A Programmer's Manual for Toolkit of Network Modeling (TNM)

Yu (Marco) Nie

Department of Civil & Environmental Engineering
University of California, Davis

March 22, 2006

Contents

Chapter 1

Introduction

1.1 A Brief History

The idea of building a handy programming toolkit for network problems of different types originated in September 2003, when I started to implement Michael Bell's Path Flow Estimator (1997) for estimating travel demands (a project sponsored by CALTRANS). I realized that I had to repeat a lot of programming work that I had done before, such as network I-O functions and shortest path search. This is a waste of time and energy, in my opinion. I therefore decided to develop a way to wrap up these common functions such that I could conveniently use them whenever a need arises in the future. At that time, I was still new to the object-oriented programming (OOP), even though I had implemented (cooperated with Xiaojian Nie) a preliminary macroscopic traffic simulation model called UCDNL using C++. However, it was very clear to me that the objective I envisioned would only be achieved using the OOP technique.

The first version of TNM, intended for static applications only, was "released" in November 2003. It is basically a network class written in C++ encapsulating data and a bunch of operations (e.g., shortest path search, maximum flow, path enumeration). Shortly after the birth of TNM1, I began to think of adding dynamic operations, particularly the function of dynamic network loading (DNL), on top of it. This was primarily driven by the need of solving the dynamic traffic assignment problems. As an alternative, I could also build dynamic applications on UCDNL I mentioned before, and let TNM exclusively deal with static problems. This option did seem appealing at that time, for the core DNL function had been finished in UCDNL. However, the price I had to pay for the convenience is the violation of the uniform programming interface I pursued from the onset. Consequently, I might have to maintain two separate sets of codes which may have substantial overlaps. I struggled for a while and decided to discard UCDNL completely and rewrote the DNL function in the framework of TNM.

The first few attempts, nevertheless, were not very successful, mainly because I did not figure out a neat and extendable class hierarchy. The integration of static and dynamic objects had not been realized until February 2004, when TNM2 finally emerged. TNM2 implemented three network loading modes (predictive, reactive and mixed) and two traffic flow models (cell transmission mode and point-queue model). In view of the potential need for post-processing, a special class was designed to communicate with MYSQL database [1]. TNM2 was immediately used to solve the dynamic traffic assignment problems with and without departure time choices. By the end of April 2004, I had tested a few small DTA examples based on TNM2 and produced numerical results consistent with analytical solutions. The follow-up developments were focused on improving the accuracy of loading results. At Professor Michael Zhang's suggestion, I introduced the concept of flow scalar into TNM2 to resolve large rounding errors associated with small loading intervals in earlier versions. TNM2 still considers all nodes as if they were freeway merge, diverge or their combinations. This is rather limited for representing urban networks with signals. Jingtao Ma, then a colleague of mine, showed interests in adding signal control elements into TNM. He built into TNM3 a family of control node types in June 2004, which however had not between tested and debugged intensively until very recent (January 2006). Since then no fundamental update was made to TNM until the summer of 2005. Some of work done during the one year period was merely incremental improvements (i.e., additional traffic flow models, incidents handler etc.) or bug removals. Others were mainly made to fit into the needs of two other products I was developing concurrently: the graphic user interface of TNM and MAT (the models and algorithms toolkit).

In September 2005, I integrated into TNM4 a microscopic traffic simulation function based on the cellular automata (CA) model. Lane-changing and merging behavior of individual vehicles were modeled using a bidding algorithm. The vehicle class was greatly extended and enhanced to handle microscopic behaviors. Accompanying with this update were also two new network classes (other than static and dynamic classes), one for microscopic application and the other for stochastic applications.

From the time when we were about to finish coding UCDNL, we had been debating on whether or not a corresponding graphical user interface (GUI) should be developed. This was considered a mission beyond the capacity of our programming skills because then both Xiaojian Nie and me were not familiar with any GUI development tool. Moreover, it seemed not wise to have doctoral students spend their precious time on producing something unlikely publishable. Professor Zhang thus advised us to leave the development of GUI to a more suitable and capable student (e.g., a master student with a CS background and interested in writing codes). Not surprisedly,

[1] This class turned out not very useful so far and I am seriously considering to remove all MYSQL-related parts from TNM.

such an ideal student did not appear very soon [2]. Yet at that time I badly wanted a tool which could automatically display networks according to our own format (DANET2). Mainly this is because the topology of networks proved very useful to debugging. In retrospect, there could be a number of ways to achieve this goal. But what I went for finally was MATLAB, partly because I was familiar with its visualization functions. I spent less than two weeks to finish the first MATLAB-based GUI, which was a very crude product and intended to be only used by myself [3] as a debugging tool. It contained nothing more than a few preliminary functions to plot link and nodes as well as their names (ID). I gradually expanded the functionality of this MATLAB program (named VILOAD) during the next year and a half, as the requirement of debugging forced me to do so. For example, I had to animate the simulation process in order to better understand why and how a gridlock (i.e., traffic comes to a halt) happens. By December 2004, VILOAD had integrated functions like interactive network editing, simulation animation and post-process visualization.

I would never think of turning VILOAD into a MFC program should I have not been asked to write a simple GUI for the PFE project. The task did push me to systematically study MFC and finally build the first MFC prototype in January 2005. Even though it was originally stimulated by the PFE project, this new GUI was intentionally designed to support TNM from the beginning. I thus named it VTNM. The first VTNM only included functions for editing and displaying networks. Yet more functions were to come soon: the shortest path search and object locating tools were added in February, the basic animation of simulation and an enhanced Plot Graphic Library (PGL) for post-processing were added in May, and in June I incorporated an incident handler and a log window to trace error messages. DTA functions were included into VTNM by the end of June. In August, I introduced a cell-based scheme for animating traffic simulation which turned out very useful. As TNM started to support microscopic simulation from v4.0, I also implemented corresponding visualization components in VTNM, which was finished on September 30, 2005.

The last significant update of VTNM was made by myself in January 2006, which added functions for dynamic estimation of O-D trip tables (The theme of my Ph.D. dissertation). Jingtao Ma joined the development of VTNM in early 2006 and was primarily targeting the design of signalized intersections. His work, nevertheless, has yet been finalized and released as this manuscript is prepared.

[2] Hu Dong joined us in Spring 2004. Although he did seem to fit the job, he was assigned more urgent research projects upon his arrival.

[3] Xiaojian Nie graduated and started to work for TrafficWare in the summer of 2003.

1.2 Organization

This manual is mainly written for potential programmers who would like to 1) use network objects defined in TNM and 2) contribute codes into TNM to extend its functionality. However, I tried not to make it like a list of classes with the description of all data members and operations. Rather, the emphasis is the overall architecture, the connection between classes and the explanation of important operations (e.g., the network loading function). Readers should refer to source codes (which were reasonably commented, in my opinion) for details not covered in this manual.

I shall first describe the hierarchy of TNM and explains how inheritance and polymorphism are realized in the next chapter. Chapter 3 introduces the major classes categorized in five groups: networks, nodes, links, O-D pairs and vehicles. Finally, in Chapter 4, I presented examples to demonstrate the usage of the TNM classes under different circumstance.

Chapter 2

The Hierarchy of TNM

2.1 Overview

TNM defines four major network classes:

- `TNM_SNET`: for static applications such as path flow estimator.
- `TNM_DNET`: for macroscopic dynamic applications such as dynamic network loading traffic assignment.
- `TNM_MNET`: for microscopic dynamic applications, such as studying vehicles' lane-changing behavior.
- `TNM_ProbeNet`: for stochastic applications such as solving the stochastic on-time arrival (SOTA) problem.

Figure 2.1 show the relationship of network classes. All the four network classes are derived from `TNM_SNET`, whose major data members include an object of *scanlist* and lists of *node*, *link* and *origin* objects. For narrative convenience, I call any of the four type of objects (namely *scanlist*, *node*, *link* and *origin*) as a building block object (BBO). *Scanlist* is an abstract class originally designed to realize polymorphism in shortest path (SP) calculation. Each of its derived class implements a labeling SP algorithm. Node and link are two underlying components of a network which are intricately connected with each other: each *node* has access to its direct connecting links whereas each *link* contains its tail node (from) and head node (to). Other than maintaining the topology of the network, *node* and *link* also play an important role in managing the transportation of commodities through networks. Origin is an imaginary facility building on an existing node which manage the demand of commodities originating from that node (e.g., where the commodities go and how they route in the network). In the next sections I shall discuss the structure of each BBO and its associated components.

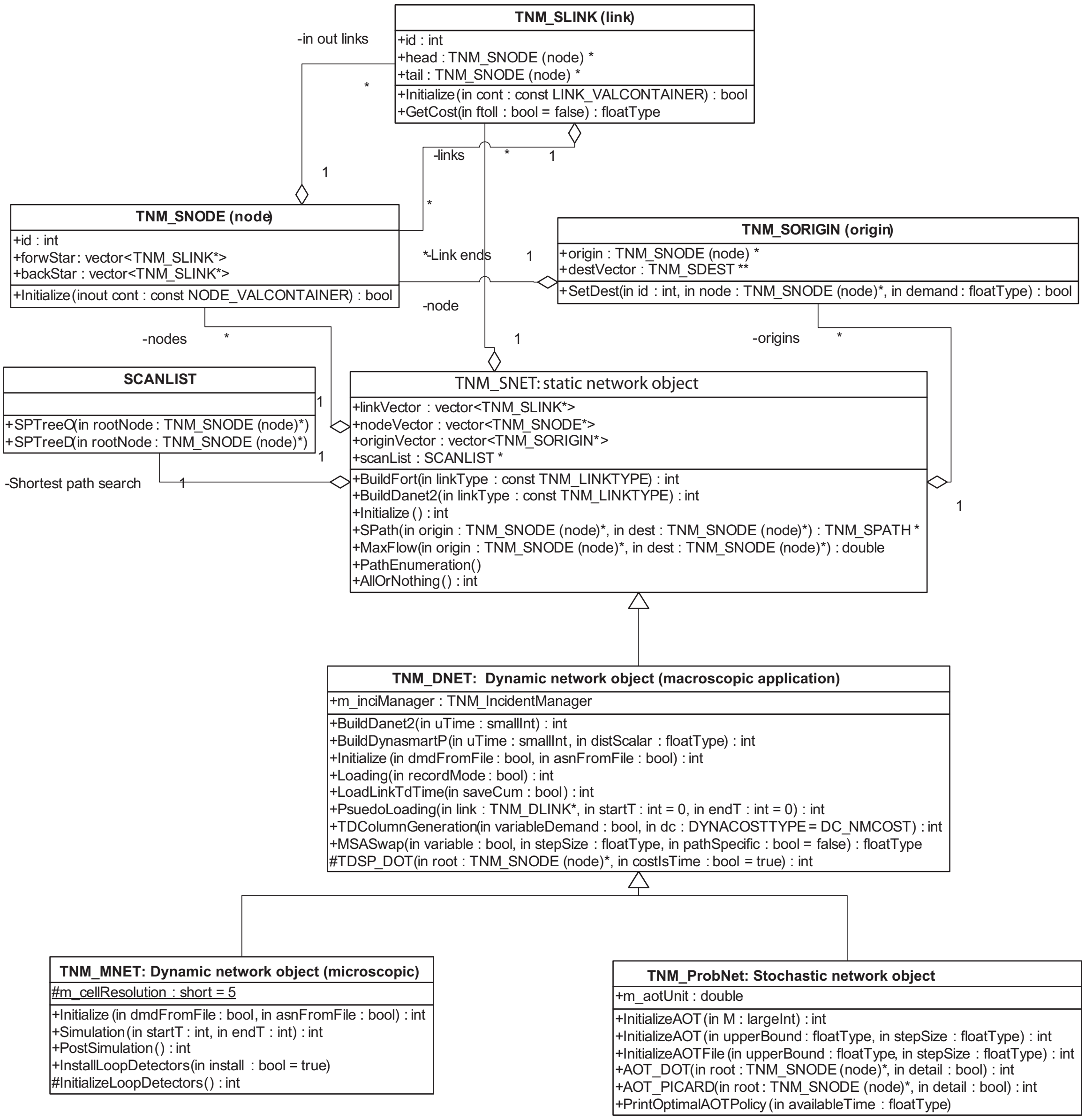


Figure 2.1: Overall Architecture of Network Classes

Each derived network class may only instantiate restricted types of each BBO, even though all stored pointers are cast into the associated base type for consistency (`ScanList`, `TNM_SNODE`, `TNM_SLINK`, `TNM_SORIGIN`). This brings about a little inconvenience in programming: a compulsory type conversion may have to be made to access the data and/or operations defined in derived objects. Theoretically, such conversion may be avoided if the base class defines all potential operations in form of virtual functions. TNM was not implemented in this way because I did not have a complete design blueprint from the onset.

As seen from Figure 2.1, the static network class defines operations for "building" and initializing networks from files of different formats, and solving basic network problems. `TNM_DNET` overrides the build and initialization functions in `TNM_SNET`. Its most important operations include the dynamic network loading and time-dependent shortest path calculation. `TNM_DNET` also contains incident handlers which are regarded meaningful only in dynamic context. Derived from `TNM_DNET`, `TNM_MNET` overrides its initialization function and provides a step-by-step simulation function mainly for visualization purpose. However, the all-in-one-step loading function defined in `TNM_DNET` can still be used in `TNM_MNET`. `TNM_ProbNet` is currently nothing more than a wrapper of several functions/data for solving the SOTA problem. I decided to separate these SOTA members from `TNM_DNET` in light of the potential needs for studying other stochastic problems.

2.2 Link Hierarchy

As shown in Figure 2.2, all link classes are derived from `TNM_SLINK`. Only defining static network connections `TNM_SLINK` is an abstract class, meaning that it should not be instantiated. `TNM_BPRLK` supports evaluating link costs using a BPR-type link performance function. Both `TNM_LINLK` and `TNM_ACHLK` are derived from `TNM_BPRLK` but they use different parameters. `TNM_CSTLK` defines a right-angle link performance function (constant + infinity) used in PFE.

`TNM_DLINK` is also an abstract base class supporting macroscopic dynamic applications. Among others, any macroscopic link must determine 1) how traffic moves from its entrance to its exit, 2) its supply, i.e., the maximum traffic that can be received from upstream links, and 3) its demand, i.e., the maximum traffic that is ready to leave for downstream links. From `TNM_DLINK` are derived seven macroscopic link classes which differ with each other in one or more of the above functions. For example, `TNM_DLINK_SQ` and `TNM_DLINK_LWR` implement respectively SQ (spatial queue) and LWR traffic flow models. Details of derived dynamic link classes will be discussed in the next chapter.

Figure 2.3 shows the relationship between micro- and macroscopic link classes and important components of dynamic links. The base link class for microscopic applications, `TNM_MLINK`, is derived from `TNM_DLINK`. Its most significant difference from its parent is the introduction of *lane*.

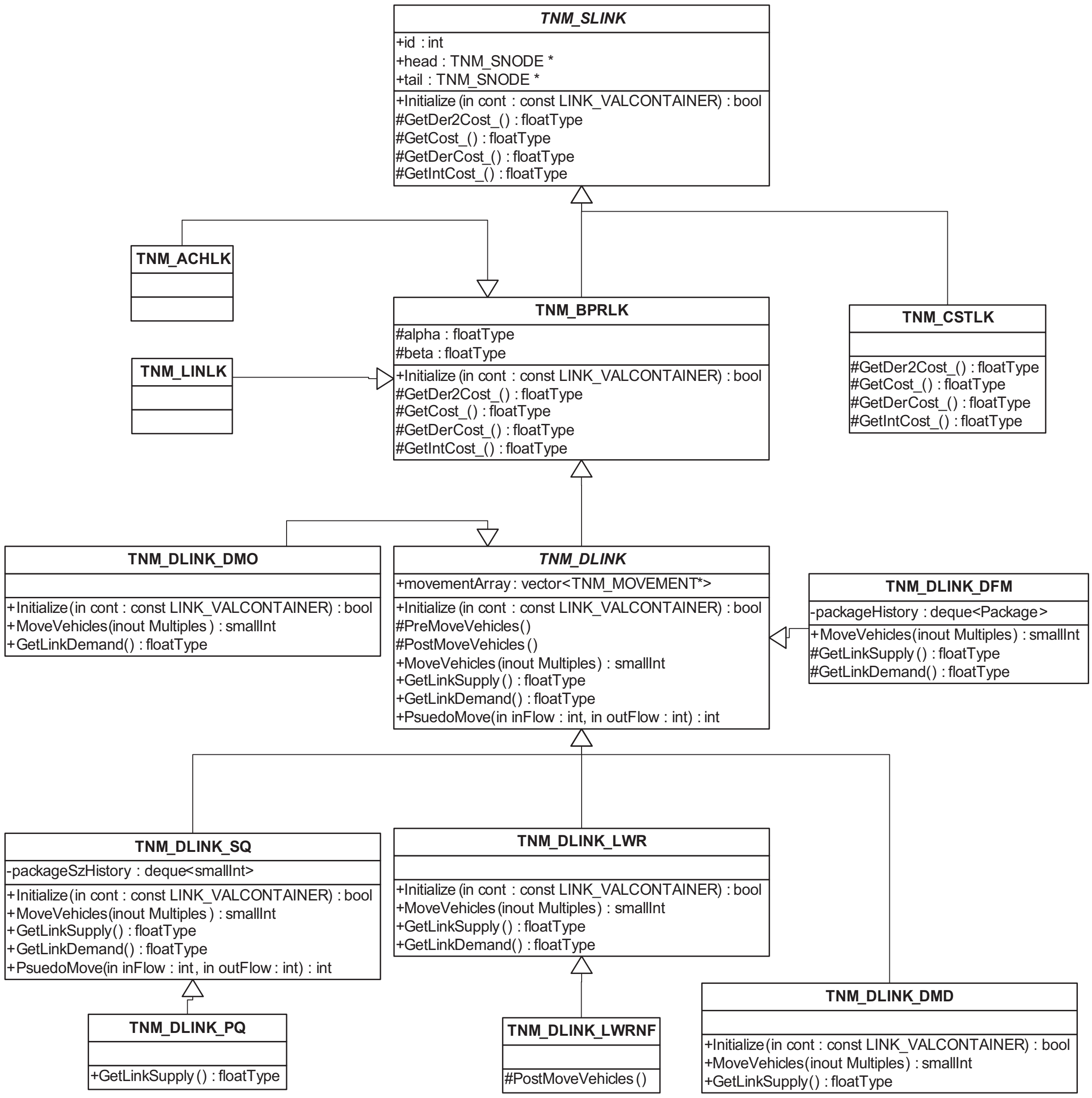


Figure 2.2: Link hierarchy I: static and dynamic links

Each lane, as seen, consists of a list of *cellular* and a loop detector manager. `TNM_MLINK_CA` implements the cellular automata model with lane-changing by overriding the `MoveVehicle` virtual function in `TNM_DLINK`.

The array of *movements* is an important data member in dynamic links. It offers a structure to classify and handle traffic heading toward different downstream links. Movements are heavily used in macroscopic links to calculate traffic flux through both controlled and uncontrolled nodes. In `TNM_MLINK` this role is weakened and movements merely contain corresponding lane groups rather than traffic. Each dynamic link is divided into a sequence of *cells*. Originally, the cell array is only defined in `TNM_DLINK_LWR`. But later it becomes a member of all dynamic links for cell-based animation of microscopic traffic simulation. Finally the Cellular class achieves two major functions: maintain the status of a space (whether it is occupied by a vehicle or not), and process vehicles' bids for the space.

Table 2.1 summarizes which types of links can be instantiated for a certain network class.

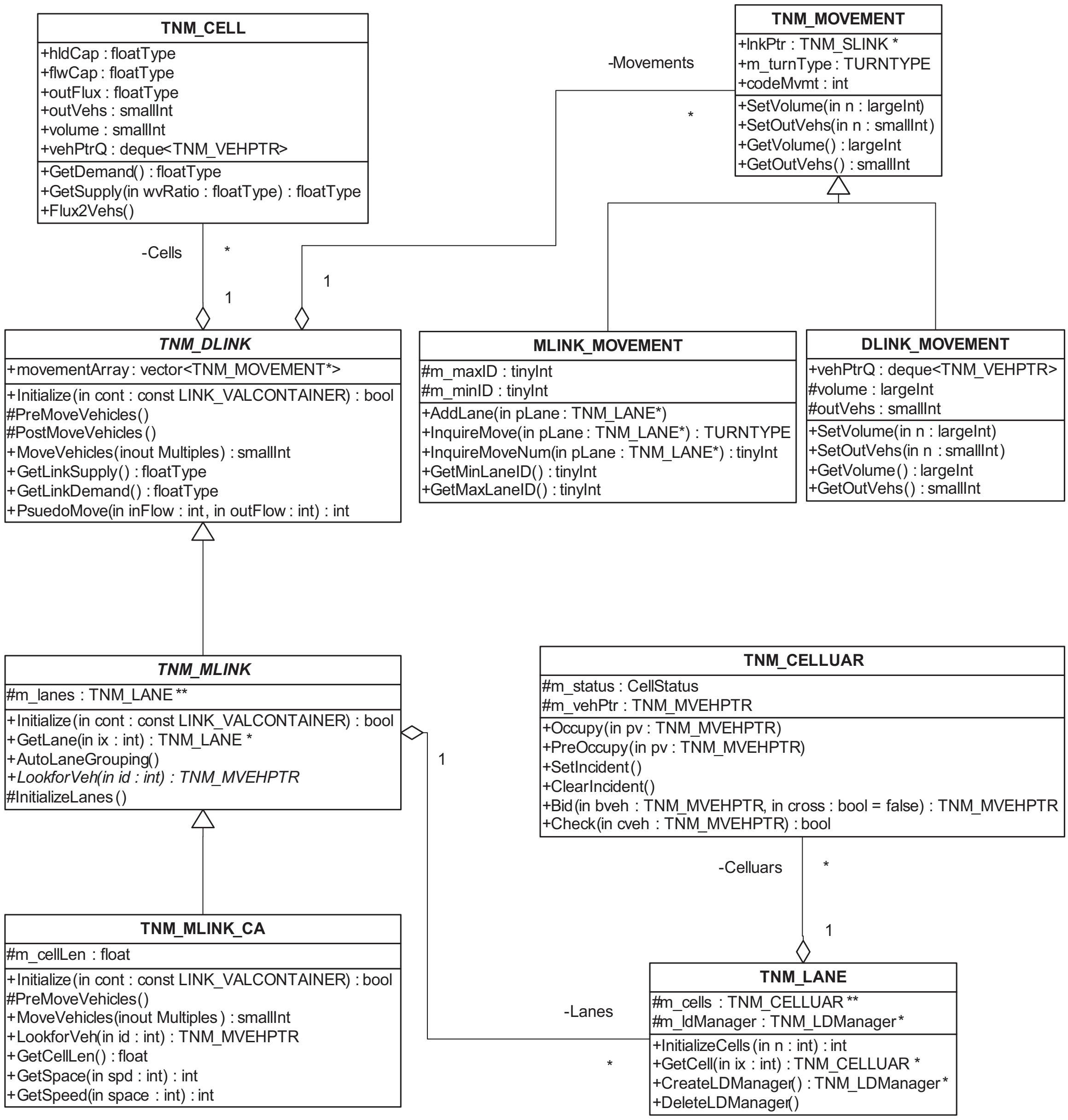


Figure 2.3: Link hierarchy II: macro- and microscopic links and their components

Table 2.1: The Applicability of Link Classes

	Desciption	TNM_SNET	TNM_DNET	TNM_MNET	TNM_ProbNet
TNM_SLINK	base static link	$\times$	$\times$	$\times$	$\times$
TNM_BPRLK	static link with BPR link performance function	$\surd$	$\times$	$\times$	$\times$
TNM_CSTLK	static link with constant link performance function	$\surd$	$\times$	$\times$	$\times$
TNM_LINLK	static link with linear link performance function	$\surd$	$\times$	$\times$	$\times$
TNM_ACHLK	static link with a quadratic link performance function	$\surd$	$\times$	$\times$	$\times$
TNM_DLINK	base dynamic link	$\times$	$\times$	$\times$	$\times$
TNM_DLINK_LWR	dynamic link with LWR traffic flow model	$\times$	$\surd$	$\times$	$\surd$
TNM_DLINK_PQ	dynamic link with point-queue traffic flow model	$\times$	$\surd$	$\times$	$\surd$
TNM_DLINK_SQ	dynamic link with spatial-queue traffic flow model	$\times$	$\surd$	$\times$	$\surd$
TNM_DLINK_LWRNF	dynamic link with LWR traffic flow model but FIFO is not reserved within link	$\times$	$\surd$	$\times$	$\surd$
TNM_DLINK_DMO	dynamic dummy origin link	$\times$	$\surd$	$\surd$	$\surd$
TNM_DLINK_DMD	dynamic dummy destination link	$\times$	$\surd$	$\surd$	$\surd$
TNM_DLINK_DFM	dynamic link with delay function traffic flow model	$\times$	$\surd$	$\times$	$\surd$
TNM_MLINK	base microscopic dynamic link	$\times$	$\times$	$\surd$	$\times$
TNM_MLINK_CA	microscopic link with cellular automata model	$\times$	$\times$	$\surd$	$\times$

2.3 Node Hierarchy

The hierarchy of node classes is given in Figure 2.4. Basically `TNM_SNODE` handles all static network operations. On top of it we derive an abstract class `TNM_DNODE` to support dynamic functions. The most important virtual function in `TNM_DNODE` is the MoveVehicle function which, together with the virtual function with the same name in the class of `TNM_DLINK`, consists of important building blocks of traffic simulation. However, in most macroscopic dynamic node types (either controlled or uncontrolled), polymorphism regarding to vehicle moving is realized through the virtual function UpdateBoundaryFlux. This function determines traffic flows between each pair of incoming-outgoing links of a node at a given time by considering the competition of different approaches. A supply-demand approach (see my dissertation) is used to formulate the UpdateBoundaryFlux function in uncontrolled node classes. `TNM_DNODE_FWJ` and `TNM_DNODE_FWJFI` are both for free-competition nodes (e.g., merge and diverge on freeways). The only difference is that the former does not enforce first-in-first-out principle. `TNM_DNODE_GCN` is designed to take into account the average effects of signal controls. It uses a UpdateBoundaryFlux function with a similar structure as in the `TNM_DNODE_FWJ` but reduces the capacity of different approaches based on a certain rule. For the two dummy types, dummy origins (`TNM_DNODE_DMO`) and destinations (`TNM_DNODE_DMO`), the MoveVehicle function is empty.

In controlled node classes (also derived from `TNM_DNODE`), the UpdateBoundaryFlux function mainly determines fluxes based on a built-in timing plan. So far four control types are defined: metered ramps (`TNM_CNODE_RM`), pre-timed signal control (`TNM_CNODE_PT`), adaptive control (`TNM_CNODE_AD`), and stop sign control (`TNM_CNODE_SP`).

In correspondence to microscopic simulation, a new abstract class `TNM_MNODE` is derived from `TNM_DNODE`. A new virtual function RequestRightOfWay is introduced to describe how vehicles will be assigned the right of way at a node. This function plays a similar role as UpdateBoundaryFlux for macroscopic nodes. Similarly, both controlled and uncontrolled types are considered. So far, only one type is implemented for each category: microscopic freeway junction (`TNM_MNODE_FWJ`) and microscopic stop sign control (`TNM_MNODE_STP`).

Table 2.2 summarizes which types of node may be instantiated for each network class.

2.4 Origin Hierarchy

Figure 2.5 shows how commodities flowing through networks are managed in TNM. All origin-destination pairs are organized using origins as roots, i.e., each origin stores a vector of destinations associated with positive demands. This forward star structure brings about some convenience when searching shortest paths. That is, a shortest path tree is established for an

Table 2.2: The Applicability of Node Classes

	Description	TNM_SNET	TNM_DNET	TNM_MNET	TNM_ProbNet
TNM_SNODE	base static ode	$\surd$	$\times$	$\times$	$\times$
TNM_DNODE	base dynamic node	$\times$	$\times$	$\times$	$\times$
TNM_DNODE_DMO	dynamic dummy origin node	$\times$	$\surd$	$\surd$	$\surd$
TNM_DNODE_DMD	dynamic dummy destination node	$\times$	$\surd$	$\surd$	$\surd$
TNM_DNODE_FWJ	dynamic node for freeway junction without FIFO enforced	$\times$	$\surd$	$\times$	$\surd$
TNM_DNODE_FWJFI	dynamic node for freeway junction with FIFO enforced	$\times$	$\surd$	$\times$	$\surd$
TNM_DNODE_GCN	dynamic node for generalized control	$\times$	$\surd$	$\times$	$\surd$
TNM_CNODE	bass dynamic controlled node	$\times$	$\times$	$\times$	$\times$
TNM_CNODE_PT	dynamic pretimed control node	$\times$	$\surd$	$\times$	$\times$
TNM_CNODE_RM	dynamic metered ramps	$\times$	$\surd$	$\times$	$\times$
TNM_CNODE_AD	dynamic adaptive control node	$\times$	$\surd$	$\times$	$\times$
TNM_CNODE_SN	dynamic stop sign control node	$\times$	$\surd$	$\times$	$\times$
TNM_MNODE	base microscopic dynamic node	$\times$	$\times$	$\times$	$\times$
TNM_MNODE_FWJ	microscopic freeway junction	$\times$	$\times$	$\surd$	$\times$
TNM_MNODE_CTL	base microscopic control node	$\times$	$\times$	$\times$	$\times$
TNM_MNODE_STP	microscopic stop sign control node	$\times$	$\times$	$\surd$	$\times$

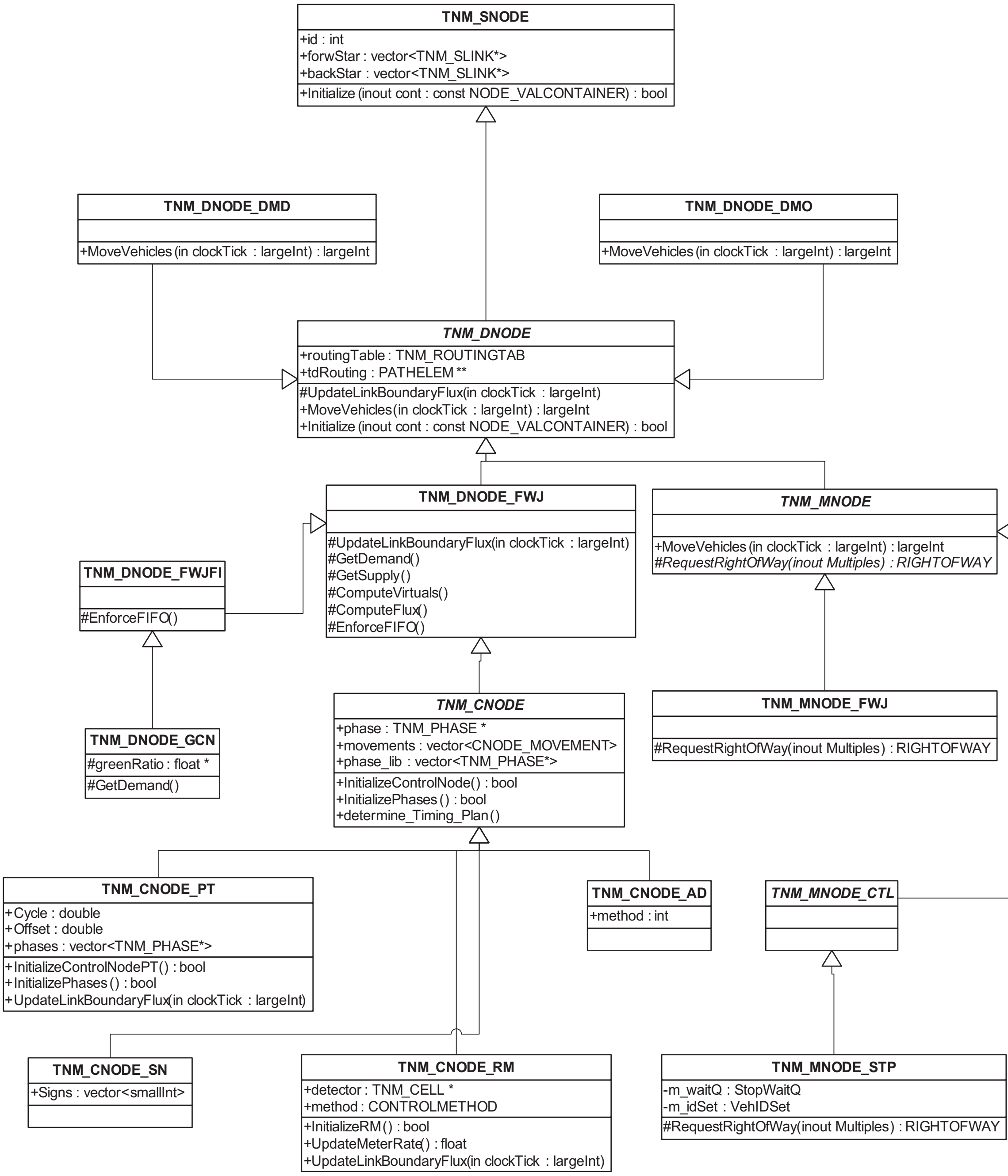


Figure 2.4: A hierarchy of node classes

origin then all shortest paths can be retrieved from the tree for each destination. Paths used by each O-D pair are stored in a TNM_SDEST object. A set of functions regarding to path-related operations are defined in TNM_SPATH class. These mainly include the evaluation of path costs of different types.

For each regime of networks (static, macroscopic or microscopic) corresponding origin and destination classes are defined, as seen from Figure 2.5. Static origin and destination classes are relatively simple and not designed to undertake important operations. The structure of dynamic destination class (`TNM_DDEST`) is more complicated because of the introduction of the assignment table. The assignment table consist of a sequence of assignment elements, each containing a certain amount of commodities associated with a departure time and possibly a path (if the path is predetermined). The underlying operation defined for `TNM_ASSIGNTAB` is the ReleaseVeh function which transforms commodities into a number of vehicular quanta (a certain rule is introduce to minimize the rounding errors) and then set their initial locations on the network. `TNM_ASSIGNTAB` also takes care of important functions related to (dynamic) traffic assignment. For example, flow swapping strategies based on either Euclidean projection and successive average have been built into `TNM_ASSIGNTAB`.

Microscopic origin and destination classes (`TNM_MORIGIN` and `TNM_MDEST`) are not substantially different from their macroscopic counterparts. For example `TNM_MDEST` only overrides one virtual function in `TNM_DDEST` to ensure only microscopic vehicular quanta are created when commodities are converted into quanta in ReleaseVeh.

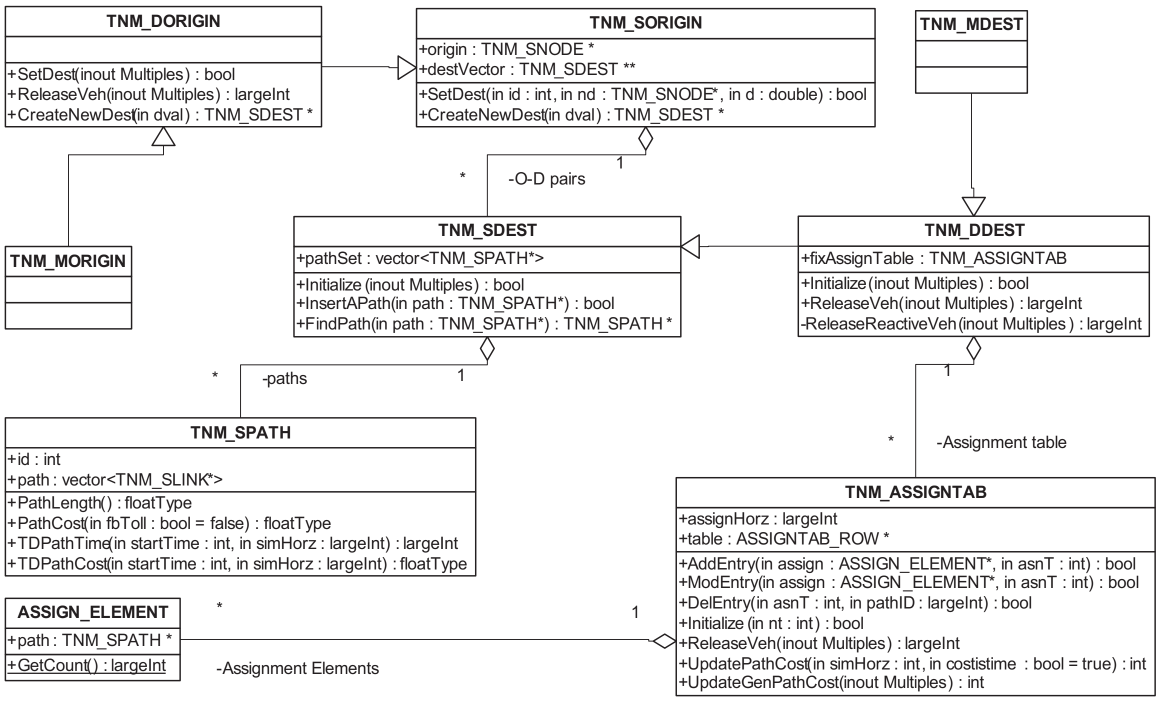


Figure 2.5: A hierarchy of origin classes and related components

2.5 Scanlist Hierarchy

`ScanList` was originally introduced into TNM to accommodate different labeling shortest path problem (SPP) algorithms. As seen in Figure 2.1, each network object will instantiate a specific `ScanList` object, when it constructs itself. The `ScanList` object determines which SPP algorithm will be used when a shortest path tree is to be searched. Users can redefine ScanList associated with any network object to choose a particular SPP algorithm best fitting their needs.

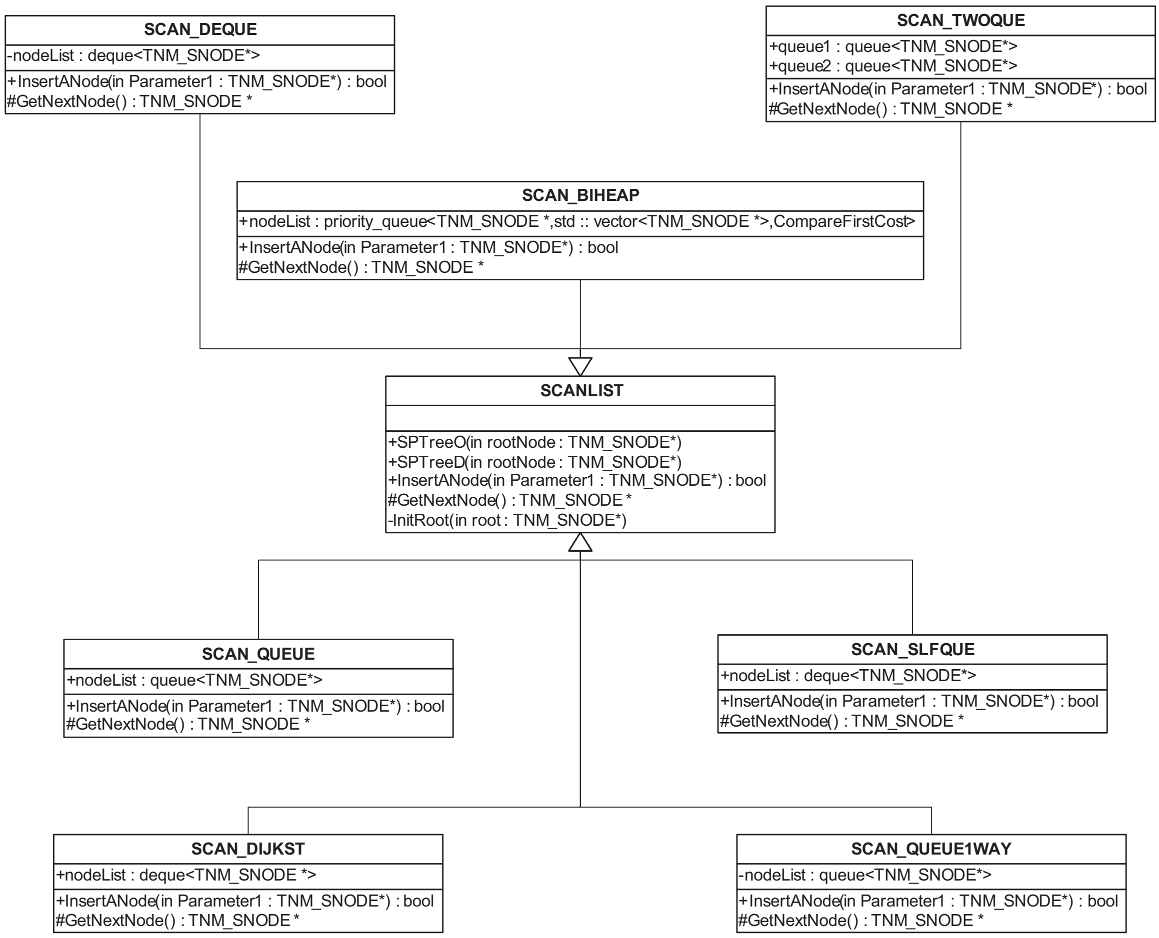


Figure 2.6: A hierarchy of scanlist

Figure 2.6 shows the structure of scanlist. Two major virtual functions are defined in the base class ScanList to realize the polymorphism: InsertANode and GetNextNode. Both are fundamental labeling operations. So far, seven derived classes are defined, each corresponding to a SPP algorithm. Recommended algorithms for transportation-type networks include `SCAN_DEQUE` and `SCAN_SLFQUE` (Both are label correcting algorithms). Heap-based label-setting algorithms (e.g., SCAN_BIHEAP) may work well for denser networks.

2.6 Vehicle Hierarchy

Before the microscopic simulation is implemented, the vehicle class is trivial and contains almost no operations. As the need of taking care of individual vehicles' behavior arises, more and more operations are added. Currently the TNM_VEHICLE class defines several basic functions managing 1) vehicles' moving from one link to the next, 2) vehicle's entering into link movements (for macroscopic only) and 3) vehicles' arrival at destination.

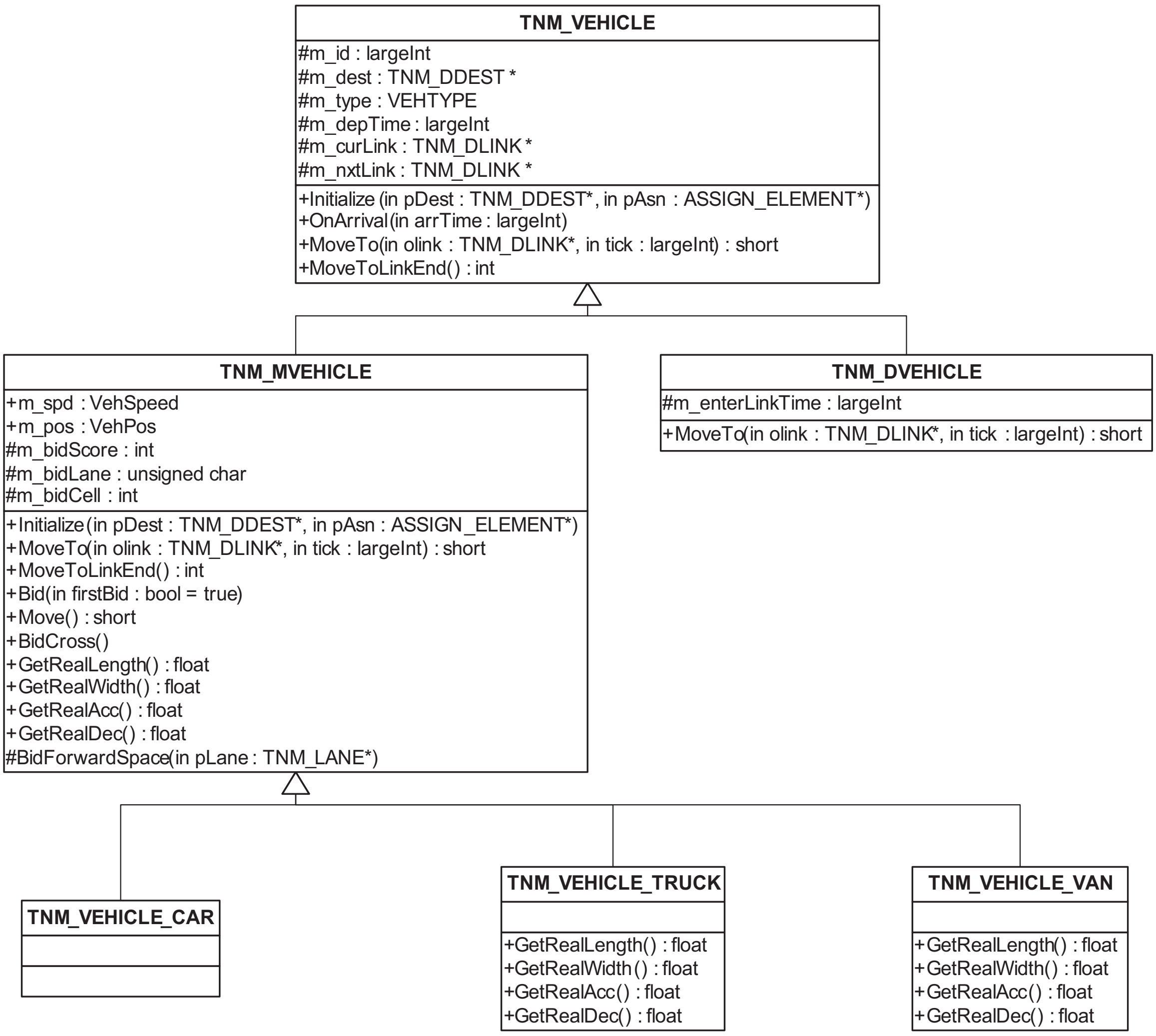


Figure 2.7: A hierarchy of vehicles

More complex vehicle behaviors are defined for microscopic vehicles (TNM_MVEHICLE). These mainly include the bidding functions such as Bid (for lane changing behavior) and BidCross (for merging behavior). However, the bidding system needs to be improved in two ways. First, in reality vehicles have to keep a certain space between those in front of itself based on a reaction time. This is not reflected in current implementation. The resulting road capacity is thus much higher than observed values. Second, the maximum deceleration is assumed to be infinity, i.e., vehicles can make a full stop right away to avoid crash. This is not realistic as well.

Some attempts are made to distinguish different vehicle types. We define three primary types: car, van and truck. In the current TNM, these vehicles differ with each other only for width and length. But I envision some more distinctions will be embedded in the future, such as the emission level, in-vehicle equipments, and so on.

2.7 Others

This section explains how incidents and loop detectors are managed in TNM. Each dynamic network has an incident manager which initialize itself (normally this means reading incident description from a disk file) when network's Initialize function is called. Users can always control whether or not these incidents should be considered using the EnableIncident switch. Apparently, there are two major operations associated with an incident: activate and deactivate it. For macroscopic simulation, activation/deactivation is realized through changing the capacity of a cell or a link (depending on link class). In microscopic networks, this is achieved by updating the occupancy status of the corresponding celluar.

The class of loop detector is currently designed for microscopic applications. Its main usage is for recording traffic counts and densities so that the fundamental diagrams can be plotted at given locations. This is originally intended to validate the cellular automata model, i.e., to see if realistic traffic flow properties can arise from the CA model.

Note that each lane has its own manager for loop detectors. This is to ensure that each vehicle has quick access to detector information.

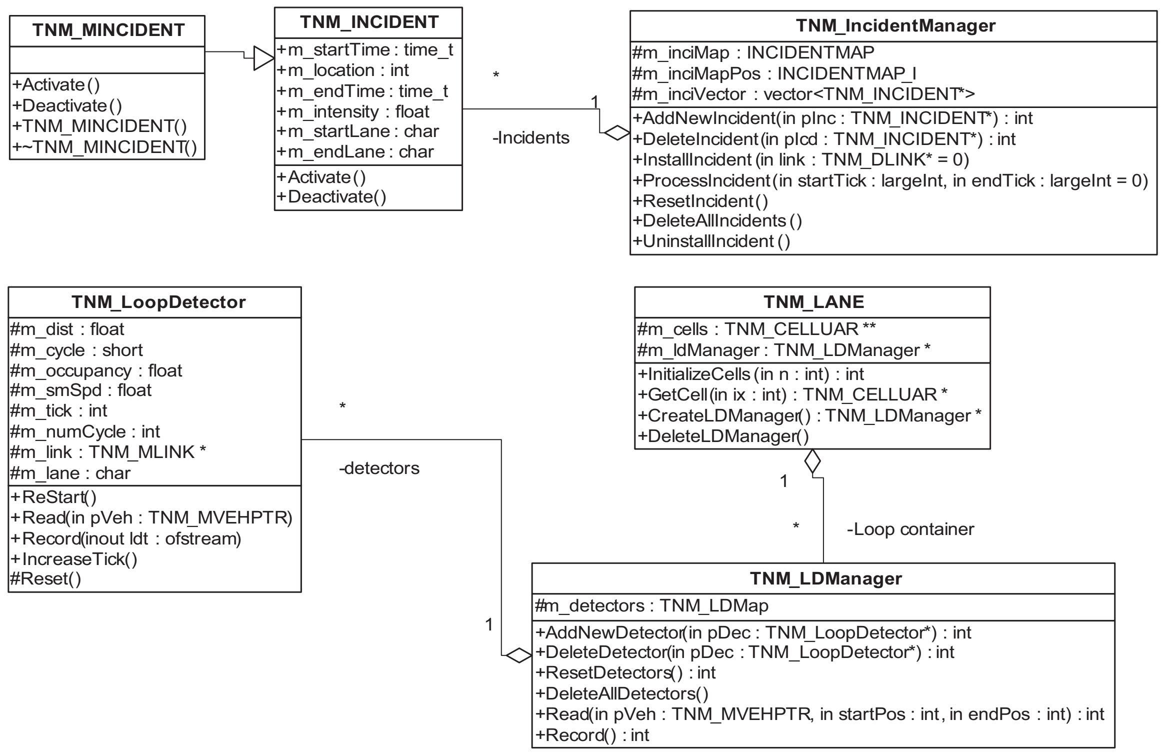


Figure 2.8: A hierarchy of incidents and loop-detectors

Chapter 3

Description of Major Classes

This chapter gives a detailed description of major classes defined in TNM. As mentioned before, the idea is not to list every data members and functions. Rather, only those considered both important for programming and not straightforward for understanding will be described. In all figures of class description, + stands for public, - for private and # for protected.

3.1 Link Classes

TNM_SLINK
+id : int +orderID : int +head : TNM_SNODE * +tail : TNM_SNODE * +cost : floatType +buffer : floatType * +markStatus : int +revLink : TNM_SLINK * +dummy : bool +oLinkPtr : ORGLINK * +pathInciPtr : vector<TNM_SPATH*> +guiLink : CGUILink *
+Initialize(in cont : const LINK_VALCONTAINER) : bool #ConnectFW() #ConnectBK() #DisconnectFW() #DisconnectBK() -CheckParallel() : bool +GetCost(in ftoll : bool = false) : floatType +GetDerCost(in ftoll : bool = false) : floatType +GetIntCost(in ftoll : bool = false) : floatType #GetDer2Cost_() : floatType #GetCost_() : floatType #GetDerCost_() : floatType #GetIntCost_() : floatType +CatchRevLink() : bool

Figure 3.1: Data members and Operations of `TNM_SLINK`

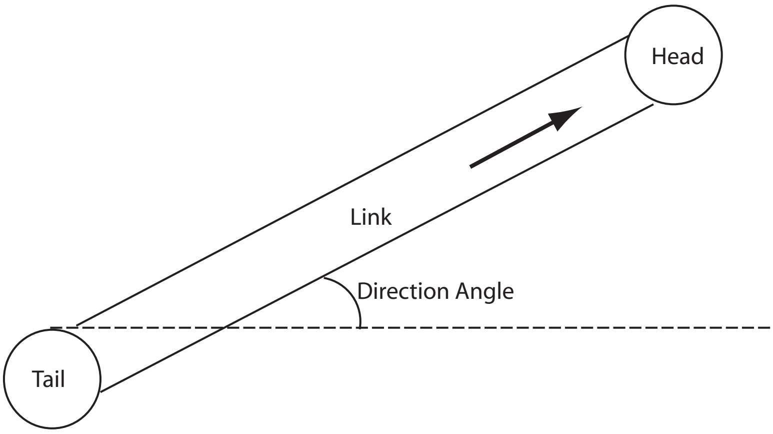


Figure 3.2: Basic Representation of a Link

CLASS TNM_SLINK PARENT CLASS: N/A SEE FIGURE 3.1

Attributes:

- id: an integer number used to identify this link.
- orderID: Represent the order of this link in linkVector, normally used in GUI.
- head: a pointer of the head (to) node of this link. See Figure 3.2.
- tail: a pointer of the tail (from) node of this link. See Figure 3.2.
- cost: represents the static traffic cost (often travel time) on this link. Shortest paths are always determined based on cost.
- buffer: a storage for temporary usage. User should allocate and release all link buffers using AllocateLinkBuffer and ReleaseLinkBuffer (see class `TNM_SNET`) respectively.
- markStatus: a status variable for temporary usage.
- revLink: a pointer of a link which forms the minimum cycle together with this link.
- dummy: a switch variable for temporary usage.
- oLinkPtr: a pointer to a link in an origin-based tree. This is intended for origin-based traffic assignment. .
- pathInciPtr: a vector storing all paths using this link. Used in Bell's iterative balancing algorithm.
- guiLink: a pointer for the graphic link object corresponding to this link. Only used by GUI..

Operations:

- CheckParallel: Called in Initialize to check whether or not a parallel link (i.e., with the same head and tail) exists.

GetCost: Calculate and return cost of a static link based on a link performance function. Marginal cost will added into the cost if `ftoll` is true.

GetIntCost: Calculate and return the integral of the link performance function. Marginal cost is considered if `ftoll` is true. Useful to evaluate Beckmann's objective function of the static traffic assignment problem.

GetDerCost: Calculate and return the derivative of the link performance function. Marginal cost is considered if `ftoll` is true. Useful to evaluate gradient.

CatchRevLink: Find the `revLink`.

`Overridables:`

Initialize: Initialize data members (link properties), make connections with head and tail nodes, and check parallel or duplicate (same id) links.

ConnectFW: Make connection with its tail node, called in Initialize.

ConnectBW: Make connection with its head node, called in Initialize.

DisconnectFW: Cut off connection with its tail node, called when link is destructed.

DisconnectBW: Cut off connection with its head node, called when link is destructed.

GetCost_: An empty virtual function.

GetIntCost_: An empty virtual function.

GetDerCost_: An empty virtual function.

GetDerCost_: An empty virtual function.

TNM_BPRLK
#alpha : floatType #beta : floatType
+Initialize(in cont : const LINK_VALCONTAINER) : bool #GetDer2Cost_() : floatType #GetCost_() : floatType #GetDerCost_() : floatType #GetIntCost_() : floatType

Figure 3.3: Data members and Operations of `TNM_BPRLK`

CLASS `TNM_BPRLK` **PARENT CLASS:** `TNM_SLINK` SEE FIGURE 3.3

`Attributes:`

alpha: a parameter of the link performance function.

beta: a parameter of the link performance function.

`Overridables:`

Initialize: First try to set alpha and beta from `lval` [1], then call Initialize in base class.

GetCost_: Calculate and return the cost evaluated from a BPR function taking the following form.

$$\tau = \tau_0[1 + \alpha(\frac{v}{c})^{\beta}] \tag{3.1}$$

where τ is travel time, τ_0 is free flow travel time, c and v are link capacity and volume respectively.

GetIntCost_: Calculate and return the integral of a BPR function.

GetDerCost_: Calculate and return the derivative of a BPR function.

GetDer2Cost_: Calculate and return the second derivative of a BPR function.

Other static link classes (e.g., `TNM_CSTLK`) will not be discussed because they are very similar to `TNM_BPRLK` in structure. Basically they implement different link performance functions (LPF) by overriding GetCost_, GetIntCost_, GetDerCost_, and GetDer2Cost_. Other type of LPF can be considered using the same way. If you want to use different parameters for a same functional form, try to pass the parameters through lval.par.

CLASS `TNM_DLINK` PARENT CLASS: `TNM_BPRLK` SEE FIGURE 3.4

`Attributes:`

inVehs: number of vehicles entering the link at the current loading interval.

unitFFT: travel time measured in unit of loading interval.

dirIndex: Relative approach direction. See Figure 3.5.

cumIn: Cumulative number of vehicular quanta entering the link by the current loading interval.

cumOut: Cumulative number of vehicular quanta leaving the link by the current loading interval.

tdTT: time-dependent link traversal times measured in unit of loading interval. Usually created from LoadLinkTdTime (see `TNM_DNET`).

tdCumIn: Cumulative number of vehicles entering the link for all loading intervals. Usually created from LoadLinkTdCum (see `TNM_DNET`).

tdCumOut: Cumulative number of vehicles leaving the link for all loading intervals. Usually created from LoadLinkTdCum (see `TNM_DNET`).

TNM_DLINK
+inVehs : smallInt +unitFFT : smallInt +dirIndex : short +cumIn : largeInt +cumOut : largeInt +tdTT : largeInt * +tdCumIn : largeInt * +tdCumOut : largeInt * +inVehQueue : deque<TNM_VEHPTR> +movementArray : vector<TNM_MOVEMENT*> +cellArray : vector<TNM_CELL*> +tdCost : floatType * +mRan : MyRandom * #wvRatio : floatType #wvRatioLastCell : floatType
+Initialize(in cont : const LINK_VALCONTAINER) : bool +GetAngle(in reverse : bool = false) : double #ConnectFW() #ConnectBK() #PreMoveVehicles() #TraceVehicles(inout out : ofstream, inout curClockTick : const largeInt) #ChooseNextLink(in vit : TNM_VEHPTR) : TNM_SLINK * #MoveToLinkEnd(in vit : TNM_VEHPTR, in timeTick : largeInt) : int #PostMoveVehicles() #InitializeCellArray() : bool +MoveVehicles(in currentTimeTick : largeInt, inout cum : ofstream, inout veh : ofstream, inout rpt : ofstream, in traceVeh : bool) : smallInt +GetLinkSupply() : floatType +GetLinkDemand() : floatType +GetFlowCapacity() : floatType +GetEstRandomness() : double +InitMvmtArray() +SetInstantTT() +DeleteVehicles() +OpenPsuedoMove() : int +PsuedoMove(in inFlow : int, in outFlow : int) : int +ClosePsuedoMove() : int +ResetPsuedoMove() +ResetLoopDetectors() +SaveDetectorInfo(inout ldt : ofstream) +DeleteLDManagers() +RecordLoopDetectors()

Figure 3.4: Data members and Operations of `TNM_DLINK`

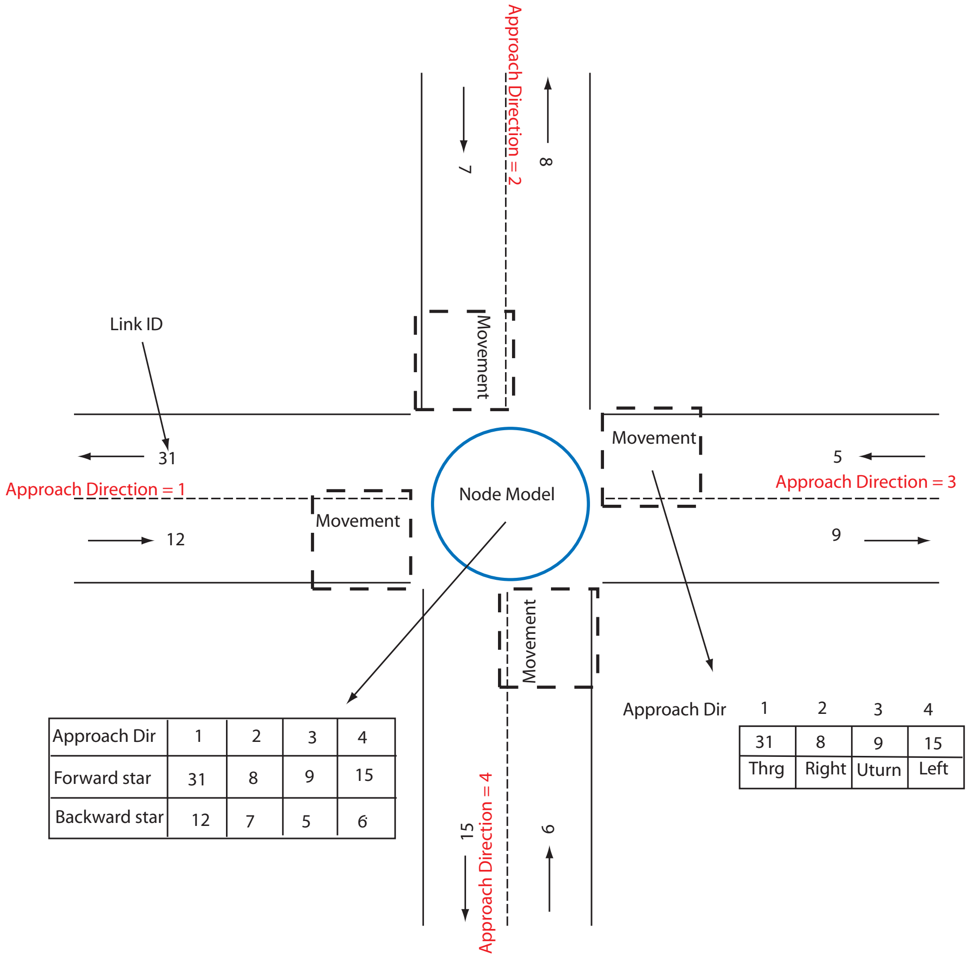
inVehQueue:	A queue temporarily holding all vehicular quanta entering the link at the current loading interval.
movementArray:	An array of movements. Each movement corresponds to a downstream link.
cellArray:	An array of cells. It is only populated in `TNM_DLINK_LWR`. For other type of links, this array is used in GUI.
tdCost:	Originally reserved for use in `TNM_ProbNet` to represent the probability density function.
mRan:	An object for random number generation. So far only used in `TNM_ProbNet`.
wvRatio:	The wave speed in the triangle fundamental diagram (the slope of the right hand side).
wvRatioLastCell:	The wave speed for the last cell. Because of the rounding error, the last cell of each link has a different size with the standard cell, thus has a different wave speed..

`Operations:`

- GetAngle: Calculate and return the direction angle of this link based on the coordinates of its head and tail nodes. See Figure 3.5.
- ChooseNxtLink: Choose the next link to visit for a given vehicle according to the vehicle's predetermined path, or if the vehicle is not assigned a path, obtain it from the `tdRoutingTable` of the head node.
- MoveToLinkEnd: Move vehicles into the movement of a link.
- TraceVehicles: Trace and record the time when a vehicle enters a link. Useful for postprocessing.
- SetInstantTT: Set the instanteous travel times. Mainly used in reactive traffic assignment, when the shortest paths needs to be recalculated based on the instanteous travel costs.

`Overridables:`

- ConnectFW: Make connection with the tail node, and allign movement with the relative approach direction.
- ConnectBW: Make connection with its head node, and allign movement with the relative approach direction.
- PreMoveVehicles: Handle necessary operations before `MoveVehicles` is called. So far only used in microscopic link types.
- PostMoveVehicles: Handle necessary operations after `MoveVehicles` is called.
- MoveVehicles: Move vehicles. For macroscopic links, this means taking out vehicles from `inVehQueue` and sending vehicles to movements; for microscopic links, it just an update of position (lane and cellular) of each vehicle.
- GetLinkSupply: Get the supply of a link, i.e., the maximum number of vehicles that a link can accommodate at the current loading interval.
- GetLinkDemand: Get the demand of a link, i.e., the maximum number of vehicles that can leave a link.
- GetFlowCapacity: Get the capacity of a link. The capacity of a link can be changed due to various reasons.
- GetEstRandomness: This is to estimate the impact of the current discretization scheme (i.e., loading interval and flow scalar).
- DeleteVehicles: Clear up all vehicles on this link. This function needs to be called in two cases: 1) a gridlock prevents loading is terminated successfully, and 2) when users terminate loading manually..
- OpenPsuedoMove: Prepare for psuedo moving. All 'Psuedo" moves are only used for post-procesing and visulization purpose.
- PsuedoMove: Reproduce time-dependent cell-based densities using cumulative in-out flows as inputs. Introduced for GUI..
- ResetPsuedoMove: Called in OpenPsuedoMove for resetting.
- ClosePsuedoMove: Release resource used in psuedo moving.
- ResetLoopDetectors: Rest recording information of all loop detectors on this link. Empty function for dynamic links.
- SaveDetectorInfo: Write current detector information to an external file. Empty functions for dynamic links.
- DeleteLDManagers: Release memory for loop detectors. Called in destruction. Empty function for dynamic links.



Approach Dir	1	2	3	4
Forward star	31	8	9	15
Backward star	12	7	5	6

Approach Dir	1	2	3	4
	31	8	9	15
	Thrg	Right	Uturn	Left

Figure 3.5: A Link-Node Connection Map For Dynamic Application

RecordLoopDetectors: Record current detector information to an internal file. Empty functions for dynamic links.

Figure 3.5 shows how node and link are connected with each other for dynaic applications. For each node, a fixed integer number called approach direction is assigned to each of its approach [2]. Approach directions for a node are determined after all its incoming and outging links have respectively called their ConnectBW and ConnectFW functions (these should usually be done in *building* a network). Approach directions are indexed from 1, 2,..., until n, and correspond to the storing positions of links in the forward/backward stars. They also correpond to the storing position of a movement (which is identified by its associated downstream link) in the `movementArray`. Note that each link's `dirIndex` stores the approach direction relative to its tail node. As such, given an incoming-outgoing link pair for any node, you can immiediately retrieve

[2]An approach is identified by any of node which directly links the node of interest

the associated movement information (i.e., geetting approach direction from the outgoing link, then retrieve movement from the incoming link based on the corresponce relationship).

Polymorphism of dynamic links's behavior is mainly captured through three functions: GetLinkSupply, GetLinkDemand, MoveVehicles. For example, if you want to create a link able to accommodate infinte number of vehciles, then you simply let its GetLinkSupply function return a very large number in any case.

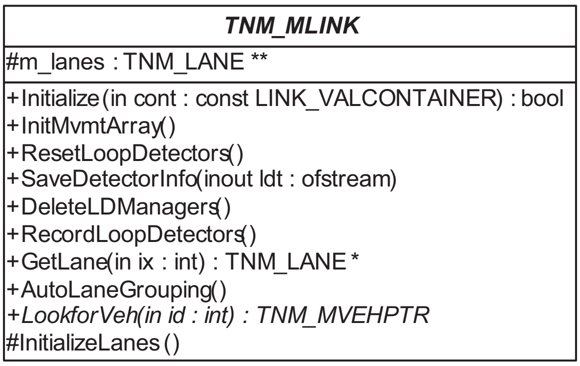


Figure 3.6: Data members and Operations of `TNM_MLINK`

CLASS `TNM_MLINK` PARENT CLASS: `TNM_DLINK` SEE FIGURE 3.6

`Attributes:`

m_lanes: an array of lane pointers.

`Operations:`

InitializeLanes: Alloate memory for lanes.

GetLane: Get a pointer to a lane using index (indexed from 0, which is the leftmost one).

AutoLaneGrouping: Automatically assign lanes to each movement (i.e., left, right or through). So far users cannot specify lane grouping. This function may be added in the future.

`Overridables:`

Initialize: Call base Initialize function, then allocate memory for lanes.

LookForVehicle: Find out a vehicle on the link using its ID.

CLASS `TNM_CELLUAR` PARENT CLASS: `NULL` SEE FIGURE 3.7

`Attributes:`

TNM_CELLUAR
#m_status : CellStatus #m_vehPtr : TNM_MVEHPTR
+IsEmpty() : bool +IsPreOccupied() : bool +IsOccupied() : bool +IsIncidented() : bool +SetEmpty() +Occupy(in pv : TNM_MVEHPTR) +PreOccupy(in pv : TNM_MVEHPTR) +SetIncident() +ClearIncident() +GetStatus() : CellStatus +Bid(in bveh : TNM_MVEHPTR, in cross : bool = false) : TNM_MVEHPTR +Check(in cveh : TNM_MVEHPTR) : bool +GetVehicle() : TNM_MVEHPTR

Figure 3.7: Data members and Operations of `TNM_CELLUAR`

m_status: the status of the cellular, either occupied (by a vehicle), pre-occupied (bid by a vehicle), empty, or incident).

m_vehPtr: the pointer of the vehicle which either occupy or preoccupy the cellular.

`Operations:`

InitializeLanes: Alloate memory for lanes.

SetEmpty: Empty the cellular.

Occupy: Occupy the cellular with the input vehicle.

Preoccupy: Bid the cellular for the input vehicle.

SetIncident: Set the cellular status as incident, which will block the lane.

Bid: The input vehicle bid for the cellular. If this cellular has been bid before, then the one with higher `bidScore` will get the position, and the other will be kicked out and find another position.

Check: This is to test whether or not the position can be occupied. No real bidding process involved here, called in LookForwardSpace.

3.2 Node Classes

CLASS `TNM_SNODE` PARENT CLASS: `NULL` SEE FIGURE 3.8

`Attributes:`

id: an integer number used to identify this node.

xCord: the x coordinate of the location of the node, measured in feet.

yCord: the y coordinate of the location of the node, measured in feet.

```
TNM_SNODE
---------------------------------------------
+id : int
+xCord : largeInt
+yCord : largeInt
+forwStar : vector<TNM_SLINK*>
+backStar : vector<TNM_SLINK*>
+pathElem : PATHELEM *
+buffer : floatType *
+scanStatus : smallInt
+dummy : bool
+pthHeap : HEAP3NODE *
+heapDone : short
+guiNode : CGUINode *
+attachedOrg : int
+attachedDest : int
+kspPathElem : KSPMAP *
+kspPEvector : KSPPATHELEM *
---------------------------------------------
+Initialize(inout cont : const NODE_VALCONTAINER) : bool
+IsControlled() : bool
+DestDummyAttached() : TNM_SLINK *
+OrigDummyAttached() : TNM_SLINK *
+SearchMinInLink(in list : SCANLIST*)
+SearchMinOutLink(in list : SCANLIST*)
+SearchOutLink(in list : SCANLIST*)
+InitPathElem()
+BuildOutHeap(inout recBin : queue<HEAP3NODE *>)
+BuildOutHeap_R(inout recBin : queue<HEAP3NODE *>)
```

Figure 3.8: Data members and Operations of `TNM_SNODE`

forwStar:	a vector of pointers for links starting at the node.
backStar:	a vector of pointers for links ending at the node.
pathElem:	a structucture to record shortest path tree. It contains shortest distance (cost) and next link on the path (via).
buffer:	a temporary storage for external use.
scanStatus:	a temporary variable.
dummy:	a temporary switch variable.
pthHeap:	A heap for retrieving k-shortest paths. Only used in Eppstein's KSPR algorithm.
heapDone:	Mark whether or not the heap has been built. Only used in Eppstein's KSPR algorithm.
guiNode:	a pointer for the graphic node object corresponding to this link. Only used by GUI..
attachedOrg:	number of origins which are built on this node. In fact, this number cannot be larger than 1 since we use a one-to-all structure to store O-D pairs..
attachedDest:	number of O-D pairs (i.e., destination objects) which are built on this node.
kspPathElem:	a structure to sort K-shortest paths. Used in labeling-setting KSPR algorithms.
kspPEVector:	a structure to record K-shortest paths. Used in all labeling KSPR algorithms.

`Operations:`

DestDummyAttached: Check whether or not this node is adjacent to a node of type `TNM_DMDND`. Used mainly in GUI.

OrgDummyAttached: Check whether or not this node is adjacent to node of type `TNM_DMOND`. Used mainly in GUI.

SearchMinInLink: Search all links in backward star, update `pathElem` based on shortest distance. It is a basic operation in all (labeling) shortest path algorithms, and is called from `ScanList`.

SearchMinOutLink: Search all links in forward star, update `pathElem` based on shortest distance. It is a basic operation in all (labeling) shortest path algorithms, and is called from `ScanList`.

SearchOutLink: Search all link in forward star, designed for solving the maximum flow problem.

InitPathElem: Reset path element (via = NULL, cost = infinity).

BuildOutHeap: Create a heap using reduced costs of its outgoing links. Only used in Eppstein's KSPR algorithm.

BuildOutHeap_R: Create a heap using reduced costs of its incoming links. Only used in Eppstein's KSPR algorithm.

`Overridables:`

Initialize: Set data members.

IsControled: Check whether or not the node is controlled.

TNM_DNODE
+routingTable : TNM_ROUTINGTAB +tdRouting : PATHELEM **
#AssociateMvmtWthDwnLnks() : bool -ArrangeConnectingLinks() : int -GetTurnType(in nxDir : int) : TURNTYPE #MoveAVehBetweenLnks(in ilink : TNM_DLINK*, in olink : TNM_DLINK*, in clockTick : largeInt) #MoveVehsThroughNode(in clockTick : largeInt) : largeInt #UpdateLinkBoundaryFlux(in clockTick : largeInt) +MoveVehicles(in clockTick : largeInt) : largeInt +GetDirectionNum() : int +GetRealNodeBound(inout point1 : POINT, inout point2 : POINT, in dIndex : int, in laneWidth : int, in scale : int = 1) : bool +Initialize(inout cont : const NODE_VALCONTAINER) : bool +InitLoading() : int +PostLoading() : int

Figure 3.9: Data members and Operations of `TNM_DNODE`

CLASS `TNM_DNODE` PARENT CLASS: `TNM_SNODE` SEE FIGURE 3.9

`Attributes:`

routingTable: A routing table designed for reactive traffic assignment. Its stores a shortest path for each different destination. That is, any vehicle, when it arrives upon the node, it can request from the tabel a current shortest path using its final destination as a key.

tdRouting: a vector of path elements. It is for time-dependent shortest path search..

`Operations:`

- ArrangeConLks: Internal function. Used to sort approach directions according the direction angle. See Figure 3.10.
- GetTurnType: Internal function. Used to assign a turn type for any given incoming-outgoing link pair.
- MoveVehBtLks: Move a vehicle from an incoming link to an outgoing link.
- MoveVehThND: Move vehicles from all incoming links to all outgoing links.
- GetDirectionNum: Return number of approaches.
- GetRealNDBound: Calculate the real bound of the node based on the layout and width of connecting links. Used in GUI to plot realistic network sketches.

`Overridables:`

- Initialize: Call base function.
- AstMvtWtDwnLks: Initialize data members in movements of all incoming links (e.g., `turnType`, `linkID`), and set `dirIndex` of all outgoing links.
- MoveVehicles: Critical virtual function to realize polymorphism of dynamic traffic flow through nodes.
- UpdateLkBdFlux: Critical virtual function to determine the flux through a node at a given time. In most cases, it is sufficient to override this function.
- InitLoading: Prepare for dynamic network loading, e.g., allocate memory for data only needed in loading.
- PostLoading: Postprocess for dynamic network loading. It's a good place to clean up temporary memory and so on.

`TNM_DNODE` will automatically determine the turn type for each movement based on a two-step procedure. In the first step, all approaches are sorted in an ascending order according to the angle between each approach and horizontal line (called approach angle). See Figure 3.10 for an illustration. In the second step, turn type is assigned based on this order. For the four-approach example in Figure 3.10, we set $1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4$ and $4 \rightarrow 1$ as right turn, and $1 \rightarrow 4, 2 \rightarrow 1, 3 \rightarrow 2$ and $4 \rightarrow 3$ as left turn. For T-intersections or intersection with more than four approaches, different rules are used to set turn types. Table 3.1 summarizes these rules.

CLASS `TNM_DNODE_FWJ` PARENT CLASS: `TNM_DNODE` SEE FIGURE 3.11

`Attributes:`

- demand: an array to store demands of all incoming links.
- supply: an array to store supplies of all outgoing links.

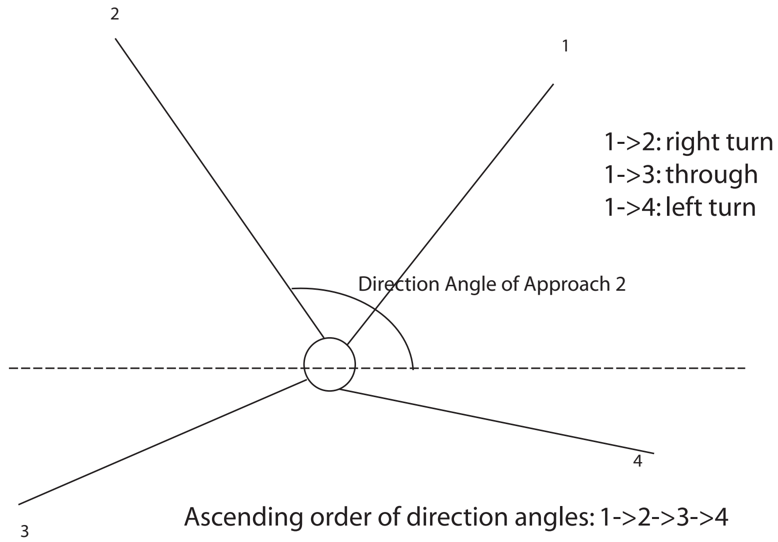


Figure 3.10: Sorting Approaches at a Node Based on Direction Angles

Table 3.1: Rules to Determine Movement Turn Types

Approach numbers	1	2	3	4	5	6	7
3	U	R	L	N/A	N/A	N/A	N/A
4	U	R	T	L	N/A	N/A	N/A
5	U	R	T	T	L	N/A	N/A
6	U	R	R	T	L	L	N/A
7	U	R	R	T	T	L	L

U: U-turn, L: Left turn, R: Right turn, T: Through
Note that the table shows the turn type of the movements associated with the incoming link from approach 1.

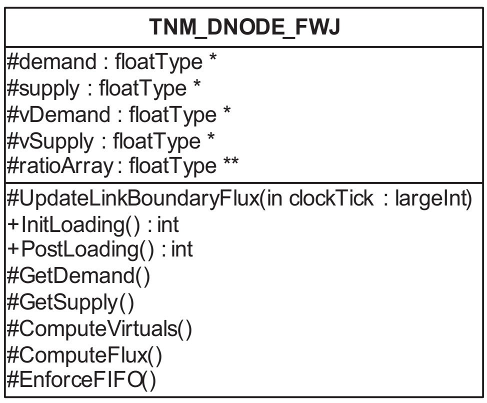


Figure 3.11: Data members and Operations of `TNM_DNODE`

vDemand: an array to store virtual demands of all incoming links.

vSupply: an array to store virtual supplies of all incoming links.

ratioArray: Ratio of movement traffic to approach traffic. A two-dimensional array.

`Operations:`

GetDemand: Set `demand` array.

GetSupply: Set `supply` array.

ComputeVirtuals: Compute vDemand and vSupply.

ComputeFlux: Compute traffic flowing through node at each time interval.

`Overridables:`

EnforceFIFO: Define the behavior related to first-in-first-out principle. It is empty in this class, but will be overridden in `TNM_DNODE_FWJFI` to enforce FIFO.

TNM_MNODE
#AssociateMvmtWthDwnLnks() : bool +MoveVehicles(in clockTick : largeInt) : largeInt *#RequestRightOfWay(in clockTick : largeInt, in incomlinkIndex : int, in pmov : TNM_MOVEMENT*, in pVeh : TNM_MVEHPTR) : RIGHTOFWAY* #PostROWRequest(in pVeh : TNM_MVEHPTR)

Figure 3.12: Data members and Operations of `TNM_DNODE`

CLASS `TNM_MNODE` PARENT CLASS: `TNM_DNODE` SEE FIGURE 3.12

`Overridables:`

RequestROW: Like the UpdateLinkBoundaryFlux function in dynamic node, this function is designed to determine which vehicles should be transmitted through nodes at a given time interval. To be overridden to realize desirable behavior.

PostROWRequest: This function is introduced to handle operations which may incur after requesting right-of-way.

Notice: Description of control node types will added later on.

3.3 Origin Classes

CLASS `TNM_SORIGIN` PARENT CLASS: `NULL` SEE FIGURE 3.13

`Attributes:`

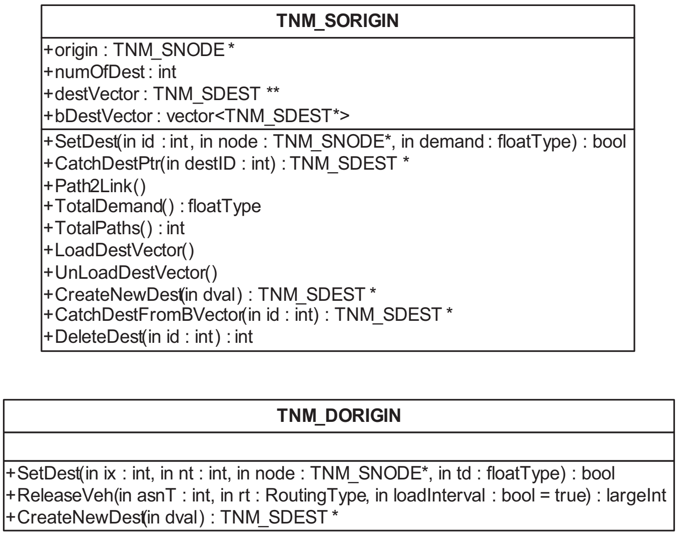


Figure 3.13: Data members and Operations of `TNM_SORIGIN` and `TNM_DORIGIN`

- origin: the pointer on which the origin stands.
- destVector: an array of destination pointers.
- bDestVector: a vector of destination pointer. Only used by GUI.

`Operations:`

- SetDest: Initialize a destination from its index.
- CatchDestPtr: Retrieve a destination pointer from its ID (i.e., the node on which the destination stands).
- Path2Link: Internal function. Aggregate link flows from path flows.
- TotalDemand: Calculate total demands for this origin.
- TotalPaths: Calculate total number of used paths (those stored in `pathSet` of `TNM_SDEST`) for this origin.
- LoadDestVector: Move all pointer stored in destVector into bDestVector, then delete destVector. Only used in GUI.
- UnLoadDestVector: Move all pointers stored in bDestVector into destVector, then clear up bDestVector. Only used in GUI.
- CatchDestFromBV: Retrieve a destination pointer from bDestVectorr.
- DeleteDest: Delete a destination from bDestVector.

`Overridables:`

- CreateNewDest: Create a new destination and add it into bDestVector. Only used in GUI.

Class `TNM_DORIGIN` Parent class: `TNM_SORIGIN` See Figure 3.13

Operations:

ReleaseVeh: Call ReleaseVeh function for each of its associated destinations to generate vehicles.

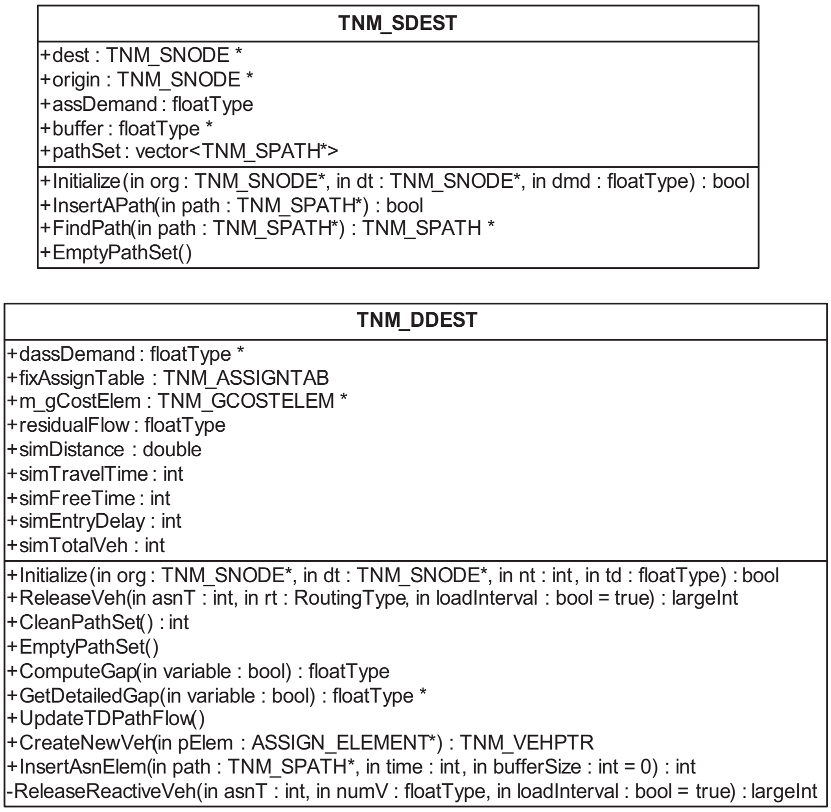


Figure 3.14: Data members and Operations of TNM_SDEST and TNM_DDEST

CLASS TNM_SDEST PARENT CLASS: NULL SEE FIGURE 3.14

Attributes:

dest: the pointer of the node on which the destination stands.

origin: the `origin` member of the associated origin..

assDemand: the demand of the commodity for the O-D pair.

buffer: a temporary storage for external use.

pathSet: a set of paths used by this O-D pair.

Operations:

FindPath: Find a path from the path set using the pointer.

AddAPath: Add a path to the tail of the path set. If a path with the same topology exists in the path set, then the existing path will be moved to the end of the path vector.

`Overridables:`

EmptyPathSet: Delete all paths in path set.

Initialize: Initialize all data members for the destination.

Class `TNM_DDEST` Parent class: `TNM_SDEST` See Figure 3.14

`Attributes:`

dassDemand: an array of time-dependent demands of commodities.

fixAssignTab: the assignment table specifying time-dependent path flow pattern, i.e, how much flow uses which path and when.

m_gCostElem: a structure to hold general cost information, i.e., punctual arrival time window, schedule cost penalty.

residualFlow: the difference between the total demand and total released vehicles. Such a difference exists because vehicles can only carry flows in unit of flow scalar.

simDistance: total simulated distance of all vehicles in network loading.

simTravelTime: total simulated travel times of all vehicles in network loading.

simFreeTime: total simulated free flow travel times of all vehicles in network loading.

simEntryDelay: total simulated entry delay (i.e., delayed incurred on dummy origin links) of all vehicles in network loading.

simTotalVeh: total simulated vehicles in network loading.

`Operations:`

ReleaseVeh: Generate vehicles which have a predefined path for the current time period, then release them into network. It will directly call ReleaseVeh defined in `TNM_AssignTab`.

ReleaseReactiveVeh: Generate vehicles which have no predefined paths. These vehicles will request a path from the RoutingTable stored in a node object. Only used in *Reactive* or *Mixed* mode..

CleanPathSet: Delete paths carrying trivial flows.

ComputeGap: Return the equilibrium gap for the current path usage.

UpdateTDPathFlow: Aggregate the total flows on each path for all times.

InsertAsnElem: Insert a new assignment element (i.e., path + flow + departure time) into `fixAssignTab`.

`Overridables:`

CreateNewVeh: Allocate memory for new vehicles. Will be overridden for different vehicle types..

TNM_SPATH
+id : int +buffer : floatType * +markStatus : short +path : vector<TNM_SLINK*> +m_refAsnElem : ASSIGN_ELEMENT * -pathCount : largeInt = 0 -pathBufferSize : smallInt = 0 -idManager : IDManager
+TNM_SPATH() +TNM_SPATH(in i : int) +GetLinkNum() : int +IsConnected() : bool +PathCostS() : floatType +PathLength() : floatType +PathCost(in fbToll : bool = false) : floatType +TDPathTime(in startTime : int, in simHorz : largeInt) : largeInt +TDPathCost(in startTime : int, in simHorz : largeInt) : floatType +FDPathCost(in fbToll : bool = false) : floatType +FDPathCostIntersect(in fbToll : bool = false, in inter : bool = true) : floatType +ToLinkFlow() +ToLinkFlow(in flow : floatType) +PutMarkOnLinks(in mark : tinyInt) +MarkLink(in mark : bool = true) +FormCycle(in link : TNM_SLINK*) : bool +AugmentFlow() +SetID() +SetID(in i : int) +SetPathBufferSize(in bf : int)

Figure 3.15: Data members and Operations of `TNM_SPATH`

CLASS `TNM_SPATH` PARENT CLASS: `NULL` SEE FIGURE 3.15

`Attributes:`

- id: an integer number identifying the path.
- buffer: a temporary storage for external use.
- markStatus: a temporary status variable.
- m_refAsnElem: a temporary reference to an assignment element.
- pathBufferSize: static, the size of buffer.
- idManager: a class that manages path ID.

`Operations:`

- TNM_SPATH(): Constructor. Use this one if you want an ID be assigned for the path by idManager.
- TNM_SPATH(int i): Constructor. Use this one if you think this path is for temporary usage and you don't need an ID.
- GetLinkNum: Get total number of links contained in the paths.
- IsConnected: Check whether or not the path is connected.
- PathCostS: Sum up all `cost` members on its links.

PathLength:	Sum up all `len` members on its links.
PathCost:	Call GetCost for all its links, them sum up all `cost` members.
TDPathTime:	Calculate time-dependent path travel times.
TDPathCost:	Calculate time-dependent path travel costs, normally this incurs schedule costs.
FDPathCost:	Sum up all returns from GetDerCost of all its links.
FDPathCostInt:	Sum up returns from GetDerCost of a subset of its links (usually the intersection of the current path and another path).
ToLinkFlow:	Aggregate path's `flow` to link's `flow`.
ToLinkFlow(float f):	Aggregate the input `f` into link's `flow`.
PutMarkOnLinks:	Set all links' `markStatus` as the given value.
MarkLink:	Set all links' `dummy` as the given value.
FormCycle:	Add a path to the tail of the path set. If a path with the same topology exists in the path set.
AugmentFlow:	A special function for solving maximum flow problem.
SetID:	Let idManager pick an ID.
SetID(int i):	Enforce the ID as the given value.
SetPathBufferSize:	Specify the buffer size for the path.

TNM_ASSIGNTAB
+assignHorz : largeInt +table : ASSIGNTAB_ROW * +refAsnElem : ASSIGN_ELEMENT *
+AddEntry(in assign : ASSIGN_ELEMENT*, in asnT : int) : bool +ModEntry(in assign : ASSIGN_ELEMENT*, in asnT : int) : bool +DelEntry(in asnT : int, in pathID : largeInt) : bool +GetRowSum(in asnT : int) : floatType +Initialize(in nt : int) : bool +ReleaseVeh(in dest : TNM_DDEST*, in asnT : int, in loadInterval : bool = true) : largeInt +UpdatePathCost(in simHorz : int, in costistime : bool = true) : int +UpdateGenPathCost(in simHorz : int, in dest : TNM_DDEST*, in costistime : bool = true) : int +GetPathCost(in path : TNM_SPATH*, in asnT : int, in simHorz : int) : floatType +GetGenPathCost(in path : TNM_SPATH*, in asnT : int, in simHorz : int, in dest : TNM_DDEST*) : floatType +ProjectNegCost_V(in dest : TNM_DDEST*) : int +ProjectNegCost_F(in dest : TNM_DDEST*) : int +SmartFlowSwap_V(in stepSize : floatType, in dest : TNM_DDEST*) : floatType +SmartFlowSwap_F(in stepSize : floatType, in dest : TNM_DDEST*) : floatType +MSAFlowSwap_V(in stepSize : floatType, in dest : TNM_DDEST*, in pathSpecific : bool = false) : floatType +MSAFlowSwap_F(in stepSize : floatType, in dest : TNM_DDEST*) : floatType +FromFlowToVeh(in dest : TNM_DDEST*, in variable : bool) : int +PostAssignmentCheck(in dest : TNM_DDEST*, in variable : bool) : int +FindZeroAssignElem(in dest : TNM_DDEST*) +UpdatePathFlow() +DelEmptyElements() +EmptyTable() -ComputeQuadObj() : floatType -ComputeQuadObj(in i : int) : floatType

Figure 3.16: Data members and Operations of `TNM_ASSIGNTAB`

Class TNM_ASSIGNTAB Parent class: NULL See Figure 3.16

Attributes:

assignHorz:	number of assignment intervals for this O-D pair.
table:	a two-dimensional table, each row of the table corresponds to an assignment interval, which consist a sequence of assignment elements identified by path.
refAsnElem:	a temporary reference for an assignment element.

Operations:

AddEntry:	Add an assignment element to a row.
ModEntry:	Modify an assignment element at a row .
DelEntry:	Delete an assignment element from a row.
GetRowSum:	Get the sum of flows for a row.
ReleaseVeh:	Create new vehicles then release them onto network.
UpdatePathCost:	Update `cost` member in all elements' path objects with current time-dependent path travel times.
UpdateGenPathCost:	Update `cost` member in all elements' path objects with current time-dependent general path costs, usually containing schedule costs.
GetPathCost:	Update and return `cost` for a given path at a given departure time interval.
GetGenPathCost:	Update and return general `cost` (with schedule cost) for a given path at a given departure time interval.
ProjectNegCost_V:	Perform projection to get a new assignment. Used in all projection-type DTA algorithm. V stands for the case with departure time choice.
ProjectNegCost_F:	Same to ProjectNegCost_V except F stands for the case without departure time choice.
SmartFlowSwap_V:	Use day-to-day swapping strategy to get a new assignment, with departure time choice.
SmartFlowSwap_F:	Use day-to-day swapping strategy to get a new assignment, without departure time choice.
MSAFlowSwap_V:	Use method of successive average to get a new assignment, with departure time choice.
MSAFlowSwap_F:	Use method of successive average to get a new assignment, without departure time choice.
FromFlowToVehicle:	Convert flows to vehicular quanta. Obsolete.
PostAssignmentCheck:	Post check assignment results to 1) ensure the flow conservation and 2) remove elements with tiny flows.
FindZeroAssignElem:	Find all assignment elements carrying zero or near-zero flows..
UpdatePathFlow:	Update `flow` member in the path object by summing flows in all assignment elements.

DelEmptyElem:	Delete elements with zero near-zero flows.
EmptyTable:	Delete all assignment elements.
ComputeQuadObj:	Called in ProjectNegCost_V to solve the quadratic program.
ComputeQuadObj(i):	Called in ProjectNegCost_F to solve the quadratic program.

3.4 Vehicle Classes

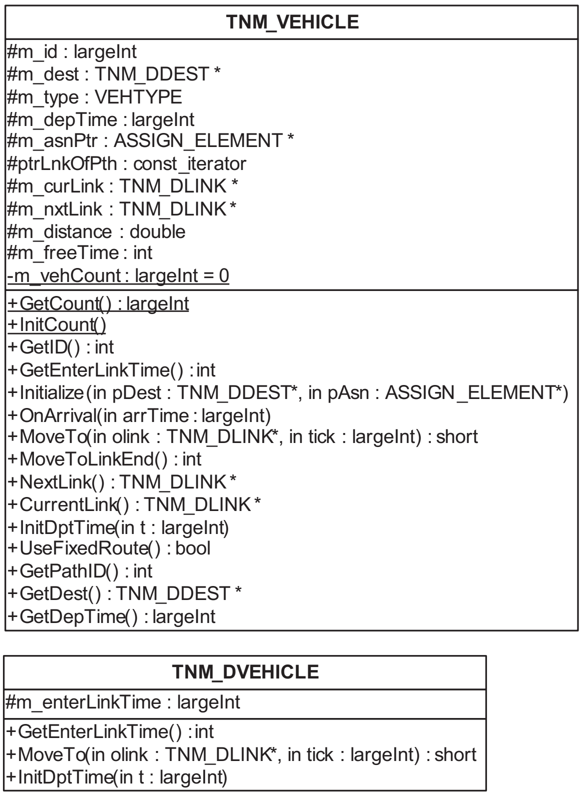


Figure 3.17: Data members and Operations of TNM_VEHICLE and TNM_DVEHICLE

CLASS TNM_VEHICLE PARENT CLASS: NULL SEE FIGURE 3.17

Attributes:

m_id:	an integer number identifying the vehicle. Usually the total count of vehicles is picked as ID upon the generation of the vehicle.
m_dest:	the pointer of the destination of the vehicle.
m_depTime:	departure time interval, in unit of loading interval.

m_asnPtr: the pointer of the assignment element from which the vehicle is released. If a vehicle is not released from such an element (i.e., the vehicles does not have a predetermined path), this is NULL.

ptrLnkOfPth: the position of current link on the assigned path. This pointer helps find the next link on the path whenever the vehicle enters a link.

m_curLnk: the link on which the vehicle currently stay.

m_nxtLnk: the next link the vehicle will visit.

m_distance: accumulative travel distance in miles.

m_freeTime: accumulative free flow travel times.

m_vehCount: static, total number of vehicles generated.

`Operations:`

GetCount: Get the total number of the vehicles generated so far.

InitCount: Reset the vehicle count to zero. Called before network loading starts.

NextLink: Find out and return the next link the vehicle should visit. How this is performed depends on whether or not the vehicle follows a predefined path or not.

CurrentLink: Return the current link.

UseFixedRoute: Whether or not the vehicle follows a predetermined path.

`Overridables:`

GetEnterLinkTime: Get the time interval when the vehicle enters the current link. Empty function in the base class.

OnArrival: Accommodate potential operations needed to be done upon arrival at the destination.

MoveTo: Accommodate potential operations need to be done when a vehicle moves from one link to another.

MoveToLinkEnd: Move vehicles into corresponding movements according to the next link to visit. This is used for macroscopic simulation.

CLASS TNM_DVEHICLE PARENT CLASS: TNM_VEHICLE SEE FIGURE 3.17

`Attributes:`

m_enterLinkTime: the time interval when the vehicle enters the current link.

`Overridables:`

GetEnterLinkTime: Return `m_enterLinkTime`.

CLASS TNM_MVEHICLE PARENT CLASS: TNM_MVEHICLE SEE FIGURE 3.18

TNM_MVEHICLE
+m_spd : VehSpeed +m_pos : VehPos #m_bidScore : int #m_lcStatus : LCStatus #m_bidLane : unsigned char #m_bidCell : int #m_cross : bool #m_agress : char
+Bid(in firstBid : bool = true) +Move() : short +BidCross() +GetRealLength() : float +GetRealWidth() : float +GetRealAcc() : float +GetRealDec() : float +EnableCross(in cross : bool = true) +IsCross() : bool +GetBidScore() : int +GetSpeed() : char +GetAggressiveness() : char +GetLaneChangeStatus() : LCStatus +UpdateParameters() +GetMaxBidScore() : int +GetTolerateSpeed() : short +GetVehLength() : short +EmptyCurrentPos(in cross : bool = false) +EmptyBidPos(in cross : bool = false) +PreOccupyBidPos(in cross : bool = false) +FullStop() +GetBidCell() : int +IsLaneChange() : bool +ArriveLinkExit() : bool +CheckIncident() : int #CheckDetector() #CheckDetector(in cross : bool) #InitializeBid () : bool #ConsiderExit() : bool #InitializeScore () #FinalizeScore() #CurrentSpdAcceptable(in spd : short = -1) : bool #OccupyBidPos (in cross : bool = false) #BidForwardSpace(in pLane : TNM_LANE*) #LookForwardSpace() : TURNTYPE #LookForwardSpace(in pLane : TNM_LANE*) : int #NextLane() : TNM_LANE * #CurrentLane() : TNM_LANE * #LeftLane() : TNM_LANE * #RightLane() : TNM_LANE * #CurrentCell() : TNM_CELLUAR * #GetMovement() : MLINK_MOVEMENT *

Figure 3.18: Data members and Operations of `TNM_MVEHICLE`

`Attributes:`

m_spd: a structure describing vehicle's speed, including `m_magnitude` (speed measured in unit of cellular length), `m_maxSpd` (maximum speed), `m_maxAcc` (maximum acceleration). The length of cellular depends on link free flow speed v_f, loading interval δt and the `m_cellResolution` in `TNM_MNET` n_{cr}, i.e.,.

$$l_{ca} = \frac{v_f \delta t}{n_{cr}} \tag{3.2}$$

m_pos: a structure describing vehicle's position, including `m_lane` (which lane), `m_cell` (which cell), `m_len` (the length of the vehicle in unit of cellular length of the current link, m_nxtLen (the length of the vehicle in unit of cellular length of the next link.

m_bidScore: bidding score. This scores determines the power of a vehicle when bidding for a position. It will change according to a certain rule.

m_lcStatus: lane changing status, whether or not the vehicle want to change to left, right or nowhere.

m_bidLane: the lane the vehicle bids for.

m_bidCell: the cell the vehicle bids for.

m_cross: whether or not the vehicle will change link at the current move.

m_agress: a behavioral parameter, measuring drivers' aggressiveness (ranging from 1 to 10). This parameter directly affects `m_bidScore`.

`Operations:`

EnableCross: Update vehicles' cross status.

UpdateParameters: Update vehicles' link-specific parameters, such as `m_len` and `m_nxtLen`.

GetMaxBidScore: Calculate and return the maximum possible bid score. The following formula is used.

$$b_{\max} = \text{m\_maxSpeed} \times (20 + \text{m\_aggress}) \tag{3.3}$$

GetTolerateSpeed: Calculate and return the minimum speed a vehicle can tolerate without seeking lane-changing. The following formula is used.

$$v_{tol} = \frac{\text{m\_maxSpeed} \times \text{m\_aggress}}{\max\{\text{m\_aggress}\} = 10} \tag{3.4}$$

EmptyCurrentPos: Leave the position the vehicle currently occupies.

EmptyBidPos: Remove the bid from the position the vehicle prefers to go.

PreoccupyBidPos: Put the bid on the position the vehicle prefers to go.

FullStop: Enforce the vehicle to make a full stop.

IsLaneChange: Check the lane changing status.

ArriveLinkExit: Whether or not the vehicle will leave the current link after the current time interval if it travels at the maximum speed.

CheckIncident: Check if any position currently occupied by this vehicle subject to an incident.

CheckDetector: Check if vehicle will pass a loop detector by moving to its next position, if so, let the loop detector record the passage.

InitializeBid:	Initialize status before bidding for a new position. Among other, this function will update the bidding score and determine whether or not a lane-changing is preferred.
ConsiderExit:	Whether or not the vehicle wants to consider the mandatory lane-changing at the link exit (because each movement is assigned a group of lanes). This is mainly determined by the distance to the exit point and `m_agress`.
InitializeScore:	Update bidding score before bidding.
FinalizeScore:	Update bidding score after moving. Usually bidding score will be reset to zero if the desirable movement is realized.
OccupyBidPos:	Move to the position the vehicle currently bid for.
BidForwardSpace:	Bid the cells forward until the furthest possible position (i.e., if the vehicle travels at maximum possible speed) is reached.
LookforwardSpace:	Look at the forward space to decide if a lane-changing is desirable.
NextLane:	Check and return the lane on the next link the vehicle will visit.
CurrentLane:	Check and return the pointer of the current lane.
LeftLane:	Check and return the lane on the left side of the current lane.
RightLane:	Check and return the lane on the right side of the current lane.
CurrentCell:	Check and return the current cell pointer.
GetMovement:	Get the pointer of the movement.

`Overridables:`

Bid:	Bid for the best position on the current link. Other vehicles will have to rebid if the bidding of this vehicle invalidates their original bids.
Move:	Move from current position to the bid position.
BidCross:	Bid for the best position on the next link.
GetRealLength:	Get the length of the vehicle in feet, not in unit of cellular length.
GetRealWidth:	Get the width of the vehicle in feet, not in unit of cellular length.
GetRealAcc:	Get the acceleration of the vehicle in feet/sec^2.
GetRealDec:	Get the deceleration of the vehicle in feet/sec^2.

3.5 Network Classes

CLASS `TNM_SNET` PARENT CLASS: `NULL` SEE FIGURE 3.19

`Attributes:`

```
TNM_SNET
---------------------------------------------------------------
+networkName: string
+numOfNode : int
+numOfLink : int
+numOfOrigin : int
+numOfOD : int
+linkVector : vector<TNM_SLINK*>
+nodeVector : vector<TNM_SNODE*>
+originVector : vector<TNM_SORIGIN*>
+destNodeVector : vector<TNM_SNODE*>
+scanList : SCANLIST *
#buildStatus : int
#initialStatus : int
#linkCostScalar : floatType
#m_regime : TNM_REGIME
#recBin_KSP : queue<HEAP3NODE*>
-nodeBufferSize : int
-linkBufferSize : int
-pathBufferSize : int
-destBufferSize : int
---------------------------------------------------------------
+BuildDanet2(in linkType : const TNM_LINKTYPE) : int
+BuildFort(in linkType : const TNM_LINKTYPE) : int
+UnBuild() : int
+CheckBuildStatus(in noteBuilt : bool) : int
+GetNetworkPath() : string
+GetNetworkName() : string
+ResetCoordinates(inout pMaxX : long, inout pMaxY : long)
+ResetLinkLengthFromCoord()
+Initialize() : int
+UnInitialize() : int
+CheckInitialStatus(in noteInitial : bool) : int
+CreateNewLink(in lval : const LINK_VALCONTAINER) : TNM_SLINK *
+CreateNewNode(in nval : const NODE_VALCONTAINER) : TNM_SNODE *
+CreateSOrigin(in nodeID : int, in nd : int) : TNM_SORIGIN *
#AllocateNewLink(inout pType : const TNM_LINKTYPE) : TNM_SLINK *
#AllocateNewNode(inout nType : const TNM_NODETYPE) : TNM_SNODE *
+ChooseSPAlgorithms(in type : DATASTRUCT)
+SPath(in origin : TNM_SNODE*, in dest : TNM_SNODE*) : TNM_SPATH *
+SPath(in orgID : int, in destID : int) : TNM_SPATH *
+EPN_KSP(in origin : TNM_SNODE*, in dest : TNM_SNODE*, in loop : bool, in K : int, in overRate : double, in range : int) : queue<TNM_SPATH*>
+EPN_KSP(in orgID : int, in destID : int, in loop : bool, in K : int, in overRate : double, in range : int) : queue<TNM_SPATH*>
+BFM_KSP(in orgID : int, in destID : int, in K : int) : queue<TNM_SPATH*>
+DIJK_KSP(in orgID : int, in destID : int, in K : int) : queue<TNM_SPATH*>
+PICA_KSP(in orgID : int, in destID : int, in K : int) : queue<TNM_SPATH*>
+UpdateSP(in rootNode : TNM_SNODE*)
+UpdateSPR(in rootNode : TNM_SNODE*)
+RetrieveSPathFromTree(in origin : TNM_SNODE*, in dest : TNM_SNODE*) : TNM_SPATH *
+MaxFlow(in origin : TNM_SNODE*, in dest : TNM_SNODE*) : double
+PathEnum_DFS(in from : TNM_SNODE*, in to : TNM_SNODE*, inout path : TNM_SPATH, inout pathSet : vector<TNM_SPATH*>)
+PathEnumeration()
+AllOrNothing() : int
+AllOrNothing(in origin : TNM_SORIGIN*) : int
+AllOrNothingP()
+AllOrNothingP(in origin : TNM_SORIGIN*)
+ColumnGeneration() : int
+ColumnGeneration(in origin : TNM_SORIGIN*) : int
+MultiColGeneration(in K : int, in overRate : double, in range : int) : int
+ReposeSP()
+BuildLink2PathPtr()
+ClearLink2PathPtr()
+Path2Link()
+UpdateLinkCost(in toll : bool = false)
+UpdatePathCost(in toll : bool = false)
+Reset()
+MinLinkCover() : queue<TNM_SLINK*>
+TotalDemand() : double
+TotalPaths() : int
+ClearZeroPaths()
+ClearPaths(in crit : double)
+CatchODPtr(in orgID : int, in destID : int) : TNM_SDEST *
+CatchNodePtr(in id : int) : TNM_SNODE *
+CatchLinkPtr(in id : int) : TNM_SLINK *
+CatchLinkPtr(in tail : int, in head : int) : TNM_SLINK *
+CatchOrgPtr(in id : int) : TNM_SORIGIN *
+AllocateNodeBuffer(in size : int) : int
+DeleteALink(in id : int) : int
+DeleteANode(in id : int) : int
+DeleteAnOrigin(in id : int) : int
+DeleteDestinations(in id : int) : int
+UpdateANode(in id : int, in nval : __unnamed_6f8f8c15_1) : TNM_SNODE *
+AllocateLinkBuffer(in size : int) : int
+AllocatePathBuffer(in size : int) : int
+AllocateDestBuffer(in size : int) : int
+ReleaseNodeBuffer() : int
+ReleaseLinkBuffer() : int
+ReleasePathBuffer() : int
+ReleaseDestBuffer() : int
#EmptyPathSet()
```

Figure 3.19: Data members and Operations of TNM_SNET

networkName: the name of the network. It contains the full path and base file name (i.e., c:/test/mynet). This parameter should be specified whenever a network is constructed.

linkVector: a vector storing all link pointers.

nodeVector: a vector storing all node pointers.

originVector: a vector storing all origin pointers.

scanList: a `ScanList` object for static shortest path search.

buildStatus: whether or not the network is built, 1 – built, 0 – not..

initialStatus: whether or not the network is initialized, 1 – initialized, 0 – not..

linkCostScalar: a scalar to change the unit of link cost. Must be set before building to be effective.

m_regime: the network type (dynamic, static, or microscopic).

recBin_KSP: a temporary variable used in Eppstein's KSPR.

nodeBufferSize: the size of buffer in all node objects.

linkBufferSize: the size of buffer in all link objects.

pathBufferSize: the size of buffer in all path objects.

destBufferSize: the size of buffer in all destination objects.

`Operations:`

BuildDanet2: Build a network defined in the DANET2 format. Building is an important procedure. Most operations can only be used on a built network. After building, all links, nodes, origins and destinations should have been created and stored in the corresponding vectors. .

BuildFort: Build a network defined in the FORT format. A detailed description of FORT format can also be found in the appendix of Zhang, Nie & Shen (2005).

UnBuild: Reverse building, i.e., deleting all objects and empty vectors.

CheckBuildStatus: Check whether or not the network has been built.

GetNetworkPath: Get the path of the network, does not include the name of network.

GetNetworkName: Get the name of the network, dos not include the path.

ResetCoordidates: Shift coordinates of all nodes to ensure all are positive, without changing the relative position of nodes. .

ResetLinkLength: Set the length (`len` member) of all links consistent with that defined by the coordinates of its tail and head nodes.

Initialize: In static network this function conducts trivial work.

CheckInitialStatus: Check whether or not the network has been initialized.

CreateNewLink: Create a new link and insert it into the vector, called either in a build function or in GUI.

CreateNewNode:	Create a new node and insert it into the vector, called in in a build function or in GUI.
CreateSOrigin:	Create a new static origin object and insert it into the vector.
ChooseSPAlgorithms:	Choose a shortest path algorithm. The default choice is DEQUE (a stack + a queue) which works well for transportation type of networks. See DATSTRUCT (in file tnm_header.h) for all options.
SPath:	Calculate and return the shortest path between two nodes (either identified by ID or pointer). Users should manage the memory associated with the path pointer.
EPN_KSP:	Calculate and return a queue of K-shortest paths between two nodes, using the Eppstein's algorithm .
BFM_KSP:	Calculate and return a queue of K-shortest paths between two nodes, using the Bellman-Form-Moore algorithm.
DIJK_KSP:	Calculate and return a queue of K-shortest paths between two nodes, using the Dijkstra's algorithm..
PICA_KSP:	Calculate and return a queue of K-shortest paths between two nodes, using the successive average algorithm..
UpdateSP:	Calculate a shortest path tree rooted as an origin node.
UpdateSPR:	Calculate a shortest path tree rooted as a destination node .
RetrieveSPFromTree:	Given a shortest path tree is available (from either UpdateSP or UpdateSPR), calculate and find shortest path. It is faster to use this function combined with UpdateSPR or UpdateSP if multiple O-D pairs are considered with identical origin or destination.
MaxFlow:	Calculate the maximum flow between two nodes.
PathEnum_DFS:	Enumerate and return all paths between two nodes using depth-first-search.
PathEnumeration:	Enumerate paths for all O-D pairs and store the paths in the `pathSet` member of destination.
AllOrNothing:	Calculate shortest paths for all O-D pairs, then compute link flows by assigning all commodities onto the shortest paths.
AllOrNothing(org):	Perform All-Or-Nothing assignment for the given origin only.
AllOrNothingP:	set the flow of the last path stored in `pathSet` of each O-D pair as total demands, set all other path flow as zero, then map path flow to link flow..
AllOrNothingP(org):	Perform AllOrNothingP for just one origin.
ColumnGeneration:	Search for shortest path for each O-D pair, then add them into the corresponding `pathSet`.
ColumnGeneration(org):	Perform ColumnGeneration for just one origin.
MultiColGeneration:	Search for K-shortest path for each O-D pair, then add them into the corresponding `pathSet`. Instead of generating at most one path every time, this function can generate more than one path according to users requirement.

ReposeSP: Recalculate path cost and pose the shortest path of each O-D pair at the end of the `pathSet` vector.

BuildLink2PathPtr: fill the `pathInciPtr` member in each link object, which contains all path pointers using the link.

ClearLik2PathPtr: clean up the `pathInciPtr` member in each link object.

Path2Link: Map path flows to link flows.

UpdateLinkCost: Compute and update costs on each link by calling links' GetCost function. For most static links, `cost` is a function of `volume`.

UpdatePathCost: Update link cost first, then compute and update path costs.

Reset: The following settings are performed: all paths are deleted, `volume` and `markStatus` on links are set to zero, `pathInciPtr` on links is cleaned up. Finally, `cost` on links is updated based on zero `volume`.

MinLinkCover: Find a minimum set of links such that all paths currently used will use at least one link contained in the set.

TotalDemand: Compute and return the total demand of commodities.

TotalPaths: Compute and return the total number of used paths.

ClearPaths: Delete paths which carry flows smaller than a certain percentage of total demands of the associated O-D pair.

CatchODPtr: Get a pointer of a destination from its starting and ending node ID.

CatchNodePtr: Get a pointer of a node from its ID.

CatchLinkPtr: Get a pointer of a link from its ID.

CatchOrgPtr: Get a pointer of an origin from its ID.

DeleteALink: Delete a link, usually used in GUI.

DeleteANode: Delete a node,usually used in GUI.

DeleteAnOrigin: Delete an origin, usually used in GUI.

DeleteDestinations: Delete a dest, usually used in GUI.

UpdateANode: Update the existing node with new properties, usually used in GUI.

AllocateNodeBuffer: Allocate memory for buffer for all nodes according to `nodeBufferSize`.

AllocateLinkBuffer: Allocate memory for buffer for all links according to `linkBufferSize`.

AllocateDestBuffer: Allocate memory for buffer for all destinations according to `destBufferSize`.

AllocatePathBuffer: Allocate memory for buffer for all paths according to `pathBufferSize`.

ReleaseNodeBuffer: Release memory for node buffers.

ReleaseLinkBuffer: Release memory for link buffers.

ReleaseDestBuffer: Release memory for destination buffers.

TNM_DNET
+routingType : RoutingType
+RTUpdateFreq: smallInt
+reactiveAssignType : ReactiveAssignType
+assignHorizon : largeInt
+simulationHorizon : largeInt
+numOfKSP : int
+reactiveRatio : floatType
+m_startTime : time_t
+numControledNode: int
+m_gCostElem : TNM_GCOSTELEM *
+m_inciManager : TNM_IncidentManager
#m_demandLevel : floatType
#m_releaseVehPerLoad : bool
#initAssignmentType : INITASNTYPE
#resFlowInLoading : floatType = 0.0
#balanceTotalResFlow : bool = false
#m_enableGenCost : bool
#unitTime : smallInt = 1
#flowScalar : smallInt = 1
#unitLoadTime : smallInt = 1
#numOfLoading : int
#lastLoadingStatus : short
+m_initDemandFile : bool
+m_initAssignFile : bool
+m_loadRecord : bool
#m_simTick : largeInt
#m_asnTick : largeInt
#numOfReleasedVeh : largeInt
#numOfActiveVeh : largeInt
#numOfVehsMoved : largeInt
#activeVehOnAsnEnd : largeInt
#m_userTerminate : bool
#m_enforceFIFO : bool = true
#m_gtFactor : float = 1.5
#m_gridlockCriterion : int
#m_enableIncident : bool
-asnElemBufferSize : int

Figure 3.20: Data members of `TNM_DNET`

ReleasePathBuffer:	Release memory for path buffers.
EmptyPathSet:	Delete all paths.

`Overridables:`

UnIntialize:	Reverse initialize operation, nothing important incurred in base class.
AllocateNewLink:	Allocate memory for new links, called in `CreateNewLink`. Different links are handled here and recast into the base pointer. Whenever a new link type is developed, this function should be first rewritten to support the new type.
AllocateNewNode:	Allocate memory for new nodes, called in CreateNewNode. Its functionality is similar to AllocateNewLink.
ClearZeroPaths:	Delete paths carrying tiny flows, similar to ClearPaths.

CLASS `TNM_DNET` PARENT CLASS: `TNM_SNET` SEE FIGURE 3.20

`Attributes:`

routingType:	routing type, either *Predictive* (all vehicles have a predetermined path), *Reactive* (no vehicle has a predetermined path), and *Mixed* (Predictive and Reactive vehicles are mixed).

```
+SetRoutingTypeStr(in str : string)
+SetInitDemandPatternStr(in str : string)
+BuildDanet2(in uTime : smallInt) : int
+BuildDanet(in scale : floatType, in uTime : smallInt) : int
+BuildDatabase(in uTime : smallInt) : int
+BuildDynasmartP(in uTime : smallInt, in distScalar : floatType) : int
+UnBuild() : int
+SaveDanet2(in nUnit : int) : int
+SaveDanet2(in nUnit : int, in cName : string) : int
+SaveIncidents() : int
#EstablishConnections() : int
+Initialize(in dmdFromFile : bool, in asnFromFile : bool) : int
+UnInitialize() : int
+CreateDOrigin(in nid : int, in nd : int) : TNM_DORIGIN *
+CreateNewIncident(in link : int, in startt : long, in endt : long, in dist : float, in intens : float, in startL : char, in endL : char) : TNM_INCIDENT *
#AllocateNewLink(inout ltype : const TNM_LINKTYPE) : TNM_SLINK *
#AllocateNewNode(inout ntype : const TNM_NODETYPE) : TNM_SNODE *
#InitAsnPattern(in fromDisk : bool) : int
#InitDmdPattern(in fromFile : bool) : int
#InitMvmtArray()
#AssociateMvmtWthDwnLnks()
#InitializeIncident() : int
#PreMoveVehicles() : int
#WriteSimulationParameters(inout sumFile : ofstream)
+Loading(in recordMode : bool) : int
+LoadLinkTdCum() : int
+LoadLinkTdTime(in saveCum : bool) : int
+ReleaseLinkTdTime()
+WriteLoadingReport(in detail : bool) : int
+GetActiveOnEnd() : largeInt
+UserTerminate()
+EvaluateRandomness() : double
+OpenPsuedoLoading(in link : TNM_DLINK* = 0) : int
+PsuedoLoading(in link : TNM_DLINK*, in startT : int = 0, in endT : int = 0) : int
+ClosePsuedoLoading(in link : TNM_DLINK* = 0) : int
#CreateRoutingTable() : int
#UpdateRoutingTable() : int
#DeleteRoutingTable()
#AllocateLinkTdTime() : int
#ReleaseLinkTdCum()
#ReleaseVehicles(in m_asnTick : largeInt) : largeInt
#InitLoading() : int
#PostLoading() : int
#CheckDestResidual() : int
+ClearZeroPaths()
+CleanPathSet()
+InitVarDTAPar() : int
+InitGenCostPar() : int
+TDColumnGeneration(in variableDemand : bool, in dc : DYNACOSTTYPE = DC_NMCOST) : int
+TDColGenII(in costIsTime : bool, in onePath : bool = false) : int
+ComputeMarginalCost_O() : int
+ComputeMarginalCost_N() : int
+MSASwap(in variable : bool, in stepSize : floatType, in pathSpecific : bool = false) : floatType
+SmartSwap(in variable : bool, in stepSize : floatType) : floatType
+ProjectNegCost(in variable : bool) : int
+AllocateASNElem(in size : int) : int
+AllocateLinkTdCost() : int
+ReleaseLinkTdCost() : int
+ReleaseAsnElem() : int
+GetAsnBuffersize() : int
+WriteFlowInto(in column : int) : int
+ReadFlowFrom(in column : int) : int
+SwapFlows(in column : int) : int
+SwapColumns(in column1 : int, in column2 : int) : int
+WriteCostInto(in column : int) : int
+ReadCostFrom(in column : int) : int
+CombineColsIntoFlow(in column1 : int, in column2 : int, in stepSize1 : floatType) : int
+CombineFlowColsIntoFlow(in column1 : int, in stepSize1 : floatType) : int
+NormFlowCol(in column : int) : floatType
+NormCostFlow_Col(in column : int) : floatType
+NormCostCol_Col(in column1 : int, in column2 : int) : floatType
+NormCol(in column1 : int, in column2 : int) : floatType
+NormCost() : floatType
+NormFlow() : floatType
+NormCol(in column : int) : floatType
+ComputeGap(in variable : bool) : floatType
+UpdateTDPathFlows()
+UpdateTDPathCost(in variable : bool, in costIsTime : bool = true)
+FromFlowToVehicles(in variable : bool) : int
+PostAssignmentCheck(in variable : bool) : int
+AdjustAssignmentTable(in variableDemand : bool, in stepSize : floatType) : floatType
+InitializeControl(in fromFile : bool) : int
#InitializeControlFile() : int
#InitializeControlAuto() : int
#TDSP_DOT(in root : TNM_SNODE*, in costIsTime : bool = true) : int
#TDSP_DOT(in root : TNM_DDEST*, in variable : bool, in costIsTime : bool = true) : int
```

Figure 3.21: Operations of `TNM_DNET`

RTUpdateFreq:	determine how frequent the routing tale needs to be updated for *Reactive* and *Mixed* routing mode.
assignHorizon:	the largest number of assignment intervals for all O-D pairs.
numOfKSP:	number of shortest paths to be used when initializing the assignment table. Set it before Initialize function is called.
reactiveRatio:	The percentage of vehicles to follow reactive routing.
m_startTime:	an absolute time point from which the analysis period starts.
numControledNode:	total number of controlled nodes.
m_gCostElem:	a uniform cost element (define on-time arrival time window, penalty parameters etc.) for all O-D pairs. In many cases, this element is shared by all O-D pairs..
m_inciManager:	an incident manager to manage all incident-related operations.
m_demandLevel:	a scalar used to easily change the overall demand level proportionally (i.e., this scalar will be multiplied with the `assDemand` of each dest to get the total demand). Should be set before Initialize function is called .
m_releaseVPL:	whether or not the vehicle should be released once per loading interval. Releasing per loading interval is less efficient (compared against release per assignment interval) but recommended to accurately follow the designated assisgnment pattern.
intAssignType:	how to distribute the total travel demand over time. Several built-in patterns (e.g., uniform, triangle, trapezoid) can be selected.
resFlowInLoad:	the discrepancy of overall demands and the sum of all released vehicles. This is introduced by rounding errors.
balanceResFlow:	whether or not the overall residual flows should be absorbed by proper balance strategy. This is not recommended unless a significant `resFlowInLoad` is observed..
m_enableGenCost:	whether or not the general cost is considered (usually for DTA with departure time choice). If this is true, then all O-D pairs' `m_genCostElem` will be initialized .
unitTime:	the length of assignment interval in seconds.
flowScalar:	how many vehicles equal to a unit of flow. Large flow scalars imply finer resolution but higher overhead.
unitLoadTime:	the length of loading interval in seconds. It can be a float number.
numOfLoading:	Number of loading conducted since last time the ResetNumOfLoading is called.
lastLoadingStatus:	the status (succeed, gridlock, user terminate or error) of the latest loading.
m_initDemandFile:	whether the time-dependent demand pattern is automatically generated (based on the value of `intAssignType`) or obtained from an .dmd file.
m_initAssignFile:	whether the time-dependent path flow pattern is automatically generated (based on `intAssignType` and `numOfKSP`) or obtained from an .asn file.

m_loadRecord: whether or not the *Record* mode is open in loading. When open, vehicles trace will be recorded and the loading process will be monitored more carefully.

m_simTick: the current simulation time interval, only used during loading (may be accessed from GUI).

m_asnTick: the current assignment time interval, only used during loading (may be accessed from GUI).

numOfReleasedVeh: total number of released vehicles in one loading.

numOfActiveVeh: the number of currently active vehicles (i.e., those on network), only used during loading (may be accessed from GUI).

numOfVehsMoved: the number of vehicles which were moved in the last time interval, only used during loading (may be accessed from GUI).

actVehOnAsnEnd: the number of active vehicles as the assignment horizon is ended. Used in GUI to predict remaining overhead.

m_userTerminate: whether or not the user wants to terminate the loading. Mainly used in GUI to give users the flexibility to terminate loading at any time at their will.

m_enforceFIFO: whether or not FIFO should be enforced. This provides an external control over FIFO behavior. FIFO principle is embedded in some node model (`TNM_DNODE_FWJFI`), yet users can always avoid FIFO by closing this switch.

m_gtFactor: this factor is intended to affect the maximum possible throughput of `TNM_DNODE_GCN` (Generalized control node). It ranges between 1 and 5. Usually a value of 1.5 - 2 is recommended.

m_gridlockCrit: after a certain number (`m_gridlockCrit`) of consecutive time intervals during which no active vehicle can move, the loading program will declare a gridlock and terminate itself.

m_enableIncident: whether or not incidents should be considered during loading.

asnElemBufferSize: the size of buffer in assignment elements.

`Operations:`

SetRoutingTypeStr: Set routing type from a string. Mainly used in GUI.

SetInitDmdPatStr: Set `initAssignType` from a string. Mainly used in GUI.

BuildDanet2: Build a dynamic network defined in a DANET2 format. There are two versions of the function. For the version without input parameter, the `unitLoadTime` is assumed to be equal to `unitTime`. The second version let users specify the `unitLoadTime` upon building. In most cases, the second version is recommended.

BuildDanet: Build a dynamic network defined in a DANET format. DANET is a predecessor of DANET2, now obsolete.

BuildDatabase: Build a dynamic network defined in a MYSQL database. Not used that often.

BuildDynasmartP: Build a dynamic network defined in DYNASMART-P. Remember the network name for DYNASMART-P is a folder name rather than the base file name as usual. This provides a way to convert DYNASMART-P networks into DANET2 format, combined with SaveDanet2 function.

SaveDanet2:	Save current network object in the format of DANET2. Users can specify which files they want to generate and where they want to save the files.
SaveIncidents:	Save the incidents stored in `inciManager` into a .icd file. Usually used in GUI.
InitAsnPattern:	An internal function called in Initialize, to generate time-dependent assignment pattern.
InitDmdPattern:	An internal function called in Initialize, to generate time-dependent demand pattern.
InitializeIncident:	Create all incident objects and add them into `inciManager`, called from Initialize.
Loading:	One of the most important functions. Perform dynamic network loading and save resulting cumulative curves in a binary file (.cum) if succeed. If *Record* mode is open, three additional text files will be written: .sim file (recording simulation process), .lip file (recording overall delay profile), and .veh file (recording detailed vehicle trace). Unless these files are in need, *Record* mode should be keep close for faster loading.
LoadLinkTdCum:	Load link cumulative curves from .cum into memory (`tdCumIn` and `tdCumOut`). This works only if the last loading is successfully concluded.
LoadLinkTdTime:	Compute time-dependent link traversal time (`tdTT`) based on link cumulative curves. Again, this requires that the last loading is successful. It will always call LoadLinkTdCum.
ReleaseLinkTdTime:	Release memory allocated to handle time-dependent link traversal time (`tdTT`).
WriteLoadingReport:	This function summarizes the last loading result (total distance, delay, average speed, etc.) and write them into a file (.sum). It also writes a binary file (.dlk) which contains all loading results on links (primarily cumulative curves). This second file is intended to be used in GUI for fast retrieving.
UserTerminate:	This function allows user to terminate loading at their will.
EvaluateRandomness:	An interval function called in WriteLoadingReport, to evaluate the potential routing errors associated with discretizaton scheme. Users will be notified the evaluation result in .sum file, and if not positive, they should consider a finer discretization scheme to avoid significant rounding errors.
OpenPsLoading:	Prepare for pseudo loading. Note that cumulative curves are assumed to be set before Pseudo Loading is opened..
ClosePsLoading:	Terminate pseudo loading.
PsuedoLoading:	Pseudo Loading is purely designed for animating loading results in GUI. The idea is to make use of established cumulative curves to separate the interaction between links. That is, post animation can be done for any individual link separately. Thus, pseudo loading can be performed for a single link or all links simultaneously, depending on users' need..
CreateRoutingTable:	Internal function, called in Loading, allocate memory for the routing table used in *Reactive* or *Mixed* routing mode.

UpdateRoutingTable:	Interval function, called in Loading, update routing table according to the instantaneous link traversal times.
DelteRoutingTable:	Release memory allocated for routing table.
AllocateLinkTdTime:	Release memory allocated for time-dependent link travel times.
ReleaseLinkTdCum:	Release memory allocated for cumulative curves.
ReleaseVehicles:	Internal function, called in Loading, create new vehicles and put them on network.
CheckDetResidual:	Internal function, called in Loading, check the residual flows (i.e., the difference between the designated demand and released vehicles) to ensure they are not too large.
CleanPathSet:	Delete paths carrying tiny flows.
InitVarDTAPar:	Read the parameters necessary for solving DTA problem with departure time choice from .dta file and save them into the structure `m_genCostElem`.
InitGenCostPar:	An interval function, called in `Intialize`, set all destinations' `m_genCostElem` as the one stored in network object.
TDColGeneration:	Like the ColumnGeneration function in `TNM_SNET`, this function generate new shortest path for each O-D pair for each time-interval, and try to add the new assignment element into the `fixAssignTable`. All travel costs associated with all assignment elements will also be updated.
TDColGenII:	Another version of Column generation, but this one is specifically designed for dynamic O-D estimation.
ComMarginalCost_O:	Compute marginal cost for solving dynamic system optimal problem using the "old" method.
ComMarginalCost_N:	Compute marginal cost for solving dynamic system optimal problem using the "new" method.
MSASwap:	Use the method of successive average to get a new assignment pattern.
SmartSwap:	Use the day-to-day swapping method to get a new assignment pattern.
ProjectNegCost:	Use the projection method to get a new assignment pattern.
AllocateASNElem:	Allocate the memory for buffer in assignment elements.
AllocateLinkTdCost:	Allocate the memory for `tdCost` in links.
ReleaseLinkTdCost:	Release the memory for `tdCost` in links.
ReleaseAsnElem:	Release the memory for buffer in assignment elements.
WriteFlowInto:	Write `flow` in the assignment element into a specified element in the buffer array (e.g., 0 means the first element of buffer array). $b[i] = f$ (b – buffer, f – flow).
ReadFlowFrom:	Read the value of a specified element in the buffer array into `flow`. $f = b[i]$.
SwapFlows:	Swap `flow` in the assignment element with a specified column in the buffer array. $t = b[i], b[i] = f, f = t$.

SwapColumns: Swap the values of two specified columns in the buffer array. $t = b[i], b[i] = b[j], b[j] = t$.

WriteCostInto: Write `cost` in the assignment element into a specified element in the buffer array. $b[i] = c$ (c – cost).

ReadCostFrom: Read the value of a specified element in the buffer array into `cost`. $c = b[i]$.

CombineColsIntoFlow: $f = \lambda b[i] + (1 - \lambda)b[j]$ (where $\lambda \in (0, 1)$ is a step size).

CombineFlwColsIFlw: $f = \lambda f + (1 - \lambda)b[i]$.

NormFlowCol: Calculate the sum of $(f - b[i])^2$ for all assignment elements.

NormCostFlow_Col: Calculate the sum of $(f - b[i])c$ for all assignment elements.

NormCostCol_Col: Calculate the sum of $(b[i] - b[j])c$ for all assignment elements.

NormCol: Calculate the sum of $b[i]^2$ for all assignment elements.

NormFlow: Calculate the sum of f^2 for all assignment elements.

NormCost: Calculate the sum of c^2 for all assignment elements.

ComputeGap: Compute the equilibrium gap for the current assignment pattern.

UpdateTDPathFlows: Update the total flows assigned to a path for each O-D pair in all assignment intervals.

PostAssignCheck: Call the PostAssignCheck function in all destinations.

InitializeControl: Internal function, called in Initialize. Set up controlled intersections.

InitializeControlFile: Read control information from a file.

InitializeControlAuto: Automatically set control information. Usually not used.

TDSP_DOT: Compute time-dependent shortest paths.

`Overridables:`

Initialize: Unlike it static counterpart, this function contains substantial operations, mainly setting up time-dependent assignment pattern, signal timing plans and incidents.

UnInitialize: this will delete all paths, assignment elements, and incidents.

CreateDOrigin: Create dynamic origins. To be overridden in `TNM_MNET`.

CreateNewIncident: Create new incidents. To be overbidden in `TNM_MNET` to handle the microscopic incidents.

PreMoveVehicles: Operations incurred before vehicles are actually moved in loading. In `TNM_MNET` it only contains incidents processing and routing table update. In `TNM_MNET` bidding process is considered as a pre-move operations.

WriteSimParameters: Internal function, called in WriteSimulationReport.

InitLoading: Operations incurred when initializing loading, such as allocate necessary memory.

PostLoading: Operations incurred when terminating loading, such as clean up allocated memory.

CLASS TNM_MNET PARENT CLASS: TNM_SNET SEE FIGURE 3.22

Attributes:

- **m_cellResolution:** This number directly determines the cellular length on each link. See Equation 3.2 for how the cellular length is computed.
- **m_cumFile:** the file pointer for storing the cumulative curves.
- **m_jamTime:** the number of consecutive time intervals during which during which no active vehicle can move.
- **m_installLD:** whether or not loop detectors should operate in simulation.
- **m_numOfLD:** total number of loop detectors.

Operations:

- **Simulation:** This function is intended for GUI purpose. Unlike Loading function, users can start and end the simulation process flexibly, not necessarily a complete network loading.
- **PostSimulation:** Operations incurred upon the termination of simulation, e.g., close .cum file.
- **SaveLDTFile:** Save all loop detector information into a .ldt file.
- **GetLoopDetector:** Retrieve a pointer of a loop detector.
- **InitializeLoopDetector:** Install loop detectors so that they will operate (recording traffic) during the simulation. A static file will be opened (_db.ldt) to write all recorded information (traffic counts, densities) upon the intialization. This will be done when Initialize is called if `m_installLD` is true.

CLASS TNM_PROBNET PARENT CLASS: TNM_DNET SEE FIGURE 3.22

Attributes:

- **m_aotUnit:** the length of unit time interval used when solving arrival on time (AOT) problem.

Operations:

- **InitializeAOT:** Initialize the discrete probability density function for each link. If input parameter is M, then the total number intervals are set to M and the probabilities density is randomly selected; if input parameters are `upperBound` and `stepSize`, then the total number interval is `upperBound/stepSize`, and the probability density is computed using a gamma distribution (parameters in the gamma function are randomly selected). Numerical integration is involved when calculating probabilities.
- **InitializeAOTFile:** Read parameters for the gamma function for each link from a file.
- **AOT_DOT:** Solving the AOT problem using the increasing order of time algorithm.

TNM_MNET
#m_cellResolution : short = 5 #m_cumFile : ofstream #m_jamTime : int #m_installLoopDetectors : bool #m_numOfLoopDetectors : int
+Initialize(in dmdFromFile : bool, in asnFromFile : bool) : int +UnInitialize() : int +CreateDOrigin(in nid : int, in nd : int) : TNM_DORIGIN * +CreateNewIncident(in link : int, in startt : long, in endt : long, in dist : float, in intens : float, in startL : char, in endL : char) : TNM_INCIDENT * #AllocateNewLink(inout ltype : const TNM_LINKTYPE) : TNM_SLINK * #AllocateNewNode(inout ntype : const TNM_NODETYPE) : TNM_SNODE * #PreMoveVehicles() : int #WriteSimulationParameters(inout sumFile : ofstream) #InitLoading() : int #PostLoading() : int +GetCellResolution() : int +SetCellResolution(in r : short) +Simulation(in startT : int, in endT : int) : int +PostSimulation() : int +InstallLoopDetectors(in install : bool = true) +IsLoopDetectorInstalled() : bool +SaveLDTFile() : int +GetLoopDetector(in link : int, in lane : int, in dist : float) : TNM_LoopDetector * #InitializeLoopDetectors() : int

TNM_ProbNet
+m_aotUnit : double
+InitializeAOT(in M : largeInt) : int +InitializeAOT(in upperBound : floatType, in stepSize : floatType) : int +InitializeAOTFile(in upperBound : floatType, in stepSize : floatType) : int +AOT_DOT(in root : TNM_SNODE*, in detail : bool) : int +AOT_PICARD(in root : TNM_SNODE*, in detail : bool) : int +ReleaseAOT() +PrintOptimalAOTPolicy(in availableTime : floatType) +PrintLinkProb() -AllocateAOTProb() : int -ReleaseAOTProb()

Figure 3.22: Data members and Operations of `TNM_MNET` and `TNM_ProbNet`

AOT_PICARD:	Solving the AOT problem using the successive average algorithm.
ReleaseAOT:	Release all temporary memory used in solving AOT problem (both tdTT and tdCost).
PrintOptAOTPolicy:	Print the optimal policy starting from each node for a given time budget.
PrintLinkProb:	Print probability density function on each link.
AllocateAOTProb:	Allocate memory for storing probability density function (tdCost).
ReleaseAOTProb:	Release memory for storing probability density function.

Chapter 4

Examples

This chapter demonstrate the usage of TNM using several typical examples. The first example shows how to use `TNM_SNET` to implement the Frank-Wolfe algorithm for solving static traffic assignment problem. The second example is about macroscopic and microscopic simulations. The last example illustrate the use of `TNM_ProbNet` for solving the stochastic on-time arrival problem.

4.1 Frank-Wolfe algorithm for TAP

Figure 4.1 and 4.2 shows a simple implementation of Frank-Wolfe algorithm based on `TNM_SNET`.

```
double dz = ComputeDzDa(TNM_SNET* network, double step)
{
        TNM_SLINK *link;
        double  tmp;
        for (int i = 0;i<network->numOfLink;i++)
        {
                link    = network->linkVector[i];
                diff    =;
                tmp  = link->volume;
                link->volume = link->buffer[0] +  (link->volume - link->buffer[0])*step;
                dz    +=  (link->volume - link->buffer[0])*link->GetCost();
                link->volume = tmp;
        }
}
double ofv = ComputeOFV(TNM_SNET* network)
{
        double ofv = 0.0;
        for (int i = 0;i<network->numOfLink;i++) ofv += network->linkVector[i]->GetIntCost();
}
```

Figure 4.1: Implementation of Frank-Wolfe algorithm: ComputeDzDa and ComputeOFV

As seen from Figure 4.2, to access TNM, a header file tnm.h should be first included. Users need

FrankWolfe algorithm:

```
Include <tnm.h> //include TNM header file to get access to all TNM objects
int main()
{
      TNM_SNET *network = new TNM_SNET("c:\\test\\mynet"); //create a new network object
      double OFV, step, oldOFV = 0;
      if(network->BuildFort(BPRLK)!=0)  //build network object
      {
            cout<<"\tEncounter problems when building a network object!"<<endl;
            return 1;
      }
      network->AllocateLinkBuffer(1); //allocate an buffer array of size 1
      network->AllOrNothing(); // put the all-or-nothing assignment results in volume
      network->UpdateLinkCost(); //update link cost
      OFV = ComputeOFV(network); //compute objective function value
      while(fabs((OFV-oldOFV)/oldOFV)>0.001)
      {
            for (int i = 0;i<network->numOfLink;i++) //save current volume
                  network->linkVector[i]->buffer[0] = network->linkVector[i]->volume;
            network->AllOrNothing(); //all-or-nothing assignment
            step = BisecSearch(network);   //bisection methof for step size.
            for (i = 0;i<network->numOfLink;i++) //move to new solution
            {
                  link = network->linkVector[i];
                  link->volume  = link->buffer[0] + (link->volume - link->buffer[0]) * step;
            }
            network->UpdateLinkCost(); //update link cost
            oldOFV = OFV; //save old objective function value
            OFV = ComputeOFV(network); //update object function value
      }
      delete network;
}
```

```
double step = BisecSearch(TNM_SNET* network)
{
      double a = 0, dz = 1.0, b = 1.0, step;
      int iter = 0;
      while(dz > 0 || (iter < maxLineSearchIter && (b-a)>=lineSearchAccuracy))
      {
            step= (a + b)/2;
            iter = iter + 1;
            dz = ComputeDzDa(network, step);
            if (dz<0.0)       a = step;
            else              b=step;
      }
      return step;
}
```

Figure 4.2: Implementation of Frank-Wolfe algorithm: main and BisecSearch

make sure proper header files and lib file (dtnm.lib) are properly located. On line 4, a static network object is created which is associated with two files (c:/run/mynet.1 and c:/run/mynet.2) in FORT format (see Zhang et al. (2005) for details of this format). On line 6, BuildFort function is called with an input argument specifying the link type (i.e., the link performance function). On line 11, a buffer of size 1 is allocated on link objects for save the auxiliary solution. On line 12, an all-or-noting assignment is performed to get an initial solution and the resulting flow pattern is saved into `volume` field in links. On line 12, link costs are updated based on current volumes. On line 13, Beckman's objective function is evaluated. How this is done is reported in Figure 4.1. Line 17-18 save the current volume into buffer. Then an auxiliary solution is obtained by performing an all-or-nothing assignment on line 19. On line 20, a bisection search subroutine is called to find out the best step size. Figure 4.2 presents a standard procedure for the bisection method which requires to calculate the derivative information with respect to the step size (ComputeDzDa). Line 21 - 24 compute the new solution based on the search direction and step size. Finally, users should delete the network object to release resource (line 30).
With the help of TNM, implementing the Frank-Wolfe algorithm only takes about 60 lines of codes. Implementing other TAP algorithms or algorithms for solving other network optimization problems can be done in a similar pattern. I developed another class library called MAT (Model and Algorithm Toolkit) which provides a platform to implement algorithms for solving network optimization problems of different types. For example, the Frank-Wolfe algorithm for TAP is encapsulated in a subclass so it can be reused easily. See programming manual for MAT.

4.2 Simulation

Figure 4.3 shows how to perform macroscopic traffic simulation using TNM. First a dynamic network object and a database object are created on line 4 and 5 respectively. Line 6 opens the database. Note that you must have MYSQL installed and run on your local computer to use the `TNM_DBOP` object. On link 7, network is built and the loading interval is set to 5 seconds. Line 13 - 20 set the parameters for initialize and loading. The meanings of most settings are self-evident (they all correspond to some attributes of `TNM_DNET`) except line 19. On line 19, a rounding method is selected through a class `TNM_RoundMethod`. This method controls how the float number is rounded to an integer number in TNM. For example, a float number 3.3 will be always rounded to 3 if rounding is not random. However, if a random rouding is used, then 3.3 has 30% chance to be rounded to 4 and 70% chance to be rounded to 3. The rounding method has a significant impact on the loading result because in each loading time traffic flows need to be transformed to integer numbers to determine how many vehicles can be actually moved. On line 21, network is initialized automatically, meaning that both time-dependent demand and assignment patterns are generated by TNM, not from files. Line 26 calls the loading function to perform traffic simulation, and line 27 check the status of the simulation. After loading is successfully terminated, a loading report (.sum file) is written on line 28, and the vehicle trace information is written into MYSQL database on line 29. Finally, both network and database are deleted.

```
//simulation, macroscopic case
1include <tnm.h>
2int main()
3{
4       TNM_DNET *net = new TNM_DNET("c:\\run\\bin\\mroundabout\\mroundabout");
5       TNM_DBOP *database = new TNM_DBOP(net);
6       database->OpenDatabase();
7       if(net->BuildDanet2(5)!=0)
8       {
9               cout<<"\tFail to build a dynamic network object. \n";
10              return 1;
11      }
12        //setting prameters
13      net->SetFlowScalar(5);
14      net->EnableBalanceRes();
15      net->numOfKSP = 2;
16      net->SetGTFactor(2.0);
17      net->SetInitAsnType(IAT_TRIAGNM);
18      net->EnforceFIFO(0);
19      TNMRoundMethod::EnableRandomRound(false);
20      net->EnableReleaseVehPerLoad(false);
21      if(net->Initialize(false, false)!=0)
22      {
23              cout<<"\tFail to initialize dynamic network object. \n";
24              return 2;
25      }
26      net->Loading(true);
27      if(!net->IsLoadingFinished())
28      {
29              cout<<"\tFail to perform nework loading. Error code = "<<net->GetLastLoadingStatus()<<endl;
30              return 3;
31      }
32      net->WriteLoadingReport(true);
33      database->WriteDynamicsToDatabase();
34      delete database;
35      detele net;
36      return 0;
37}
```

Figure 4.3: Perform Macroscopic Traffic Simulation in TNM

Figure 4.4 demonstrate how to access the microscopic simulation function in TNM, which is slightly different than its macroscopic counterpart. Notice that line 7 and 8 conduct two settings unique for microscopic simulation: install the loop detectors and set the cellular resolution. As

```
//Simulation: microscopic
include <tnm.h>
int main()
{
    TNM_MNET *net = new TNM_MNET("c:\\run\\bin\\mroundabout\\mroundabout");
    TNM_DBOP *database = new TNM_DBOP(net);
    database->OpenDatabase();
    net->InstallLoopDetectors();
    net->SetCellResolution(10);
    if(net->BuildDanet2(-2)!=0)
    {
        cout<<"\tFail to build a microscopic network object. \n";
        return 1;
    }
    net->SetDemandLevel(0.1);
    net->numOfKSP = 2;
    if(net->Initialize(false, false)!=0)
    {
        cout<<"\tFail to initialize a microscopic network object. \n";
        return 2;
    }
    net->Loading(true);
    if(!net->IsLoadingFinished())
    {
        cout<<"\tFail to perform nework loading. Error code = "<<net->GetLastLoadingStatus()<<endl;
        return 3;
    }
    database->WriteLDDataToDatabase();
    for (int i = 1;i<=500;i++)
    {
        cout<<" i = "<<i<<endl;
        net->Simulation(i, i);
    }
    net->Simulation(1, 400);
    getchar();
    net->Simulation(401, 500);
    getchar();
    net->Simulation(1, 50);
    delete database;
    delete net;
}
```

Figure 4.4: Perform Microscopic Traffic Simulation in TNM

shown before, larger m_cellResolution leads to smaller cellular length, and in turn higher simulation accuracy. On line 9, the input argument -2 in BuildDanet2 function actually means a simulation interval $1/2 = 0.5$ seconds. As a genera rule, to define a loading interval $0 < \delta t < 1$ requires to pass an argument $-[1/\delta t]$ [1] in a building function. On line 27, a database function WriteLDDataToDatabase is called to write all loop detector data collected during last simulation into MYSQL database. Line 28-37 show the flexibility of calling Simulation function.

4.3 SOTA problem

Figure 4.5 gives an example how to use TNM to solve the stochastic on time arrival (SOTA) problem. On line 4, a random seed is selected through MyRandom class (MyRandom is developed by others for random number generation and has been integrated into TNM). This seed will affect the random numbers generated for a given distribution. Yet this seed does not affect the results of this illustration. Line 12 - 23 provides a simple interface for users to specify necessary parameters to define the SOTA problem (e.g., the upper bound of the time budget TT and the discrete interval unit). Note that line 21 generates discrete probability density functions and that can be a time-consuming procedure if unit is very small. The loop defined by line 29 - 35 select two different algorithms, successive average (SA) and increasing order of time budget algorithm (IOTA), to solve the SOTA problem. The loop defined by line 37 - 42 print optimal routing policy for all nodes according to the time budget input by users. Finally, on line 43, resources are released and finally network is deleted on line 44.

[1] $[a]$ is the largest integer less than a

```
#include <tnm.h>
int main()
{
	MyRandom::Set(0.66875);
	TNM_ProbNet *net = new TNM_ProbNet("c:\\run\\bin\\test\\test");
	floatType conv;
	if(net->BuildDanet2(12)!=0)
	{
		cout<<"Fail to build a dynamic network object. \n";
		return 1;
	}
	floatType TT = 10.0, unit = 0.01;
	int rootID = 25;
	int aType = 1;
	cout<<"Upper bound of time budget: ";
	cin>>TT;
	cout<<"Step size: ";
	cin>>unit;
	cout<<"Destination: ";
	cin>>rootID;
	net->InitializeAOT(TT, unit);
	cout<<"Algorithm (1 for IOTB, 2 for SA): ";
	cin>>aType;
	TNM_SNODE * root = net->CatchNodePtr(rootID);
	if(root == NULL)
	{
		cout<<rootID<<" is not a valid node"<<endl;
		return 1;  }
	while(aType!=0)
	{
		if(aType == 2) net->AOT_PICARD(root, false);
		else       net->AOT_DOT(root, false);
		cout<<"Algorithm (1 for IOTB, 2 for SA, 0 to exit): ";
		cin>>aType;
	}
	floatType at = 1.0;
	while (at != 0)
	{
		cout<<"\n\tTime budget (0, "<<(net->simulationHorizon - 1) * net->aotUnit<<"], input 0 to exit: ";
		cin>>at;
		net->PrintOptimalAOTPolicy(at);
	}
	net->ReleaseAOT();
	delete net;
	return 0;
}
```

Figure 4.5: Solve SOTA problem in TNM

Bibliography

Bell, M. G. H. & Iida, Y. (1997), *Transportation Network Analysis*, John Wiley and Sons Inc., New York.

Zhang, H., Nie, Y. & Shen, W. (2005), Development of a path flow estimator for inferring steady-state and time-dependent origin-destination trip matrices, Path technical report, University of California, Davis.