new
double[ListdFlowArray[CurrentIteration - 3].Length];
                for (int i = 0; i < CurrentFlowArray.Length; i++)
                {
```

```
                    CurrentFlowArray[i] =
ListdFlowArray[CurrentIteration - 2][i];
                    PreviousFlowArray[i] =
ListdFlowArray[CurrentIteration - 3][i];
                }
                byte indicator = 0;
                for (int i = 0; i < CurrentFlowArray.Length; i++)
                {
                    if (System.Math.Abs(CurrentFlowArray[i] -
PreviousFlowArray[i]) > StopThreshold)
                    {
                        indicator = 1;
                    }
                    indicator += indicator;
                }
                if (indicator >= 1)
                {
                    return false;
                }
                else
                {
                    return true;
                }
            }

        }

        //Generate LINGO script
        public static string GenLingoScript(string LingoModelName)
        {
            return "set echoin 0 \n take " + LingoModelName + " \n gen
\n  go \n mem \n quit \n";//gen \n  mem \n
        }

        public static string GoLingoScript(string LingoModelName)
        {
            return "take " + LingoModelName + " \n go \n mem \n quit
\n";
        }

        //Function to create an intermediate flow array
        public static Flow[] CreateIterFlowArray(int PT, int numlinks,
int numdest, List<NodePair> NodePairList, double[] dFlowArray)
        {
            Flow[] IterFlowArray = new Flow[PT * (PT - 1) *
NodePairList.Count * numdest / 2];
            for (int i = 0; i < IterFlowArray.Length; i++)
            {
                IterFlowArray[i] = new Flow();
            }

            int Flow_iter = -1;
```

```
            while (Flow_iter < IterFlowArray.Length - 1)
            {
                for (int t = 0; t < PT; t++)
                {
                    for (int tao = t + 2; tao < PT + 1; tao++)
                    {
                        for (int nodepair = 0; nodepair <
NodePairList.Count; nodepair++)
                        {
                            for (int dest = 0; dest < numdest; dest++)
//dest+1
                            {
                                Flow_iter++;
                                IterFlowArray[Flow_iter].in_time = t +
1;
                                IterFlowArray[Flow_iter].up_node =
NodePairList[nodepair].upnode_no;
                                IterFlowArray[Flow_iter].out_time =
tao;
                                IterFlowArray[Flow_iter].down_node =
NodePairList[nodepair].downnode_no;
                                IterFlowArray[Flow_iter].dest = dest +
1;
                            }
                        }
                    }
                }
            }
            Array.Sort(IterFlowArray);
            for (int k = 0; k < dFlowArray.Length; k++)
            {
                IterFlowArray[k].size = dFlowArray[k];
            }
            return IterFlowArray;
        }

        //Convert Iterative Flow Array to Flow Output to generate
demand for Sub model
        public static List<Flow[]> GetFlowOutput(Flow[] IterFlowArray,
List<NodePair> NodePairList)
        {
            List<Flow> TempFlowList = new List<Flow>();
            List<Flow[]> FlowOutput = new List<Flow[]>();
            for (int i = 0; i < NodePairList.Count; i++)
            {
                for (int j = 0; j < IterFlowArray.Length; j++)
                {
                    if (IterFlowArray[j].up_node ==
NodePairList[i].upnode_no && IterFlowArray[j].down_node ==
NodePairList[i].downnode_no && IterFlowArray[j].size != 0)
                    {
                        TempFlowList.Add(IterFlowArray[j]);
```

```
                    }
                }
                Flow[] TempFlowArray = new Flow[TempFlowList.Count];
                for (int k = 0; k < TempFlowArray.Length; k++)
                {
                    TempFlowArray[k] = (Flow)TempFlowList[k].Clone();
                }
                FlowOutput.Add(TempFlowArray);
                TempFlowList.Clear();
            }
            return FlowOutput;
        }

        //Match up down node pair to a specific link no
        public static int GetLinkNo(double unode, double dnode,
List<NodePair> NodeList)
        {
            int index = 0;
            for (int i = 0; i < NodeList.Count; i++)
            {
                if (unode == NodeList[i].upnode_no && dnode ==
NodeList[i].downnode_no)
                {
                    index = i + 1;
                }
            }
            return index;
        }

        //Convert Flow Output to Sub Demand List
        public static List<Demand>
ConvertFlowOutputDemandList(List<Flow[]> FlowOutput, List<NodePair>
NodeList)
        {
            int NumLinks = FlowOutput.Count;
            //int LinkFlowCount; //number of flow outputs for a link
            List<Demand> AllLinkDemandList = new List<Demand>();
            for (int i = 0; i < NumLinks; i++)
            {
                Flow[] FlowArray = FlowOutput[i]; //Each FlowArray
corresponds to one link
                double[] FlowArrayFlag = new double[FlowArray.Length];
                double Flag = 0;
                while (Factorial(FlowArrayFlag) == 0)
                {
                    Flag++;
                    int FirstNoneZeroFlag = 0; //it is actually the
first ZERO flag
                    for (int j = 0; j < FlowArray.Length; j++)
                    {
                        if (FlowArrayFlag[j] == 0)
                        {
```

```
                            FirstNoneZeroFlag = j;
                            //FlowArrayFlag[j] = Flag;
                            break;
                        }
                    }
                    FlowArrayFlag[FirstNoneZeroFlag] = Flag;
                    double CompareTime =
FlowArray[FirstNoneZeroFlag].in_time;
                    double CompareDest =
FlowArray[FirstNoneZeroFlag].dest;
                    for (int k = FirstNoneZeroFlag + 1; k <
FlowArray.Length; k++)
                    {
                        if (FlowArray[k].in_time == CompareTime &&
FlowArray[k].dest == CompareDest)
                        {
                            FlowArrayFlag[k] = Flag;
                        }
                    }
                }
                for (int k = 0; k < Flag; k++)
                {
                    double TotalDemandSize = 0;
                    double InTime = 0;
                    double Dest = 0;
                    double GenLinkNo = 0;
                    for (int q = 0; q < FlowArray.Length; q++)
                    {
                        if (k + 1 == FlowArrayFlag[q])
                        {
                            TotalDemandSize += FlowArray[q].size;
                            InTime = FlowArray[q].in_time;
                            Dest = FlowArray[q].dest;
                            GenLinkNo =
(double)GetLinkNo(FlowArray[q].up_node, FlowArray[q].down_node,
NodeList);

                        }
                    }
                    Demand temp_Demand = new Demand(GenLinkNo, InTime,
Dest, TotalDemandSize);
                    AllLinkDemandList.Add(temp_Demand);
                }
            }
            return AllLinkDemandList;
        }

        //Determine whether two flow variables are the same except
size
        public static bool EqualFlow(Flow f1, Flow f2)
        {
```

```
            if (f1.in_time == f2.in_time && f1.out_time == f2.out_time
&& f1.up_node == f2.up_node && f1.down_node == f2.down_node && f1.dest
== f2.dest)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        //Get Sub Demand List
        public static List<Demand> GetSubDemandList(Flow[]
IterFlowArray, List<NodePair> NodePairList)
        {
            List<Flow[]> TempFlowList = GetFlowOutput(IterFlowArray,
NodePairList);
            List<Demand> TempDemandList =
ConvertFlowOutputDemandList(TempFlowList, NodePairList);
            return TempDemandList;
        }

        //Flow List to Array Function
        public static Flow[] ConvertFlowOutputToArray(List<Flow[]>
FlowOutput)
        {
            int NumFlows = 0;
            foreach (Flow[] FOiter in FlowOutput)
            {
                NumFlows += FOiter.Length;
            }
            Flow[] FlowArray = new Flow[NumFlows];
            int FlowArrayStartIndex = 0;
            foreach (Flow[] FOiter in FlowOutput)
            {
                for (int i = 0; i < FOiter.Length; i++)
                {
                    FlowArray[FlowArrayStartIndex] =
(Flow)FOiter[i].Clone();
                    FlowArrayStartIndex++;
                }
            }
            return FlowArray;
        }

        #region Sep 2nd Capacity Update Rule
        //Create iterative capacity array
        public static Flow[] CreateIterCapArray(int PT, int numlinks,
int numdest, int nIteration, List<Flow[]> FlowOutput, List<NodePair>
NodePairList, List<Flow[]> ListIterCapArray)
        {
```

```
            Flow[] IterCapArray = new Flow[PT * (PT - 1) *
NodePairList.Count * numdest / 2];
            for (int i = 0; i < IterCapArray.Length; i++)
            {
                IterCapArray[i] = new Flow();
            }

            int Flow_iter = -1;
            while (Flow_iter < IterCapArray.Length - 1)
            {
                for (int t = 0; t < PT; t++)
                {
                    for (int tao = t + 2; tao < PT + 1; tao++)
                    {
                        for (int nodepair = 0; nodepair <
NodePairList.Count; nodepair++)
                        {
                            for (int dest = 0; dest < numdest; dest++)
//dest+1
                            {
                                Flow_iter++;
                                IterCapArray[Flow_iter].in_time = t +
1;
                                IterCapArray[Flow_iter].up_node =
NodePairList[nodepair].upnode_no;
                                IterCapArray[Flow_iter].out_time = tao;
                                IterCapArray[Flow_iter].down_node =
NodePairList[nodepair].downnode_no;
                                IterCapArray[Flow_iter].dest = dest +
1;
                            }
                        }
                    }
                }
            }
            Array.Sort(IterCapArray);

            Flow[] FlowOutputArray =
ConvertFlowOutputToArray(FlowOutput);
            Array.Sort(FlowOutputArray);

            //Create FirstArrivalFlows Array from FlowOutputArray
            List<Flow> FirstArrivalFlows = new List<Flow>();
            FirstArrivalFlows.Add(FlowOutputArray[0]);
            Flow CompareFlow = (Flow)FlowOutputArray[0].Clone();
            for (int i = 0; i < FlowOutputArray.Length; i++)
            {
                if (FlowOutputArray[i].in_time != CompareFlow.in_time
|| FlowOutputArray[i].up_node != CompareFlow.up_node ||
FlowOutputArray[i].down_node != CompareFlow.down_node)
                {
                    CompareFlow = (Flow)FlowOutputArray[i].Clone();
```

```
                    FirstArrivalFlows.Add((Flow)CompareFlow.Clone());
                }
            }
            //add intime to links with flow array
            List<TNodePair> TLinksWithFlow = new List<TNodePair>();
            for (int i = 0; i < FirstArrivalFlows.Count; i++)
            {
                TNodePair tnp = new TNodePair();
                tnp.upnode = FirstArrivalFlows[i].up_node;
                tnp.downnode = FirstArrivalFlows[i].down_node;
                tnp.intime = FirstArrivalFlows[i].in_time;
                TLinksWithFlow.Add((TNodePair)tnp.Clone());
            }
            //Identify links with flow
            NodePair[] LinksWithFlow = new NodePair[FlowOutput.Count];
            for (int i = 0; i < FlowOutput.Count; i++)
            {
                LinksWithFlow[i] = new
NodePair(FlowOutput[i][0].up_node, FlowOutput[i][0].down_node);
            }


            //update IterCapArray
            for (int p = 0; p < IterCapArray.Length; p++)
            {
                if (THasFlow(TLinksWithFlow, IterCapArray[p]))
                {
                    for (int k = 0; k < FlowOutputArray.Length; k++)
//firstly search the flow output
                    {
                        if (FunctionDef.EqualFlow(IterCapArray[p],
FlowOutputArray[k]))
                        {
                            IterCapArray[p].size =
FlowOutputArray[k].size;
                            break; //find it in flow output then break
                        }
                        else
                        {
                            for (int i = 0; i <
FirstArrivalFlows.Count; i++)
                            {
                                if (IterCapArray[p].in_time ==
FirstArrivalFlows[i].in_time && IterCapArray[p].up_node ==
FirstArrivalFlows[i].up_node && IterCapArray[p].down_node ==
FirstArrivalFlows[i].down_node)
                                {
                                    if (IterCapArray[p].out_time <
FirstArrivalFlows[i].out_time)
                                    {
                                        IterCapArray[p].size = 0;
                                    }
```

```
                                    else
                                    {
                                        for (int s = 0; s <
ListIterCapArray[nIteration - 1].Length; s++)
                                        {
                                            if
(FunctionDef.EqualFlow(IterCapArray[p], ListIterCapArray[nIteration -
1][s]))
                                            {
                                                IterCapArray[p].size =
ListIterCapArray[nIteration - 1][s].size;
                                            }
                                        }
                                    }
                                    break;
                                }
                                //if (IterCapArray[p].in_time >
FirstArrivalFlows[i].in_time && IterCapArray[p].up_node ==
FirstArrivalFlows[i].up_node && IterCapArray[p].down_node ==
FirstArrivalFlows[i].down_node)
                                //{
                                //    for (int s = 0; s <
ListIterCapArray[nIteration - 1].Length; s++)
                                //    {
                                //        if
(FunctionDef.EqualFlow(IterCapArray[p], ListIterCapArray[nIteration -
1][s]))
                                //        {
                                //            IterCapArray[p].size =
ListIterCapArray[nIteration - 1][s].size;
                                //        }
                                //    }
                                //    break;
                                //}
                            }
                        }
                    }
                }
                else //without flow then
                {
                    for (int s = 0; s < ListIterCapArray[nIteration -
1].Length; s++)
                    {
                        if (FunctionDef.EqualFlow(IterCapArray[p],
ListIterCapArray[nIteration - 1][s]))
                        {
                            IterCapArray[p].size =
ListIterCapArray[nIteration - 1][s].size;
                            break;
                        }
                    }
                }
```

```
            }
            ListIterCapArray.Add(IterCapArray);
            return IterCapArray;
        }
        #endregion

        #region Sep2nd Revised Update Rule
        public static Flow[] CreateIterCapArray(int PT, int numdest,
int nIteration, double InitialCap, List<Flow[]> FlowOutput,
List<NodePair> NodePairList, List<Flow[]> ListIterCapArray)
        {
            Flow[] IterCapArray = new Flow[PT * (PT - 1) *
NodePairList.Count * numdest / 2];
            for (int i = 0; i < IterCapArray.Length; i++)
            {
                IterCapArray[i] = new Flow();
            }

            int Flow_iter = -1;
            while (Flow_iter < IterCapArray.Length - 1)
            {
                for (int t = 0; t < PT; t++)
                {
                    for (int tao = t + 2; tao < PT + 1; tao++)
                    {
                        for (int nodepair = 0; nodepair <
NodePairList.Count; nodepair++)
                        {
                            for (int dest = 0; dest < numdest; dest++)
//dest+1
                            {
                                Flow_iter++;
                                IterCapArray[Flow_iter].in_time = t +
1;
                                IterCapArray[Flow_iter].up_node =
NodePairList[nodepair].upnode_no;
                                IterCapArray[Flow_iter].out_time = tao;
                                IterCapArray[Flow_iter].down_node =
NodePairList[nodepair].downnode_no;
                                IterCapArray[Flow_iter].dest = dest +
1;
                            }
                        }
                    }
                }
            }
            Array.Sort(IterCapArray);

            Flow[] FlowOutputArray =
ConvertFlowOutputToArray(FlowOutput);
            Array.Sort(FlowOutputArray);
```

```
            //Create FirstArrivalFlows Array from FlowOutputArray
            List<Flow> FirstArrivalFlows = new List<Flow>();
            FirstArrivalFlows.Add(FlowOutputArray[0]);
            Flow CompareFlow = (Flow)FlowOutputArray[0].Clone();
            for (int i = 0; i < FlowOutputArray.Length; i++)
            {
                if (FlowOutputArray[i].in_time != CompareFlow.in_time
|| FlowOutputArray[i].up_node != CompareFlow.up_node ||
FlowOutputArray[i].down_node != CompareFlow.down_node ||
FlowOutputArray[i].dest != CompareFlow.dest)
                {
                    CompareFlow = (Flow)FlowOutputArray[i].Clone();
                    FirstArrivalFlows.Add((Flow)CompareFlow.Clone());
                }
            }
            //add intime to links with flow array
            List<TNodePair> TLinksWithFlow = new List<TNodePair>();
            for (int i = 0; i < FirstArrivalFlows.Count; i++)
            {
                TNodePair tnp = new TNodePair();
                tnp.upnode = FirstArrivalFlows[i].up_node;
                tnp.downnode = FirstArrivalFlows[i].down_node;
                tnp.intime = FirstArrivalFlows[i].in_time;
                TLinksWithFlow.Add((TNodePair)tnp.Clone());
            }

            //Identify last arrival flows
            List<Flow> LastArrivalFlows =
FindLastArrivalFlows(FlowOutputArray);

            for (int p = 0; p < IterCapArray.Length; p++)
            {
                if (THasFlow(TLinksWithFlow, IterCapArray[p]))  //has
flow then
                {
                    for (int k = 0; k < FlowOutputArray.Length; k++)
//firstly search the flow output
                    {
                        if (FunctionDef.EqualFlow(IterCapArray[p],
FlowOutputArray[k]))
                        {
                            IterCapArray[p].size =
FlowOutputArray[k].size;
                            break;
                        }
                        else
                        {
                            if (IsNextLastArrivalFlow(IterCapArray[p],
LastArrivalFlows))
                            {
                                IterCapArray[p].size = InitialCap;
                            }
```

```
                            else
                            {
                                IterCapArray[p].size = 0;
                            }
                        }
                    }
                }

                else //without flow then
                {
                    for (int s = 0; s < ListIterCapArray[nIteration -
1].Length; s++)
                    {
                        if (FunctionDef.EqualFlow(IterCapArray[p],
ListIterCapArray[nIteration - 1][s]))
                        {
                            IterCapArray[p].size =
ListIterCapArray[nIteration - 1][s].size;
                            break;
                        }
                    }
                }
            }
            ListIterCapArray.Add(IterCapArray);
            return IterCapArray;
        }
        #endregion Sep2nd Revised Update Rule

        public static bool HasFlow(NodePair[] LinksWithFlow, Flow flow)
        {
            for (int i = 0; i < LinksWithFlow.Length; i++)
            {
                if (flow.up_node == LinksWithFlow[i].upnode_no &&
flow.down_node == LinksWithFlow[i].downnode_no)
                {
                    return true;
                }
            }
            return false;
        }

        public static bool THasFlow(List<TNodePair> TLinksWithFlow,
Flow flow)
        {
            for (int i = 0; i < TLinksWithFlow.Count; i++)
            {
                if (flow.up_node == TLinksWithFlow[i].upnode &&
flow.down_node == TLinksWithFlow[i].downnode && flow.in_time ==
TLinksWithFlow[i].intime)
                {
                    return true;
                }
```

```
            }
            return false;
        }

        public static List<Flow> FindLastArrivalFlows(Flow[]
FlowOutputArray)
        {
            //Identify latest time-space links that carry flow from
flow output
            List<Flow> LastArrivalFlows = new List<Flow>();

            for (int i = 0; i < FlowOutputArray.Length; i++)
            {
                Flow CompareFlow = (Flow)FlowOutputArray[i].Clone();
                for (int j = i + 1; j < FlowOutputArray.Length; j++)
                {
                    if (CompareFlow.in_time ==
FlowOutputArray[j].in_time && CompareFlow.up_node ==
FlowOutputArray[j].up_node && CompareFlow.down_node ==
FlowOutputArray[j].down_node && CompareFlow.dest ==
FlowOutputArray[j].dest)
                    {
                        if (CompareFlow.out_time <
FlowOutputArray[j].out_time)
                        {
                            CompareFlow =
(Flow)FlowOutputArray[j].Clone();
                        }
                    }
                }
                if (LastArrivalFlows.Count == 0)
                {
                    LastArrivalFlows.Add(CompareFlow);
                }
                int flag = 1;
                for (int k = 0; k < LastArrivalFlows.Count; k++)
                {

                    if (FunctionDef.EqualFlow(LastArrivalFlows[k],
CompareFlow))
                    {
                        flag = flag * 0;
                    }
                }
                if (flag != 0)
                {
                    LastArrivalFlows.Add(CompareFlow);
                }
            }
            return LastArrivalFlows;
        }
```

```
        public static bool IsNextLastArrivalFlow(Flow flow, List<Flow>
LastArrivalFlows)
        {
            for (int i = 0; i < LastArrivalFlows.Count; i++)
            {
                if (flow.in_time == LastArrivalFlows[i].in_time &&
flow.up_node == LastArrivalFlows[i].up_node && flow.down_node ==
LastArrivalFlows[i].down_node && flow.dest == LastArrivalFlows[i].dest)
                {
                    if (flow.out_time == LastArrivalFlows[i].out_time
+ 1)
                    {
                        return true;
                    }
                }
            }
            return false;
        }

        //Get dCapArray
        public static double[] GetdCapArray(Flow[] IterCapArray)
        {
            int dCapArrayDim = IterCapArray.Length;
            double[] dCapArray = new double[dCapArrayDim];
            for (int i = 0; i < dCapArrayDim; i++)
            {
                dCapArray[i] = IterCapArray[i].size;
            }
            return dCapArray;
        }



        ////////////////////////////////////////////Math
Functions////////////////////////////////////
        public static double Factorial(double[] array_fac)
        {
            double fac = 1;
            for (int i = 0; i < array_fac.Length; i++)
            {
                fac = fac * array_fac[i];
            }
            return fac;
        }

        public static double Min(double a, double b, double c, double
d)
        {
            //a:current occ; b out capacity c:in capacity d: congest
effect
            double temp = a;
            if (temp > b)
```

```
            {
                temp = b;

            }
            if (temp > c)
            {
                temp = c;
            }
            if (temp > d)
            {
                temp = d;
            }
            return temp;
        }
    }
}
```

Main Program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.Remoting;

namespace LP_DTA_MOD
{
    class Program
    {
        //public const int PT = 30;                  //Planning Horizon
        //public const double clock_interval = 10;


        static void Main(string[] args)
        {

////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////
            /*-----------------------------------------Initialization
Module-------------------------------------------------*/

////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////
            //Small Test Network
            int numlinks = 0;
            int numnodes = 0;
            const int PT = 15;                  //Planning Horizon
            const double clock_interval = 10;
            const double InitialCap = 40;       //Initial value for
capacity array
```

```
            const double StopThreshold = 0.1;  //Flow sum change
percentage



            string linkdatafile = "Link.txt";
            string LingoDemandFile = "LingoDemand.txt";
            string IncidentFile = "Incident.txt";

            //--------------------------Initialize the Network--------
------------------//
            DateTime StartTime = DateTime.Now;
//Record the current time
            FunctionDef.GetNetworkDim(linkdatafile, ref numlinks, ref
numnodes);

            Network testnetwork = new Network(numlinks, numnodes);
//Create the network
            FunctionDef.ini_linklist(linkdatafile,
testnetwork.linklist);        //Create link list the network
            testnetwork.ConstructAdjMatrix();
//Construct the adjacency matrix

            double[] NodePred = new double[numnodes];
            double[] NodeSucc = new double[numnodes];
            //FunctionDef.CreateNodeAdjArrays(NodePred, NodeSucc,
testnetwork.AdjacencyMatrix);  //Predecessor and successors passed to
LINGO

            List<NodePair> NodePairList =
FunctionDef.ConvertAdjMatrixNodePairList(testnetwork.AdjacencyMatrix);
            FunctionDef.CreateNodeAdjArrays(NodePred, NodeSucc,
testnetwork.AdjacencyMatrix);
            for (int i = 0; i < testnetwork.linklist.Length; i++)
            {
                testnetwork.linklist[i].ini_celllist(clock_interval);
//Initialize cell list for each link
            }

            //Create Incident List
            List<Incident> IncidentList = new List<Incident>();
            FunctionDef.CreateIncidentList(IncidentFile,  IncidentList);

            //Create Demand Array to Initialize LINGO Variables
            int Ldemand_horizon = 0;     //latest time interval during
which demand will be generated
            int no_dest = 0;
            List<LingoDemand> LDemandList =
FunctionDef.CreateLingoDemandList(LingoDemandFile);
            FunctionDef.AnalyzeLingoDemand(LDemandList, ref
Ldemand_horizon, ref no_dest);
```

```
            int dFlowArrayDim = PT * (PT - 1) * NodePairList.Count *
no_dest / 2;  //!!!!!!!!!!!!!dimension change!!!!!!!!!!!!!!!

            double[] LStartNode = new double[no_dest];
            double[] LEndNode = new double[no_dest];
            FunctionDef.GetLingoNodeArrays(LDemandList, LStartNode,
LEndNode);

            double Cdemand_horizon =
System.Convert.ToDouble(Ldemand_horizon);
            double Cno_dest = System.Convert.ToDouble(no_dest);
            double[] LingoDemandSizeArray =
FunctionDef.CreateArrayLingoDemandSize(LDemandList, ref
Cdemand_horizon, ref Cno_dest, numnodes);

            //Create the two major variables
            double[] dFlowArray = new double[dFlowArrayDim];
            double[] dCapArray = new double[dFlowArrayDim];
            dFlowArray.Initialize();


            //Results Storage
            List<double[]> ListdFlowArray = new List<double[]>();
//Store dFlowArray for each iteration
            List<double> ListdObjective = new List<double>();
//Store Obj value for each iteration
            List<Flow[]> ListIterCapArray = new List<Flow[]>();
//Store IterCapArray for each iteration



            ////////////////////////INITIALIZATION OF LINGO API
MODEL//////////////////////////////////////
            IntPtr pLingoEnv;
            int nError = -1, nPointersNow = -1;    //Error indicator,
Pointer locator
            double dObjective = -1, dStatus = -1;
            int nIteration = 1;                    //Number of
Iterations

            // Get a pointer to a Lingo environment
            pLingoEnv = lingo.LScreateEnvLng();
            if (pLingoEnv == IntPtr.Zero)
            {
                Console.WriteLine("Unable to create Lingo
environment.\n");
                goto FinalExit;
            }
            // Open LINGO's log file
            nError = lingo.LSopenLogFileLng(pLingoEnv, "lingo.log");
            if (nError != lingo.LSERR_NO_ERROR_LNG) goto ErrorExit;
```

```
            //Pin LINGO's transfer areas in memory
            unsafe
            {
                fixed (
                    double* Pnumnodes = new double[1],
                            PPT = new double[1],
//Planning Horizon
                            Pdemand_horizon = new double[1],
                            Pno_dest = new double[1],
                            Pnodeno = new double[numnodes],
                            PNodePred = NodePred,
                            PNodeSucc = NodeSucc,
                            PLStartNode = LStartNode,
                            PLEndNode = LEndNode,
                            PdFlowArray = dFlowArray,
                            PdCapArray = dCapArray,
                            PdLdemandArray = LingoDemandSizeArray)
                {
                    for (int i = 0; i < dFlowArrayDim; i++)
                    {
                        dCapArray[i] = InitialCap;
//Initialize the dCapArray
                    }
                    Flow[] InitialCapArray =
FunctionDef.CreateIterFlowArray(PT, numlinks, no_dest, NodePairList,
dCapArray);
                    ListIterCapArray.Add(InitialCapArray);

                    for (int i = 0; i < numnodes; i++)
                    {
                        Pnodeno[i] = i + 1;
                    }

                    //Initialize the pointers
                    double Tnumnodes =
System.Convert.ToDouble(numnodes);
                    Pnumnodes[0] = Tnumnodes;
                    double TPT = System.Convert.ToDouble(PT);
                    PPT[0] = TPT;
                    double Tdemand_horizon =
System.Convert.ToDouble(Ldemand_horizon);
                    Pdemand_horizon[0] = Tdemand_horizon;
                    double Tno_dest = System.Convert.ToDouble(no_dest);
                    Pno_dest[0] = Tno_dest;

                    ////////////////////////////Associate C# pointers
with LINGO variables//////////////////////////

                    //numnodes -> Pointer(1)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
Pnumnodes, ref nPointersNow);
```

```
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //nodeno -> Pointer(2)
                    nError = lingo.LSsetPointerLng(pLingoEnv, Pnodeno,
ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PNodePred -> Pointer(3)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PNodePred, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PNodeSucc -> Pointer(4)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PNodeSucc, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PPT -> Pointer(5)
                    nError = lingo.LSsetPointerLng(pLingoEnv, PPT, ref
nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //Pdemand_horizon -> Pointer(6)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
Pdemand_horizon, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //no_dest -> Pointer(7)
                    nError = lingo.LSsetPointerLng(pLingoEnv, Pno_dest,
ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //startnode -> Pointer(8)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PLStartNode, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //endnode -> Pointer(9)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PLEndNode, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PdFlowArray -> Pointer(10)
```

```
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PdFlowArray, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PdCapArray -> Pointer(11)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PdCapArray, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //PdLdemandArray -> Pointer(12)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
PdLdemandArray, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //dObjective -> Pointer(13)
                    nError = lingo.LSsetPointerLng(pLingoEnv,
&dObjective, ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    //dStatus -> Pointer(14)
                    nError = lingo.LSsetPointerLng(pLingoEnv, &dStatus,
ref nPointersNow);
                    if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;

                    ////////////////////////Pointers Assignment Ends
Here//////////////////////////////////

                    //Generate the LINGO script
                    string LingoModelName =
"C:\\Users\\weihao\\Documents\\Visual Studio
2008\\Projects\\LP_DTA_MOD\\LP_DTA_MOD\\LP-DTA-MOD.lng";
                    string cScript =
FunctionDef.GenLingoScript(LingoModelName);

                    while (!FunctionDef.StopRuleCheck(ListdFlowArray,
nIteration, StopThreshold))
                    {
                        if (nIteration == 2)
                        {
                            cScript =
FunctionDef.GoLingoScript(LingoModelName);
                        }
                        nError = lingo.LSexecuteScriptLng(pLingoEnv,
cScript);
                        if (nError != lingo.LSERR_NO_ERROR_LNG) goto
ErrorExit;
```

```
                        // Any problems?
                        if (nError != 0 || dStatus !=
lingo.LS_STATUS_GLOBAL_LNG)
                        {
                            // Had a problem
                            Console.WriteLine("Unable to solve!");
                            Console.WriteLine("Error occurs in
iteration {0} !", nIteration);
                            goto ErrorExit;
                            //break;
                        }

                        else
                        {
                            double[] TdFlowArray =
(double[])dFlowArray.Clone();
                            ListdFlowArray.Add(TdFlowArray);
                            ListdObjective.Add(dObjective);
                            Flow[] IterFlowArray =
FunctionDef.CreateIterFlowArray(PT, numlinks, no_dest, NodePairList,
dFlowArray);
                            List<Demand> SubDemandList =
FunctionDef.GetSubDemandList(IterFlowArray, NodePairList);
                            dCapArray =
LinkSubModel.LinkModel(nIteration, SubDemandList, testnetwork,
IncidentList, NodePairList, ListIterCapArray);
                            for (int i = 0; i < dCapArray.Length; i++)
//capacities for links without flow should not change
                            {
                                PdCapArray[i] = dCapArray[i];
                            }
                            nIteration++;
                        }
                    }
                }
            }
            goto NormalExit;
        ErrorExit:
            if (dStatus != 0)
            {
                Console.WriteLine("LINGO Status Code: {0}\n", dStatus);
            }
            else
            {
                Console.WriteLine("LINGO Error Code: {0}\n", nError);
            }

        NormalExit:
            // Close the log file
            lingo.LScloseLogFileLng(pLingoEnv);

            // Free Lingo's envvironment to avoid a memory leak
```

```
            lingo.LSdeleteEnvLng(pLingoEnv);

        FinalExit:
            DateTime EndTime = DateTime.Now;
            TimeSpan ExecutionTime = EndTime - StartTime;
            double[] BasicOutput = new double[5];
            BasicOutput[0] = ExecutionTime.TotalSeconds;
            BasicOutput[1] = System.Convert.ToDouble(PT);
            BasicOutput[2] = System.Convert.ToDouble(numnodes);
            BasicOutput[3] = System.Convert.ToDouble(numlinks);
            BasicOutput[4] = System.Convert.ToDouble(no_dest);
            //Console.WriteLine("press enter...");
            //String sTemp = Console.ReadLine();
            Flow[] FinalFlowArrayOutput =
FunctionDef.CreateIterFlowArray(PT, numlinks, no_dest, NodePairList,
ListdFlowArray[ListdFlowArray.Count - 1]);
            OutputModule.OutputExcel(FinalFlowArrayOutput,  BasicOutput,
PT, numlinks, NodePairList);


        }
    }
}
```

Link Flow Model

```
using System;
using System.Collections.Generic;
using System.Text;

namespace LP_DTA_MOD
{
    class LinkSubModel
    {
        const double precision_control = 0.000001;
        const double wave_coeff = 1;
        const int no_dest = 1;              //how to reference
variables in other classes'main function
        const int PT = 15;
        const double InitialCap = 40;

        public static double[] LinkModel(int nIteration, List<Demand>
SubDemandList, Network testnetwork, List<Incident> IncidentList,
List<NodePair> NodePairList, List<Flow[]> ListIterCapArray)
        {


            for (int link_iter = 0; link_iter < testnetwork.numlinks;
link_iter++)
            {
```

```
                List<Demand> CurrentLinkDemandList = new
List<Demand>();
                CurrentLinkDemandList =
FunctionDef.GetLinkDemand(SubDemandList,
testnetwork.linklist[link_iter].linkno); //find the demand for this
link
                double total_demand = 0;
                for (int i = 0; i < CurrentLinkDemandList.Count; i++)
                {
                    total_demand = total_demand +
CurrentLinkDemandList[i].demand_size;
                }


                //duplicate the cell list for each link
                Cell[] celllist =
testnetwork.linklist[link_iter].celllist;
                int total_no_cells = celllist.Length;

                for (int t = 0;
System.Math.Abs(celllist[total_no_cells - 1].occ_record[t] -
total_demand) > precision_control; t++)
                {
                    //determine the demand at the current time
interval
                    double current_demand = 0;  //aggregate demand,
demand is loaded from t=0
                    List<int> demandlist_index = new List<int>();

                    demandlist_index =
FunctionDef.Get_currentdemand_index(t, CurrentLinkDemandList); //index
of the demand in the demand list at the current time interval
                    for (int i = 0; i < demandlist_index.Count; i++)
                    {
                        int temp_index = demandlist_index[i];
                        current_demand = current_demand +
CurrentLinkDemandList[temp_index].demand_size;
                    }


                    //Determine Cell Capacity for the current time
interval
                    double[] AllCellCapArray =
FunctionDef.DetermineAllCellCapacity(testnetwork.linklist[link_iter],
t, link_iter + 1, IncidentList);
                    for (int i = 0; i < celllist.Length; i++)
                    {
                        celllist[i].cap = AllCellCapArray[i];
                    }
                    for (int i = 0; i < celllist.Length; i++)
                    {
```

```
                        if (celllist[i].type == 1)
                        {
                            double congest = 0;
                            congest = wave_coeff * (celllist[i +
1].max_occ - celllist[i + 1].occ_record[t]);

                            celllist[i].outflow =
FunctionDef.Min(celllist[i].occ_record[t], celllist[i].cap, celllist[i
+ 1].cap, congest);
                            if (celllist[i].outflow <
precision_control)
                            {
                                celllist[i].outflow = 0;
                            }

celllist[i].flow_record.Add(celllist[i].outflow);
                            double next_occ = celllist[i].occ_record[t]
- celllist[i].flow_record[t] + current_demand;
                            celllist[i].occ_record.Add(next_occ);

                            if (current_demand != 0)
                            {
                                for (int j = 0; j <
demandlist_index.Count; j++)
                                {
                                    Packet temp_packet = new
Packet(CurrentLinkDemandList[demandlist_index[j]]);// Generate packets
according to demand

celllist[i].packetlist.Add(temp_packet);
                                }
                            }
                        }

                        if (celllist[i].type == 2)
                        {
                            double congest = 0;
                            congest = wave_coeff * (celllist[i +
1].max_occ - celllist[i + 1].occ_record[t]);
                            //determine outflow for the current time
interval
                            if (celllist[i + 1].type != 3) //sink cell
has infinite capacity and max occupancy
                            {
                                celllist[i].outflow =
FunctionDef.Min(celllist[i].occ_record[t], celllist[i].cap, celllist[i
+ 1].cap, congest);
                                if (celllist[i].outflow <
precision_control)
                                {
                                    celllist[i].outflow = 0;
                                }
```

```
                            }
                            else
                            {
                                celllist[i].outflow =
System.Math.Min(celllist[i].occ_record[t], celllist[i].cap);
                                if (celllist[i].outflow <
precision_control)
                                {
                                    celllist[i].outflow = 0;
                                }
                            }

celllist[i].flow_record.Add(celllist[i].outflow);
                            double next_occ = celllist[i].occ_record[t]
- celllist[i].flow_record[t] + celllist[i - 1].flow_record[t];
                            celllist[i].occ_record.Add(next_occ);
                        }

                        if (celllist[i].type == 3)
                        {
                            double next_occ = celllist[i].occ_record[t]
+ celllist[i - 1].flow_record[t];
                            celllist[i].occ_record.Add(next_occ);

                            // Record the inflow to the sink cell at
each time interval
                            double current_inflow = celllist[i -
1].flow_record[t];

celllist[i].flow_record.Add(current_inflow); //sink cell's flow record
is for inflow
                        }


                        int agg_packet_no = 0; //agg_packet_no is the
index of the time interval that should be split in all_entry array
                        celllist[i].packetlist =
FunctionDef.ListManipulate(celllist[i].packetlist);//merge packets
first
                        if (celllist[i].flow_record[t] != 0)
                        {
                            if (celllist[i].type != 3)
                            {
                                double temp_flow =
celllist[i].flow_record[t]; //how many vehicles leave
                                Packet[] current_packets = new
Packet[celllist[i].packetlist.Count]; //current_packets is an array of
packets
                                current_packets =
FunctionDef.ListToArray(celllist[i].packetlist);
```

```
                                double[] all_entry =
FunctionDef.GetAllEntryTime(celllist[i].packetlist); //identify all
the entry times in current packetlist
                                double[] agg_size =
FunctionDef.AggPacketSize(celllist[i].packetlist); //aggregate packets
based on entry time intervals
                                double cumulative_size = agg_size[0];

                                while (temp_flow - cumulative_size >
precision_control)   //(cumulative_size < temp_flow)
                                {
                                    agg_packet_no++;

                                    cumulative_size +=
agg_size[agg_packet_no];
                                }

                                double latest_entry =
all_entry[agg_packet_no]; //latest entry time packets can leave
                                int num_packets =
FunctionDef.GetNumPacketsLeave(latest_entry,
current_packets);//determine total number of packets to leave
                                Packet[] move_packets = new
Packet[num_packets];

                                if (System.Math.Abs(cumulative_size -
temp_flow) < precision_control)//(cumulative_size == temp_flow) //no
need to split
                                {
                                    for (int j = 0; j < num_packets;
j++)
                                    {
                                        current_packets[j].entry_time
= t + 1;
                                        move_packets[j] =
current_packets[j].copy_packet(current_packets[j]); //packets allowed
to leave
                                    }
                                    for (int j = 0; j < num_packets;
j++)
                                    {
                                        celllist[i +
1].packetlist.Add(move_packets[j]);
                                    }


celllist[i].packetlist.RemoveRange(0, num_packets); //remove packets
having left from packetlist of current cell


                                }
```

```
                                if (cumulative_size - temp_flow >
precision_control)
                                {
                                    double size_wholepacket = 0;
//record the total size of packets leave as a whole
                                    for (int j = 0; j <
current_packets.Length; j++) //THERE IS A CHANGE IN THIS LOOP
                                    {
                                        if
(current_packets[j].entry_time <= latest_entry - 1) //time previous to
latest-entry is the latest time for packets leaving in whole
                                        {
                                            size_wholepacket +=
current_packets[j].size;
                                            move_packets[j] =
current_packets[j].copy_packet(current_packets[j]);  //packets allowed
to leave as a whole

                                        }
                                    }

                                    int num_packets_whole = 0;
//packet 0 to packet (num_packet_whole-1) within move_packets list are
ones exiting as a whole
                                    for (int p = 0; p <
move_packets.Length; p++)
                                    {
                                        if (move_packets[p] != null)
                                        {
                                            num_packets_whole++;
                                        }
                                    }

                                    //identify packets that need to be
split
                                    double size_partial_outflow =
temp_flow - size_wholepacket; //total outflow for packets to be split

                                    int num_packets_split =
num_packets - num_packets_whole;//find the number of packets to split

                                    //find all the packets to be split
and store them in an array  //THERE IS A CHANGE HERE
                                    Packet[] packets_split_array = new
Packet[num_packets_split];
                                    List<Packet> packets_split_list =
new List<Packet>();
                                    for (int p = 0; p <
current_packets.Length; p++)
                                    {
```

```
                                        if
(current_packets[p].entry_time == latest_entry)
                                        {

packets_split_list.Add(current_packets[p]);
                                            if
(celllist[i].packetlist[p].entry_time > latest_entry)
                                            {
                                                break;
                                            }
                                        }
                                    }
                                    packets_split_array =
FunctionDef.ListToArray(packets_split_list);

                                    //Generate new packets after
splitting
                                    double[] size_newpartialpackets =
FunctionDef.GetPartialPacketSize(size_partial_outflow,
packets_split_array); //Calculate size for newly generated packets
                                    for (int p = 0; p <
packets_split_list.Count; p++)
                                    {
                                        packets_split_list[p].size =
size_newpartialpackets[p];
                                    }
                                    packets_split_array =
FunctionDef.ListToArray(packets_split_list);
                                    for (int p = num_packets_whole; p
< num_packets; p++)
                                    {
                                        move_packets[p] =
packets_split_array[p -
num_packets_whole].copy_packet(packets_split_array[p -
num_packets_whole]);
                                    }

                                    //update entry time of packets in
the move_packets array by adding 1 time increment
                                    for (int p = 0; p <
move_packets.Length; p++)
                                    {
                                        move_packets[p].entry_time = t
+ 1;
                                    }

                                    //Update current packet lists then
add move_packets to the next cell's packet list

celllist[i].packetlist.RemoveRange(0, num_packets_whole);
                                    for (int p = 0; p <
num_packets_split; p++)
```

```
                                    {
                                        celllist[i].packetlist[p].size
= celllist[i].packetlist[p].size - size_newpartialpackets[p];
                                    }

                                    //add all the packets leaving to
the end of the packetlist of the downstream cell
                                    for (int j = 0; j < num_packets;
j++)
                                    {
                                        celllist[i +
1].packetlist.Add(move_packets[j]);

                                    }
                                }
                            }
                        }
                    }
                }

            ////////////////////////////////Post Processing to Obtain
One-dimensional Array Output//////////////////////
            List<Flow[]> FlowOutput = new List<Flow[]>();
            for (int link_iter = 0; link_iter < testnetwork.numlinks;
link_iter++)
            {
                Link CurrentLink = testnetwork.linklist[link_iter];
                int SinkCell_index =
testnetwork.linklist[link_iter].celllist.Length - 1;
                Cell SinkCell =
testnetwork.linklist[link_iter].celllist[SinkCell_index];
                int FlowArraySize = SinkCell.packetlist.Count;
                Flow[] FlowArray = new Flow[FlowArraySize];
                if (FlowArraySize != 0)
                {
                    for (int i = 0; i < FlowArraySize; i++)
                    {
                        FlowArray[i] = new Flow(CurrentLink,
SinkCell.packetlist[i]);
                    }
                    FlowOutput.Add(FlowArray);
                }
            }
            //Flow[] IterCapArray = FunctionDef.CreateIterCapArray(PT,
testnetwork.numlinks, no_dest, nIteration, FlowOutput, NodePairList,
ListIterCapArray);

            Flow[] IterCapArray = FunctionDef.CreateIterCapArray(PT,
no_dest, nIteration, InitialCap, FlowOutput, NodePairList,
ListIterCapArray);
```

```
            double[] dCapArray =
FunctionDef.GetdCapArray(IterCapArray);

            ////////////////////////////////Reset Flow Record and
Occupancy Record for each cell//////////////////////////
            foreach (Link link_iter in testnetwork.linklist)
            {
                foreach (Cell cell_iter in link_iter.celllist)
                {
                    cell_iter.flow_record.Clear();
                    cell_iter.occ_record.Clear();
                    cell_iter.occ_record.Add(0);
                    cell_iter.occ = 0;
                    cell_iter.outflow = 0;
                    cell_iter.packetlist.Clear();
                    cell_iter.cap = cell_iter.default_cap;
                }
            }

            return dCapArray;
        }
    }
}
```

Output Module

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using Microsoft.Office.Interop.Excel;
using System.Reflection;

namespace LP_DTA_MOD
{
    class OutputModule
    {
        public static void OutputExcel(Flow[] FlowArrayOutput, double[]
BasicOutput, int PT, int numlinks, List<NodePair> NodePairList)
        {
            Microsoft.Office.Interop.Excel.Application m_objExcel =
new Microsoft.Office.Interop.Excel.Application();

            m_objExcel.Visible = false;
            m_objExcel.UserControl = true;
            Workbooks m_objBooks = m_objExcel.Workbooks;

            System.Globalization.CultureInfo ci = new
System.Globalization.CultureInfo("en-US");
```

```
            m_objBooks.GetType().InvokeMember("Add",
BindingFlags.InvokeMethod, null, m_objBooks, null, ci);

            Workbook m_objBook =
m_objBooks.Add(XlWBATemplate.xlWBATWorksheet);
            Worksheet m_objSheet = (Worksheet)m_objBook.Worksheets[1];

            // Create an array for the headers and add it to cells
A1:A5.
            object[] objHeaders = { "Program Execution Time",
"Planning Horizon", "Number of Nodes", "Number of Links", "Number of
OD Pairs" };
            Range m_objRange = m_objSheet.get_Range("A1", "E1");

m_objRange.set_Value(XlRangeValueDataType.xlRangeValueDefault,
objHeaders);  //write column headers
            Font m_objFont = m_objRange.Font;  //change fonts
            m_objFont.Bold = true;
            m_objRange.EntireColumn.AutoFit();
            m_objRange.HorizontalAlignment = XlHAlign.xlHAlignCenter;

            //Create an array for the headers for flow output
            object[] ObjHeadersFlow = { "Departure Time", "Departure
Node", "Arrival Time", "Arrival Node", "OD Pair", "Size" };
            m_objRange = m_objSheet.get_Range("A4", "F4");

m_objRange.set_Value(XlRangeValueDataType.xlRangeValueDefault,
ObjHeadersFlow);  //write column headers
            m_objFont = m_objRange.Font;  //change fonts
            m_objFont.Bold = true;
            m_objRange.EntireColumn.AutoFit();
            m_objRange.HorizontalAlignment = XlHAlign.xlHAlignCenter;

            //Write Output to Excel Spreadsheet
            //Write Basic Network Characteristics
            m_objRange = m_objSheet.get_Range("A2", "E2");

m_objRange.set_Value(XlRangeValueDataType.xlRangeValueDefault,
BasicOutput);
            m_objRange.EntireColumn.AutoFit();
            m_objRange.HorizontalAlignment = XlHAlign.xlHAlignCenter;

            //Write Flow Output
            string[] RangeName = new string[2];
            for (int i = 0; i < FlowArrayOutput.Length; i++)
            {
                double[] TempFlowInfo = new double[6];
                TempFlowInfo[0] = FlowArrayOutput[i].in_time;
                TempFlowInfo[1] = FlowArrayOutput[i].up_node;
                TempFlowInfo[2] = FlowArrayOutput[i].out_time;
                TempFlowInfo[3] = FlowArrayOutput[i].down_node;
                TempFlowInfo[4] = FlowArrayOutput[i].dest;
```

```
                TempFlowInfo[5] = FlowArrayOutput[i].size;
                if (TempFlowInfo[5] != 0)
                {
                    int RowNum = 5;
                    Range RowtoWrite = m_objSheet.get_Range("A5",
"F5");
                    Range FirstCell = (Range)m_objSheet.Cells[5, 1];
                    while (FirstCell.Value2 != null)
                    {
                        RowNum++;
                        FirstCell = (Range)m_objSheet.Cells[RowNum, 1];
                    }

                    RangeName[0] = "A" + (RowNum).ToString();
                    RangeName[1] = "F" + (RowNum).ToString();
                    m_objRange = m_objSheet.get_Range(RangeName[0],
RangeName[1]);

m_objRange.set_Value(XlRangeValueDataType.xlRangeValueDefault,
TempFlowInfo);
                    m_objRange.EntireColumn.AutoFit();
                    m_objRange.HorizontalAlignment =
XlHAlign.xlHAlignCenter;
                }
            }



            //Save the Output Excel Spreadsheet
            string SaveTime = DateTime.Now.Hour.ToString() +
DateTime.Now.Minute.ToString() + DateTime.Now.Second.ToString();
            string path =
"C:\\Users\\weihao\\Documents\\ProjectOutput\\" + SaveTime;
            Directory.CreateDirectory(path);
            string ExcelFileName = path + "\\" + "FlowOutput";
            m_objExcel.ActiveWorkbook.SaveAs(ExcelFileName,
                XlFileFormat.xlWorkbookDefault,
                Type.Missing,
                Type.Missing,
                false,
                false,
                XlSaveAsAccessMode.xlNoChange,
                Type.Missing,
                Type.Missing,
                Type.Missing,
                Type.Missing,
                Type.Missing);

            m_objBook.Close(true,
"C:\\Users\\weihao\\Documents\\ProjectOutput\\Output", false);
            m_objExcel.Quit();
```

```
            //CSV Output
            List<double[]> AggFlow = new List<double[]>();
            for (int i = 0; i < numlinks; i++)  //loop through links
            {
                double[] AggFlowArray = new double[PT];
                for (int t = 0; t < AggFlowArray.Length + 1; t++)
                {
                    for (int j = 0; j < FlowArrayOutput.Length; j++)
                    {
                        if (FlowArrayOutput[j].up_node ==
NodePairList[i].upnode_no && FlowArrayOutput[j].down_node ==
NodePairList[i].downnode_no && FlowArrayOutput[j].in_time <= t + 1 &&
FlowArrayOutput[j].out_time >= t + 2)
                        {
                            AggFlowArray[t]  +=  FlowArrayOutput[j].size;
                        }
                    }
                }
                AggFlow.Add(AggFlowArray);
            }

            //Write to a CSV file

            string CSVFileName = path + "\\" + "AggFlows.csv";
            using (StreamWriter writer = new StreamWriter(CSVFileName))
            {
                int colCount = AggFlow.Count;  //number of columns
                for (int i = 0; i < colCount + 1; i++)
                {
                    if (i == 0)
                    {
                        writer.Write("Time Interval" + ",");
                    }
                    else
                    {
                        if (i != colCount)
                        {
                            writer.Write("Link " + i.ToString() + ",");
                        }
                        else
                        {
                            writer.Write("Link " + i.ToString());
                        }
                    }
                }
                string LineTerminator = writer.NewLine;
                writer.Write(LineTerminator);
                for (int i = 0; i < PT; i++)
                {
                    writer.Write((i + 1).ToString() + ",");
                    for (int j = 0; j < numlinks; j++)
                    {
```

```
                        writer.Write(AggFlow[j][i].ToString() + ",");
                    }
                    writer.Write(LineTerminator);
                }
                writer.Flush();
                writer.Close();

            }
        }
    }
}
```