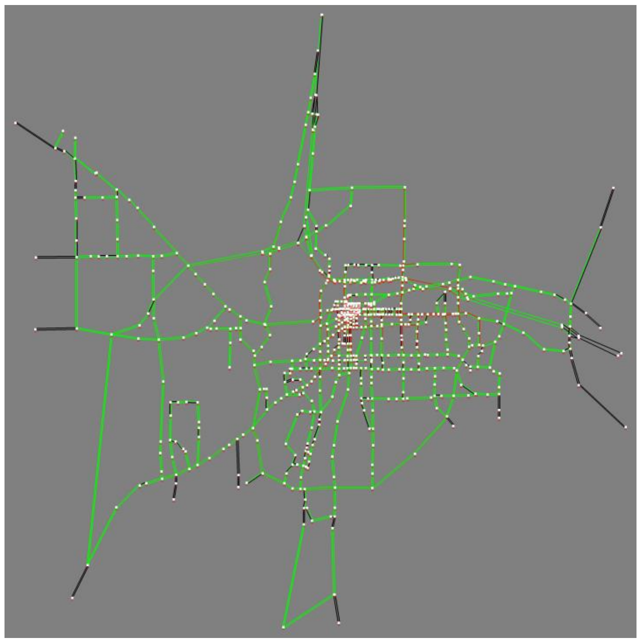
FINAL REPORT

FHWA-WY-13/05F

State of Wyoming
Department of Transportation

U.S. Department of Transportation
Federal Highway Administration



STATEWIDE MESOSCOPIC SIMULATION FOR WYOMING

By:
Stephen D. Boyles, Ph.D.

Department of Civil, Architectural & Environmental Engineering
The University of Texas at Austin
301 E. Dean Keeton St. Stop C1761
Austin, TX 78712

Department of Civil & Architectural Engineering
University of Wyoming
1000 E. University Ave Dept. 3295
Laramie, WY 82071

October 2013

Technical Report Documentation Page

1. Report No. FHWA-WY-13/05F	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Statewide Mesosopic Simulation for Wyoming		5. Report Date October 2013 6. Performing Organization Code
7. Author(s) Stephen D. Boyles		8. Performing Organization Report No.
9. Performing Organization Name and Address Center for Transportation Research The University of Texas at Austin 1616 Guadalupe Street, Suite 4.202 Austin, TX 78701		10. Work Unit No. (TRAIS) 11. Contract or Grant No.
12. Sponsoring Agency Name and Address Wyoming Department of Transportation 5300 Bishop Blvd. Cheyenne, WY 82009-3340 WYDOT Research Center (307) 777-4182		13. Type of Report and Period Covered Final Report for January 2013–July 2013 14. Sponsoring Agency Code

15. Supplementary Notes
WYDOT Technical Contacts: Lee Roadifer, Sherm Wiseman

16. Abstract
This study developed a mesoscopic simulator which is capable of representing both city-level and statewide roadway networks. The key feature of such models are the integration of (i) a traffic flow model which is efficient enough to scale to large regions, while realistic enough to represent traffic dynamics, including queue growth and dissipation and intersection control; and (ii) a user behavior model in which drivers choose routes based on minimizing travel times. Integrating these models is nontrivial, because route choices depend on route travel times, but route travel times are determined from route choices through the traffic flow model. An iterative approach is used to seek a consistent solution to this problem, using the cell transmission model as the traffic flow model.

These features have been implemented in a software program, for which source code and tutorials have been provided as appendices to this report. Additional modules are provided for generating graphical views of networks, performing warrant analysis based on MUTCD procedures (either to assist with network creation, or as a post-processing step), and a spreadsheet interface to the program itself. Ready-to-use networks have been provided representing the city of Casper and the state of Wyoming.

17. Key Words Mesoscopic simulation; dynamic traffic assignment	18. Distribution Statement Unlimited		
19. Security Classif. (of report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 259	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

SI* (Modern Metric) Conversion Factors

Approximate Conversions from SI Units

Symbol	When You Know	Multiply By	To Find	Symbol
Length				
mm	millimeters	0.039	inches	in
m	meters	3.28	feet	ft
m	meters	1.09	yards	yd
km	kilometers	0.621	miles	mi
Area				
mm^2	square millimeters	0.0016	square inches	in^2
m^2	square meters	10.764	square feet	ft^2
m^2	square meters	1.195	square yards	yd^2
ha	hectares	2.47	acres	ac
km^2	square kilometers	0.386	square miles	mi^2
Volume				
ml	milliliters	0.034	fluid ounces	fl oz
l	liters	0.264	gallons	gal
m^3	cubic meters	35.71	cubic feet	ft^3
m^3	cubic meters	1.307	cubic yards	yd^3
Mass				
g	grams	0.035	ounces	oz
kg	kilograms	2.202	pounds	lb
Mg	megagrams	1.103	short tons (2000 lbs)	T
Temperature (exact)				
°C	Centigrade temperature	1.8 C + 32	Fahrenheit temperature	°F
Illumination				
lx	lux	0.0929	foot-candles	fc
cd/m^2	$candela/m^2$	0.2919	foot-Lamberts	fl
Force and Pressure or Stress				
N	newtons	0.225	poundforce	lbf
kPa	kilopascals	0.145	pound-force per square inch	psi

Approximate Conversions to SI Units

Symbol	When You Know	Multiply By	To Find	Symbol
Length				
in	inches	25.4	millimeters	Mm
ft	feet	0.305	meters	M
yd	yards	0.914	meters	M
mi	miles	1.61	kilometers	Km
Area				
in^2	square inches	645.2	square millimeters	mm^2
ft^2	square feet	0.093	square meters	m^2
yd^2	square yards	0.836	square meters	m^2
ac	acres	0.405	hectares	Ha
mi^2	square miles	2.59	square kilometers	km^2
Volume				
fl oz	fluid ounces	29.57	milliliters	Ml
gal	gallons	3.785	liters	L
ft^3	cubic feet	0.028	cubic meters	m^3
yd^3	cubic yards	0.765	cubic meters	m^3
Mass				
oz	ounces	28.35	grams	G
lb	pounds	0.454	kilograms	Kg
T	short tons (2000 lbs)	0.907	megagrams	Mg
Temperature (exact)				
°F	Fahrenheit temperature	5(F-32)/9 or (F-32)/1.8	Celsius temperature	°C
Illumination				
fc	foot-candles	10.76	lux	Lx
fl	foot-Lamberts	3.426	$candela/m^2$	cd/m^2
Force and Pressure or Stress				
lbf	pound-force	4.45	newtons	N
psi	pound-force per square inch	6.89	kilopascals	kPa

Executive Summary

The purpose of this project was to develop a mesoscopic simulation model which can be applied to Wyoming either at the statewide level or the level of individual cities. The main concept behind mesoscopic simulation is to bridge the gap between microscopic simulation, which models small geographic areas (such as a single corridor) with high detail, and macroscopic modeling, which models large geographic areas (such as a state) with relatively little detail. This is particularly important in rural states such as Wyoming, in which a larger share of trips are long-distance, and in which a significant share of freeway volume is out-of-state traffic such as heavy vehicles.

The key to mesoscopic simulation is combining (1) a reasonably simple traffic flow model which can still capture fundamental traffic behavior such as queue growth and dissipation, and signal control at intersections, and (2) a route choice model which reflects drivers' desire to choose the fastest routes to their destinations. These two models interact heavily: for instance, if a major roadway is temporarily closed for a construction project, the route choice model is needed to predict how traffic patterns will shift, but doing so requires knowing how travel times throughout the network will change as well. Briefly, the traffic flow model requires route choices as input and produces travel times, while the route choice model requires travel times as input and produces route choices.

The ultimate goal is to achieve a mutually consistent solution between the traffic flow and route choice modules. In this project, such a system was developed and implemented in the C programming language, with a Microsoft Excel frontend to allow editing and program operation in a more familiar environment. This system is based on iteration between the cell transmission model (the traffic flow model deemed most appropriate after a literature review), a time-dependent version of the A* algorithm (which finds the least travel time route between two points in a network), and the method of successive averages (which adjusts vehicles' route choice based on the updated travel times). Additional modules produce graphical versions of the output data, and perform a warrant analysis to either recommend updated intersection control and signal timing, or generate an initial set of intersection controls and timings when creating a network from scratch.

Three case studies are provided: a small "toy" network to demonstrate the model capabilities; a network representing the city of Casper, under a hypothetical road closure for construction; and a network representing the state of Wyoming, under a hypothetical toll on Interstate 80. For the latter two case studies, "before-and-after" analyses are shown to demonstrate potential applications of the software.

Appendices include more extensive tutorials, a programmer's guide to the simulator, and the C code used to implement the model. In this way, users may continue to modify the software as needed for specific applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this project was to develop a mesoscopic simulation model which can be applied to Wyoming either at the statewide level or the level of individual cities. This report documents the research work performed, and includes additional deliverables as appendices. This chapter reviews the motivation behind the project, the primary research accomplishments, and describing how this report is organized.

1.1 Motivation

Traditionally, planning and operations models used by transportation professionals use markedly different assumptions and spatiotemporal scope. By necessity, planning models consider a large geographic area and long-term time horizon, while operational models focus on a smaller region (such as a single corridor and a few alternate routes). This allows the latter greater detail and fidelity at the expense of scope. However, this difference can lead to inconsistency between these models.

For instance, planning models generally do not account for queuing behavior from signals or temporary congestion, and thus may produce link values which are physically implausible and unsuitable for use in microsimulation programs without significant “tweaking” based on engineering judgment. At the same time, microsimulation models cannot give a proper treatment of diversion and rerouting effects, since by their nature they only focus on a portion of most trips, and diversion depends crucially on the origin and destination, not just the portion of the trip included in the microsimulation. This is a particularly significant issue with modeling diversion due to network interruptions (such as work zones), and can help identify necessary changes to traffic control and other operational strategies to mitigate the effects of these disruptions.

Mesoscopic simulation is a promising technique for bridging this gap, using a level of detail intermediate between planning models and microsimulation. Several methods are available to represent traffic flow, and a major part of the project involved selecting the most appropriate

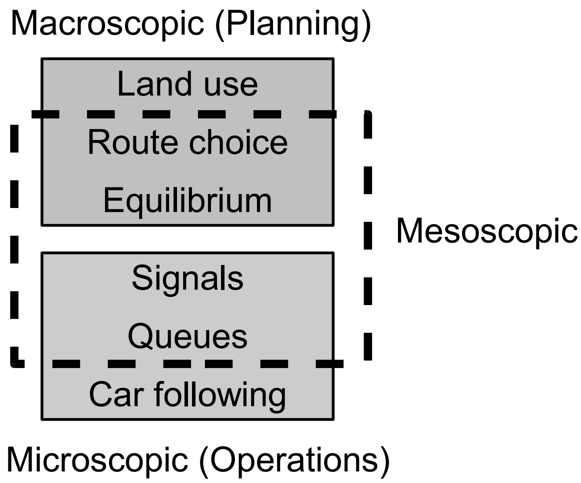


Figure 1.1: Microscopic, mesoscopic, and macroscopic modeling.

model. Particularly, little attention has been paid to the best traffic representation for rural areas. Furthermore, recent advances in computing technology now allows a very large area to be represented in a mesoscopic simulator, even at the statewide level in states with low population density, such as Wyoming.

Mesoscopic simulation adopts a middle ground between the extremes of macroscopic simulation (large area, low detail) and microscopic simulation (small area, high detail), by choosing to represent only the most significant aspects of traffic flow in a computationally efficient manner. The resulting model combines aspects of macroscopic and microscopic modeling, using a consistent set of modeling assumptions (Figure 1.1). Mesoscopic simulation has recently been applied to model traffic flow in major metropolitan areas, including Atlanta, Georgia; Austin, Texas; Dallas-Ft. Worth, Texas; and Chicago, Illinois; but application to rural areas has been lacking even though the transformative potential is just as great, if not more so.

Successfully developing a statewide mesoscopic simulator would benefit WYDOT's current modeling process in several ways:

- Allowing traffic studies to include elements of traditional planning models (such as trip generation, land use, and route choice), leading to more accurate predictions and greater consistency.
- Allowing planning models to benefit from traffic data by serving as a calibration and validation tool, and to generate predicted traffic counts directly from land use models, applied consistently throughout the State.
- Providing WYDOT with a tool which can be used for analysis of policies with statewide implications on traffic flow, such as tolling and its associated diversion effect.

- Creating opportunities for collaboration and data sharing between operations and planning personnel, reducing data collection costs and duplication of effort.

1.2 Primary Accomplishments

In accordance with the proposal, the following primary tasks have been accomplished during the course of this research project:

- A thorough review of the traffic flow theory literature identified the cell transmission model as suitable for statewide, mesoscopic implementation.
- A mesoscopic simulation program has been developed using the C programming language.
- A graphical user interface for the simulator has been developed using Microsoft Excel VBA.
- The simulation program was extended to include basic warrant analysis and signal timing.
- Two Wyoming-specific case studies have been prepared, one for the city of Casper and the other for the entire State.
- Network files for the case studies have been created based on traffic count data and TransCAD files.
- Training tutorials have been developed based on these case studies.
- The C and VBA source code has been documented and provided to WYDOT.

1.3 Outline

The remainder of this report is organized as follows. Chapter 2 presents the engineering concepts underlying the mesoscopic simulator. Chapter 3 explains how these concepts were implemented, using the C language for the simulation code and Microsoft Excel VBA for the interface. Chapter 4 demonstrates this simulation framework, using three examples: a small "artificial" network where all changes can be easily seen; a network representing the city of Casper, before and after a work zone closes a major arterial; and a network representing the state of Wyoming, before and after a toll on I-80. Finally, Chapter 5 concludes the report and identifies an implementation strategy.

Four appendices provide additional deliverables. Appendix A packages the case studies from Chapter 4 in a stand-alone format suitable for training materials. Appendix B provides an overview of the source code organization and data structures, which will be useful for programmers seeking to modify or extend the code. Finally, Appendix D contains the C the simulator and other modules.

Chapter 2

Methodology

2.1 Overview

A mesoscopic simulator contains two equally-important components: a *traffic flow model* which describes the operational nature of traffic flow and delay, and a *user behavior model* which describes how travelers choose their routes. Macroscopic or microscopic simulation tends to focus on one or the other exclusively. Microsimulation software, such as CORSIM or VISSIM, employs a highly realistic traffic flow model, but has a very limited user behavior model, assuming that route flows and turning proportions can be specified exogenously and do not respond to the simulated traffic flow. Macroscopic software, such as TransCAD or VISUM, has a sophisticated user behavioral model which can account for route choice and diversion all throughout a network, but with a much simpler traffic flow model based on unrealistic impedance functions. In a mesoscopic simulator, however, both components are equally important.

This raises the core challenge of mesoscopic modeling: traffic flow and user behavior are closely connected. On the one hand, traffic conditions are determined by the routes that drivers choose to take when they travel. On the other hand, the routes that drivers choose to take are determined by traffic conditions. This circular dependency is indicated schematically in Figure 2.1.

As an example, consider a major maintenance project which will close one or more lanes on a major arterial for several weeks. If no drivers were to change their routes, congestion would arise during the peak period due to the loss of capacity. However, some drivers will divert onto parallel routes (or take entirely different routes, based on their origin and destination). As this happens, congestion will ease somewhat at the work zone location, and perhaps increase slightly along these parallel routes. This in turn will ease the pressure for additional drivers to divert.

Resolving this mutual dependency is the primary methodological challenge in a mesoscopic model. This is accomplished by seeking an *equilibrium* solution, that is, a mutually consistent set of route travel times and route choices. The phrase “mutually consistent” means that the route travel times are exactly those which would occur based on the route choices made, and the route choices are exactly those which would occur based on the route travel times. Under the most

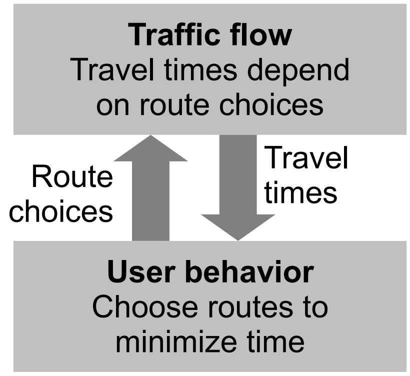


Figure 2.1: Mutual dependency between traffic flow and user behavior.

common assumption — that drivers want to choose the route which minimizes their travel time — the only possible equilibrium solution is one in which all used routes have identical travel time. If any pair of used routes had unequal travel time, drivers would divert from the slower route to the faster one.

This equilibrium principle was first introduced to traffic engineering by Wardrop (1952) and Beckmann Beckmann et al. (1956)[1], which eventually led to the development of macroscopic models such as those in TransCAD or VISUM. However, the traffic flow models used here are exceptionally simplistic, using delay functions which do not reflect the dynamic nature of traffic flow — queues grow and shrink over time (especially at signals), and travel demand varies over time. The main advantage of mesoscopic modeling over this early work is the use of innovative traffic flow models which can account for these dynamics while still being simple enough to apply on large, even statewide, scales.

Finding such an equilibrium involves iteration between the traffic flow and user behavior models. For a given set of route choices, the traffic flow model gives updated route travel times (including any congestion which may arise). Then, the user behavior model updates the route choices as people seek to avoid congestion and divert onto faster routes. Control then returns to the traffic flow model, which updates travel times yet again, and so on until an equilibrium (or near-equilibrium) is found. Since each model must be run multiple times, efficient models are of the utmost importance.

The remainder of this chapter explains the traffic flow model and user behavioral model in greater detail, in Sections 2.2 and 2.3, respectively.

[1]Although this work was prefigured by the economist Pigou (1920)

2.2 Traffic flow model

To be suitable for the project purposes, the chosen traffic flow model must satisfy all of the following conditions. It must be:

1. Scalable to very large regions, including metropolitan areas and statewide networks.
2. Efficient and tractable, not requiring more computational resources than is available on desktop machines.
3. Capable of representing traffic dynamics, in particular how queues grow and dissipate.
4. Capable of representing changes in tripmaking rates over the course of the simulation period.
5. Capable of representing diverse intersection controls (such as signals, stop signs, or grade-separated interchanges).
6. Relatively simple in terms of data input requirements.
7. Transparent in terms of understanding exactly how traffic flow is being propagated, and simple enough to convey to decision-makers and the public.

After a thorough review of the traffic flow literature, the cell transmission model was selected as a model which satisfies these desiderata. The remainder of this section explains the cell transmission model in the context of these criteria, beginning by contrasting it with other well-known traffic flow models, and then deriving the model from the hydrodynamic ("shockwave") theory of traffic flow. These criteria will be repeatedly referred to throughout this section.

2.2.1 Overview of contrasting models

This subsection provides a brief overview of macroscopic and microscopic traffic flow models, to provide context for the description of the cell transmission model which follows.

Macroscopic equilibrium models, such as those used in TransCAD and VISUM, are based on *link performance functions* which report the travel time on a roadway link as a function of the number of vehicles on this link. One standard function was developed by the Bureau of Public Roads (BPR), which has the following form:

$$t = t_0 \left(1 + \alpha \left(\frac{x}{c} \right)^{\beta} \right) \tag{2.1}$$

where t is the travel time on a roadway link, t_0 is the free-flow travel time, x and c are the roadway volume and capacity, and α and β are parameters which can be calibrated to date.

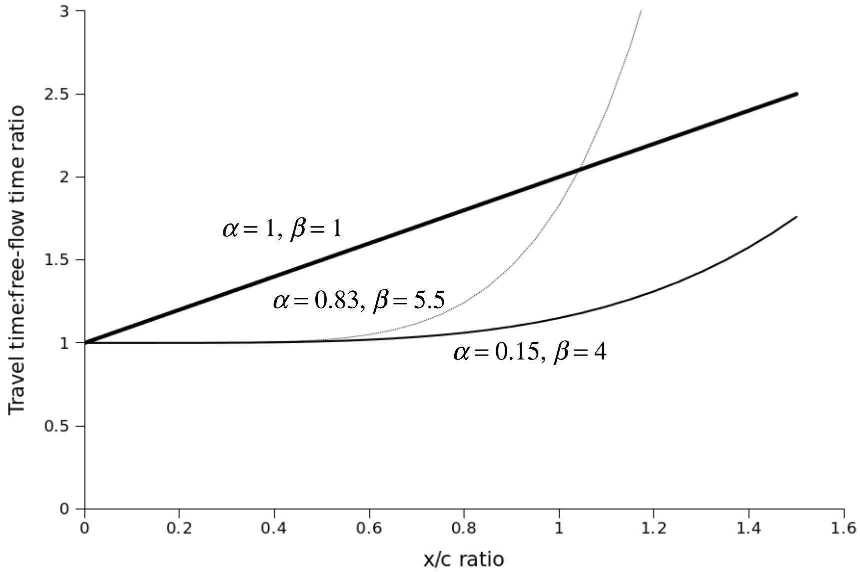


Figure 2.2: BPR functions with different α and β values.

Figure 2.2 shows different BPR functions based on the values of α and β. The most typical choices in practice are $\alpha = 0.15$ and $\beta = 4$.

The advantage of this formulation is that it lends itself to easy solution even on very large networks. Consider a large network with a set of roadway links A, a set of intersection nodes N, and a set of centroid nodes $Z \subseteq N$. Let t_{ij}, x_{ij}, and c_{ij} be the travel time, volume, and capacity on each link, and d^{rs} is the travel demand from origin $r \in Z$ to destination $s \in Z$. Let Π^{rs} be the set of routes between centroids r and s, and h^π the number of drivers choosing route π. Then the equilibrium solution solves the following mathematical optimization problem Beckmann et al. (1956):

$$\min_{\mathbf{x},\mathbf{h}} \quad \sum_{(i,j)\in A} \int_0^{x_{ij}} t_{ij}(x)dx \tag{2.2}$$

$$\text{s.t.} \quad x_{ij} = \sum_{(r,s)\in D} \sum_{\pi\in\Pi^{rs}:(i,j)\in\pi} h^\pi \qquad \forall (i,j) \in A \tag{2.3}$$

$$d^{rs} = \sum_{\pi\in\Pi^{rs}} h^\pi \qquad \forall (r,s) \in Z^2 \tag{2.4}$$

$$h^\pi \geq 0 \qquad \forall \pi \in \bigcup_{(r,s)\in D} \Pi^{rs} \tag{2.5}$$

There are many algorithms which can solve this optimization program very efficiently, even for networks with tens of thousands of links. For an overview of such algorithms, see Patriksson (1994) or Boyles and Waller (2010).

However, the use of link performance functions as the traffic flow model has several significant shortcomings. Referring to the list of criteria in the previous section, it fails to satisfy Criteria 3,

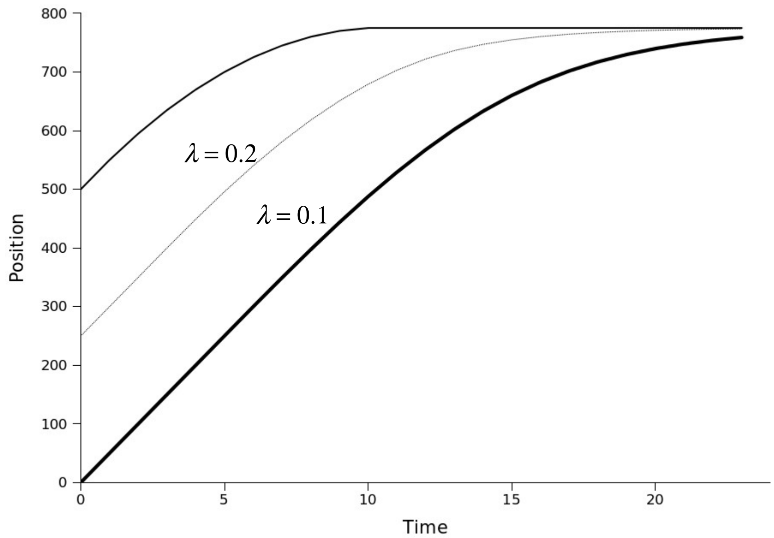


Figure 2.3: Car following with different λ values, $T = 1$.

4, and 5: there is no concept of time dynamics in this model, either regarding the evolution of network conditions or of travel demand. Furthermore, this model is extremely limited in how it can reflect intersection dynamics, because the travel time functions t_{ij} only depend on the flow on that specific link. By contrast, intersection delays typically depend on the flow from conflicting movements, depending on the control type. Perhaps more seriously, there is nothing in the model which enforces $x_{ij} \leq c_{ij}$, that is, that volumes be less than capacity! It should be clear that such functions cannot represent the realities of traffic flow at anything more than a very approximate level.

Microscopic simulation, on the other hand, is usually based on the concepts of car following and gap acceptance. Car following models describe how vehicles move in a traffic stream. Let $z(t)$ describe the position of a vehicle at time t, and $z'(t)$ describe the position of the vehicle immediately in front of it. A basic car following model is

$$\frac{\partial^2 z}{\partial t^2}(t+T) = \lambda \left(\frac{\partial z'}{\partial t} - \frac{\partial z}{\partial t} \right) \tag{2.6}$$

where T is the reaction time of a driver. In other words, the acceleration of a vehicle is proportional to the relative velocity between that vehicle and the vehicle in front of it. The parameter λ can be calibrated to data to reflect the strength of this response; see Figure 2.3 Experiments place the value of λ in practice at approximately 1/3; if $T\lambda < 1/2$, the overall stream of traffic is stable with respect to disturbances.

More sophisticated car-following models have been proposed, taking into account potential coupling between vehicles further ahead, the spacing, and other factors. The primary advantage of these models is their ability to easily represent different vehicle types and driver-specific behavior. A disadvantage is that solving these models requires either the solution of complex partial differential equations, or numerical simulation which is difficult to analyze.

Models for yielding behavior at intersections or merges is generally based on the theories of gap acceptance and queueing theory. When traffic streams intersect, as at a merge or two-way stop, vehicles in one stream must yield the right of way to another. In such cases, vehicles in the minor stream must wait for gaps of sufficient size to appear in the major stream. Let t_c and t_f denote the *critical gap* and *follow-up gap*, respectively, both measured in seconds. The critical gap is the smallest gap in the major stream that a vehicle is willing to accept; the follow-up gap is the additional time required for subsequent vehicles to follow the first vehicle to move during a gap. We generally observe $t_f \leq t_c$. Assuming that headways in the primary stream are exponentially-distributed, one can show that the capacity on the minor approach is given by

$$c = \frac{x_M \exp(-x_M t_c)}{1 - \exp(-x_M t_f)} \tag{2.7}$$

In simulation, individual gaps will be tracked, and the appropriate number of vehicles moved.

While these car-following and gap-acceptance models can be used to model traffic very realistically, like the macroscopic models they do not satisfy all of the criteria listed above. While they can model traffic dynamics, they do not satisfy criteria 1 and 2 because they do not scale well to larger regions. Accurate simulation using these models requires a very fine time step (typically on the order of a tenth of a second) and knowing the exact location of every vehicle on the network during each step.

What is needed is a model which can account for traffic dynamics, while requiring less spatiotemporal precision than the microscopic models. The cell transmission model satisfies all necessary criteria. Before presenting this model, the hydrodynamic theory of traffic flow is reviewed, since this forms the theoretical basis for the cell transmission model.

2.2.2 The hydrodynamic theory

The hydrodynamic theory of traffic flow was independently derived by Lighthill and Whitham (1955) and Richards (1956); it is often referred to as the LWR model. This theory is a *continuum model* or *fluid model* — rather than modeling vehicles as separate, discrete entities, the LWR model approximates traffic flow as a continuous fluid, and applies results similar to those in fluid dynamics. There are three key variables which describe the traffic stream at any location x and time t: the *density* k, the *flow* q (also known as the *volume*), and the *speed* u. Density is measured in vehicles per unit length (often vehicles per mile), and flow is measured in vehicles per unit time (often vehicles per hour). These three variables are related by the *fundamental equation* $q = uk$, which must hold at any point and time.

The LWR theory makes two additional postulates. The first is that the flow at a point depends only on the density at that point, that is, that

$$q(x,t) = Q(k(x,t)) \tag{2.8}$$

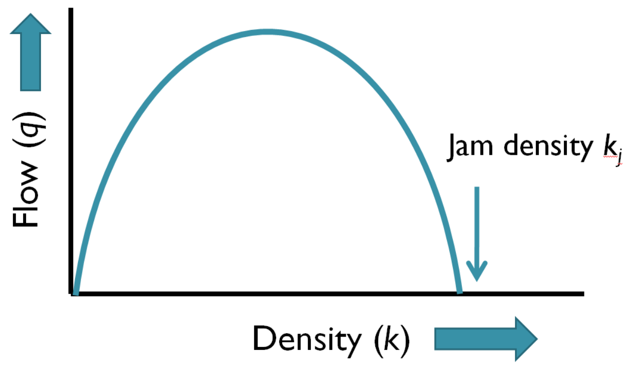


Figure 2.4: A typical fundamental diagram.

for some function Q. This function is known as the *fundamental diagram*. It is typically continuous, concave, and has two zeros: one at $k = 0$, and the other at the *jam density* $k = k_j$. In other words, there are two possible reasons for zero flow: either there are no vehicles and $k = 0$, or traffic is completely jammed and no vehicles are moving, and therefore $k = k_j$. A typical fundamental diagram is shown in Figure 2.4.

The second major postulate of the LWR theory is a conservation law, which states that vehicles cannot "appear" or "disappear" in the network (except for when departing from an origin or arriving at a destination). To enforce this, let $N(x, t)$ represent the *cumulative count* at location x and time t, that is, the total number of vehicles which have passed point x from the start of the modeling period until time t. Under the fluid assumption, $N(x, t)$ is a continuous function. Note that the cumulative count is related to the flow and density. In particular

$$q(x, t) = \frac{\partial N}{\partial t} \tag{2.9}$$

and

$$k(x, t) = -\frac{\partial N}{\partial x} \tag{2.10}$$

where the negative sign in the latter equation reflects the sign convention used: x increases in the direction of flow. Therefore, at any point x, the cumulative count increases as t increases; but at any time t, the cumulative count *decreases* as we move in the direction of increasing x. If N is twice continuously differentiable, then

$$\frac{\partial^2 N}{\partial x \partial t} = \frac{\partial^2 N}{\partial t \partial x} \tag{2.11}$$

or, substituting (2.10) and (2.9) and rearranging, we have

$$\frac{\partial q}{\partial x} + \frac{\partial k}{\partial t} = 0 \tag{2.12}$$

This is the conservation law in the LWR model.

It may not be clear why this derivation expresses the conservation of vehicles, so a geometric argument is presented to help clarify this interpretation. Consider the region of (x, t) space in

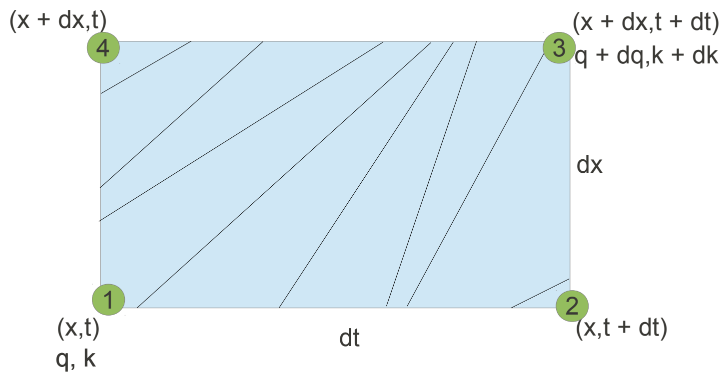


Figure 2.5: Deriving the conservation equation.

Figure 2.5, which has duration dt and spatial extent dx. Let $N(x,t) = N_0$, and assume that the values of $q \equiv q(x,t)$ and $k \equiv k(x,t)$ are known. Similarly, let $q(x+dx, t+dt) = q + dq$ and $k(x+dx, t+dt) = k + dk$. Then, moving to point 2, $N(x, t+dt) = N_0 + q\,dt$ because of (2.9). Moving from point 2 to point 3 and applying (2.10), we have

$$N(x+dx, t+dt) = N(x, t+dt) - (k+dk)dx = N_0 + q\,dt - (k+dk)dx \tag{2.13}$$

Moving from point 3 to point 4, we have

$$N(x+dx, t) = N(x+dx, t+dt) - (q+dq)dt = N_0 + q\,dt - (k+dk)dx - (q+dq)dt \tag{2.14}$$

And finally, moving from point 4 to point 1, we have

$$N(x,t) = N(x+dx, t) + k\,dx = N_0 + q\,dt - (k+dk)dx - (q+dq)dt + k\,dx \tag{2.15}$$

But we already know $N(x,t) = N_0$. Therefore $q\,dt - (k+dk)dx - (q+dq)dt + k\,dx = 0$, or equivalently,

$$\frac{\partial q}{\partial x} + \frac{\partial k}{\partial t} = 0 \tag{2.16}$$

This equation was derived using a conservation principle: that if we move around a closed curve, the value of N should not change when we return, because all vehicles are accounted for.

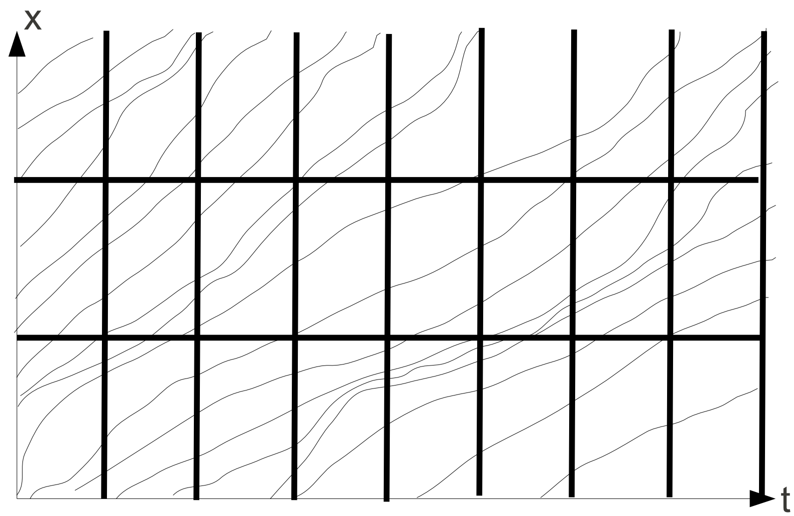
Therefore, the LWR model can be formulated as the solution to a system of partial differential equations: in particular, finding functions $N(x,t)$, $q(x,t)$, and $k(x,t)$ such that

$$q(x,t) = \frac{\partial N(x,t)}{\partial t} \tag{2.17}$$

$$k(x,t) = -\frac{\partial N(x,t)}{\partial x} \tag{2.18}$$

$$q(x,t) = Q(k(x,t)) \tag{2.19}$$

given a set of boundary conditions (for instance, link inflow rates, or constraints on outflows due to a signal).



3	3	3	2	2	1	2	3
3	4	4	4	2	2	4	5
4	4	4	4	5	4	3	1

Figure 2.6: Cell discretization of a trajectory diagram.

2.2.3 Cell transmission model

In general, it is difficult to solve the system of partial differential equations (2.17)–(2.19). However, the cell transmission model (Daganzo, 1994) provides an extremely simple way to solve this system for a particular fundamental diagram Q. Furthermore, the final formulas are in fact quite simple (keeping in mind Criterion 7) and easy to interpret in terms of physical traffic flow. The cell transmission model involves two major steps. First, space and time are discretized into "cells", and the LWR model is reformulated in terms of the number of vehicles moving from one cell to the next during a time step. Second, the fundamental diagram Q is approximated as piecewise-linear (trapezoidal).

Figure 2.6 shows how the discretization functions. The top panel of this figure shows a trajectory diagram, in which each thin line represents the path a vehicle takes (its x coordinate at each time t). The thick horizontal and vertical lines are drawn at regular spacings Δt and Δx, respectively. The bottom panel shows how the cell transmission model would represent this traffic flow: rather than tracking the individual locations of all vehicles at all points in time, it suffices to track the number of vehicles in each cell during each time interval; denote this by $n(x, t)$, and let $y(x, t)$ be the number of vehicles flowing through cell x at time t.

Calculations are greatly simplified by assuming a trapezoidal fundamental diagram, as shown in Figure 2.7. This is in keeping with Criterion 6, regarding data input requirements — a trapezoidal

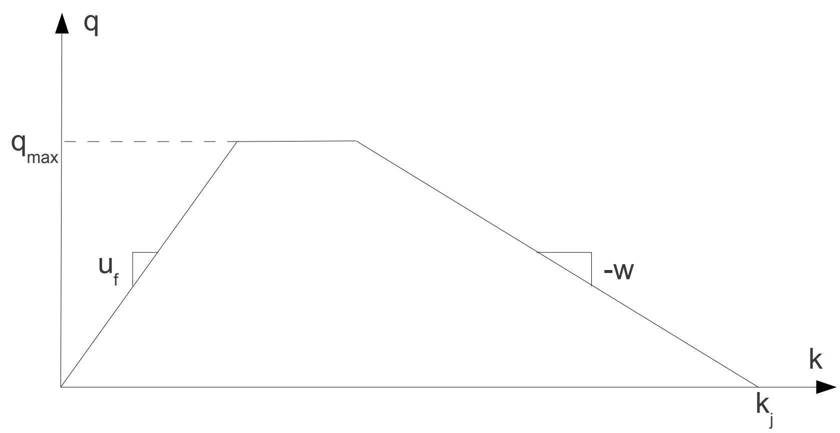


Figure 2.7: The trapezoidal fundamental diagram.

fundamental diagram is completely specified by four parameters, three of which are calculated routinely by traffic engineers. These are u_f, the free-flow speed on a roadway link; Q_{max}, the roadway capacity; k_j, the roadway jam density; and w, the speed at which backward-moving shockwaves travel[2]. u_f and Q_{max} are routinely calculated using procedures such as those in the Highway Capacity Manual (Transportation Research Board, 2010). k_j is typically estimated based a typical front bumper-to-front bumper spacing of vehicles when stopped (say, 20 ft) and the number of lanes. Only w is difficult to calibrate, but experience shows that w is roughly 1/3 to 1/2 of u_f. A key feature of this diagram is that travel speeds do not drop below the free-flow speed until the roadway capacity is reached. This trapezoidal diagram can be mathematically represented by the equation:

$$Q(k) = \min \{u_f k, Q_{max}, w(k_j - k)\} \tag{2.20}$$

The cell transmission model makes one further assumption, that the space and time discretizations are related:

$$\Delta x = u_f \Delta t \tag{2.21}$$

that is, the length of each cell is the distance a vehicle would travel at the free-flow speed. This simplifies calculations in two relatively intuitive ways, and in one deeper way. First, no vehicle can ever cross more than one cell boundary between ticks; to see how many vehicles may enter a cell during a time interval, one only need look at the cell immediately upstream. Second, if there is no congestion, *all* of the vehicles in each cell can proceed to the next. More deeply, the solution method presented next actually solves the system of partial differential equations (2.17)–(2.19) using what is known as a Godunov scheme; for such a scheme to be numerically stable, Δx and Δt must satisfy the Courant-Friedrichs-Lewy condition. One can show that choosing these values using equation (2.21) satisfies this condition, and thus proves the stability of the cell transmission model. The interested reader is referred to Godunov (1959) and Courant et al. (1928) for more details.

[2]For instance, consider a roadway where traffic is flowing at capacity. If a traffic signal turns red, the back of the queue will travel upstream at speed w.

Using this discretization, $n(x,t) = k(x,t)\Delta x$ and $y(x,t) = q(x,t)\Delta t$. We can then calculate $y(x,t)$ as follows, introducing $N(x,t) = k_j \Delta x$ as the maximum number of vehicles which can fit into a cell, and $\delta = w/u_f$ as the ratio of the backward wave speed to free-flow speed (typically 1/3–1/2):

$$
\begin{aligned}
y(x,t) &= q\left(x,t\right)\Delta t && (2.22)\\
&= Q\left(k\left(x,t\right)\right)\Delta t && (2.23)\\
&= Q\left(\frac{n\left(x,t\right)}{\Delta x}\right)\Delta t && (2.24)\\
&= \min\left\{u_f\frac{n\left(x,t\right)}{\Delta x}, Q_{max}, w\left(k_j - \frac{n\left(x,t\right)}{\Delta x}\right)\right\}\Delta t && (2.25)\\
&= \min\left\{u_f n(x,t)\frac{\Delta t}{\Delta x}, Q_{max}\Delta t, w\left(N(x,t) - n(x,t)\right)\frac{\Delta t}{\Delta x}\right\} && (2.26)\\
&= \min\left\{n(x,t), Q_{max}\Delta t, \delta\left(N(x,t) - n(x,t)\right)\right\} && (2.27)
\end{aligned}
$$

where the last equality is obtained using (2.21). Equation (2.27) is the key equation in the cell transmission model, and describes how many vehicles can pass through cell x at time t. Despite its somewhat intimidating appearance, the equation actually has a very intuitive interpretation. The flow through a cell can be limited by one of three things:

1. The number of vehicles in the cell (because no vehicle can travel more than one cell at a time).
2. The capacity of the cell.
3. The available space in the cell, if it is congested.

The three terms in the minimization in (2.21) correspond exactly to these three criteria: if the cell is uncongested, the maximum flow is $n(x,t)$, the number of vehicles in the cell; at all times, the cell capacity must be obeyed, and no more than $Q_{max}\Delta t$ vehicles may pass through; if the cell is congested, the amount of free space $N(x,t) - n(x,t)$ in the cell is what controls the flow. The goal of the cell transmission model, then, is to simulate the movement of vehicles from one cell to another, taking into account these three possible flow regimes (uncongested flow, capacity flow, and congested flow).

2.2.4 Traffic flow on a roadway link

Each roadway link is divided into a number of cells, based on the free flow time and time discretization. As a concrete example, consider a 30-second time discretization ($\Delta t = 30$ s) on a roadway which is one mile long and has a free-flow speed of 30 mph, capacity of 2000 vehicles per hour, and jam density of 240 vehicles per mile. At 30 mph, a vehicle can travel a quarter of a mile in 30 seconds; therefore, this link would be divided into four cells (Figure 2.8). Let $i \in \{1, 2, 3, 4\}$ index these four cells.

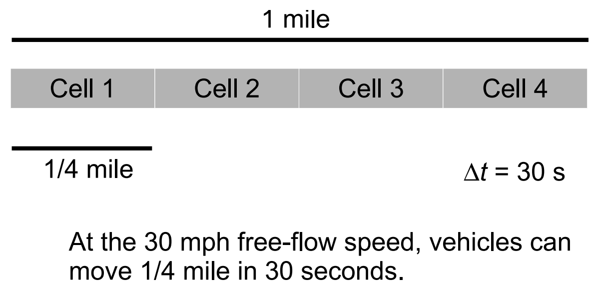


Figure 2.8: Division of a link into cells.

Let $y_{ij}(t)$ denote the number of vehicles which flow out of cell i and into cell j during time t. (Every vehicle must always be in a cell; it cannot exit cell i until it enters cell j.) Let $S_i(t)$ denote the *sending flow*, that is, the maximum number of vehicles which can leave cell i during time interval t

$$S_i(t) = \min\{n(i,t), Q_{max}\Delta t\} \tag{2.28}$$

and $R_i(t)$ the *receiving flow*, that is, the maximum number of vehicles which can enter cell i during time interval t:

$$R(i)(t) = \min\{\delta(N(i,t) - n(i,t)), Q_{max}\Delta t\} \tag{2.29}$$

Since a vehicle cannot flow from between cells i and j unless it can both be sent from i and received at j, we have

$$y_{ij}(t) = \min\{S_i(t), R_j(t)\} \tag{2.30}$$

The number of vehicles in each cell can then easily be calculated for the next time interval:

$$n(i, t+1) = n(i,t) + y_{i-1,i}(t) - y_{i,i+1} \tag{2.31}$$

that is, the previous number of vehicles in the cell, adding the number of vehicles which entered from the upstream cell, and subtracting the number of vehicles which left for the downstream cell. Note that flow into the first cell, and out of the last cell, cannot be handled by these formulas (since the cells $i-1$ or $i+1$ are out of range). Instead, they depend on the flow model at intersections, which is the topic of the next subsection.

2.2.5 Traffic flow at intersections

Roadway links meet at intersections, and the goal of intersection models is to move flow from the downstream ends of incoming roadway links onto the upstream ends of outgoing roadway links, accounting for any traffic control at the intersection (such as stop signs or traffic signals). There are two main ways to represent intersection flow: explicit simulation and implicit simulation. In the first method, explicit simulation, gap acceptance and the red and green indications of traffic signals are directly simulated in the model. With implicit simulation, the "average" effect of traffic control (given the current flow patterns and capacities) is simulated, without modeling each gap and change of signal phase. As Yperman (2007) explains, there are three major disadvantages to explicit simulation:

1. Explicit simulation can require a small value of Δt. For instance, if a short protected left-turn phase only provides 5 seconds of green, then Δt can be no greater than 5 seconds. Since small values of Δt greatly increase the computational burden of running a simulation model, even one intersection with a short signal phase can dramatically affect memory and time requirements (cf. Criterion 2) when using explicit simulation.

2. The main goal of mesoscopic simulation is to integrate traffic flow and driver behavior. However, when drivers choose routes, they base their decision on the *average* travel times, and do not time their departure time based on anticipated signal phases at arrival. (That is, no driver would think "I need to leave at 8:15 and 30 seconds so I can catch the signal at First and Main just as it turns green.") Explicitly simulating signal phases *over-optimizes* the model in the sense that it introduces more detail than drivers would actually account for in real life, which leads to unnecessary complication and increased computational burden.

3. Field conditions exhibit stochasticity and randomness, so that even if drivers wanted to time departures and arrivals exactly based on signal phasing, they would be unable to do so, due to fluctuations in driver speed, vehicles turning in and out of driveways, and so forth. Therefore it makes more sense to simulate average conditions.

For these reasons, the implicit simulation approach is preferred for mesoscopic simulation.

The remainder of this subsection describes how general intersection of any type are handled. Diverges (one incoming link, multiple outgoing links) are first discussed, followed by merges (multiple incoming links, one outgoing link and general intersections (with any number of incoming and outgoing links). Diverges and merges form the "prototype" models upon which the general intersection model is built. In all cases, the intersection models *only* depend on (a) the sending flow from the downstream cell on all incoming links; (b) the receiving flow for the upstream cells on all outgoing links; and (c) additional intersection-specific parameters (such as cycle length).

Diverges A diverge intersection is one with only one incoming link (labeled u, for upstream), but more than one outgoing link (here labeled 1 and 2), as in Figure 2.9. Our interest is calculating the rate of flow from the upstream link to the downstream ones, that is, the flow rates y_{u1} and y_{u2}.The upstream sending flow S_u and downstream receiving flows R_1 and R_2 will play a central role. For the first time, we also need to represent some model of *route choice*, since some drivers may choose link 1, and others link 2. Let p_1 and p_2 be the proportions of drivers choosing these two links, respectively. Naturally, p_1 and p_2 are nonnegative, and $p_1 + p_2 = 1$. These values can change with time.

There are two possibilities, one corresponding to free flow conditions at the diverge, and the other corresponding to congestion. For the diverge to be freely flowing, *both* of the downstream links must be able to accommodate the flow which seeks to enter them. The rates at which vehicles

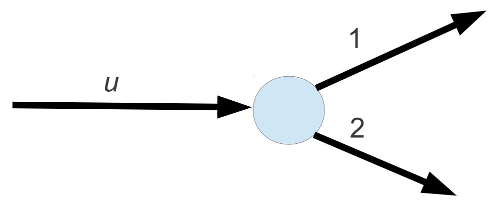


Figure 2.9: Prototype diverge intersection.

want to enter the two links are $p_1 S_u$ and $p_2 S_u$, so if both downstream links can accommodate this, we need $p_1 S_u \leq R_1$ and $p_2 S_u \leq R_2$. In this case we simply have $y_{u1} = p_1 S_u$ and $y_{u2} = p_2 S_u$: all of the flow which wants to leave the diverge can.

In the congested case, One common assumption is that flow waiting to enter one link at a diverge will obstruct every other vehicle on the link (regardless of which link it is destined for). This most obviously represents the case where the upstream link has only a single lane, so any vehicle which has to wait will block any vehicle behind it; but this model is commonly used even in other cases. When there is congestion, only some fraction ϕ of the upstream sending flow can move. The assumption that any vehicle waiting blocks every vehicle upstream implies that this same fraction applies to both of the downstream links, so $y_{u1} = \phi p_1 S_u$ and $y_{u2} = \phi p_2 S_u$.

The inflow rate to a link cannot exceeds its receiving flow, so $q_{u1} = \phi p_1 S_u \leq R_1$ and $q_{u2} = \phi p_2 S_u \leq R_2$, or equivalently $\phi \leq R_1 / p_1 S_u$ and $\phi \leq R_2 / p_2 S_u$. Every vehicle which can move will, so

$$\phi = \min \left\{ \frac{R_1}{p_1 S_u}, \frac{R_2}{p_2 S_u} \right\} \tag{2.32}$$

Furthermore, we can introduce the uncongested case into this equation as well, and state

$$\phi = \min \left\{ \frac{R_1}{p_1 S_u}, \frac{R_2}{p_2 S_u}, 1 \right\} \tag{2.33}$$

regardless of whether there is congestion at the diverge or not. If the diverge is at free flow, then $\phi = 1$, but $R_1 / p_1 S_u \geq 1$ and $R_2 / p_2 S_u \geq 1$. Introducing 1 into the minimum therefore gives the correct answer for free flow. Furthermore, if the diverge is not at free flow, then either $R_1 / p_1 S_u < 1$ or $R_2 / p_2 S_u < 1$, so adding 1 does not affect the minimum value. Therefore, this formula is still correct even in the uncongested case.

This formula easily generalizes to more than one outgoing link. If D is the set of downstream links (indexed by d), the general diverge flow formula is

$$y_{ud}(t) = p_d S_u \min_{d' \in D} \left\{ \frac{R'_d}{p'_d S_u}, 1 \right\} \qquad \forall d \in D \tag{2.34}$$

where $\sum_{d \in D} p_d = 1$ and $p_d \geq 0$ for all d.

Merges A merge intersection has only one outgoing link (labeled d, for downstream), but more than one incoming link (here labeled 1 and 2), as in Figure 2.10. We want to calculate the rate of

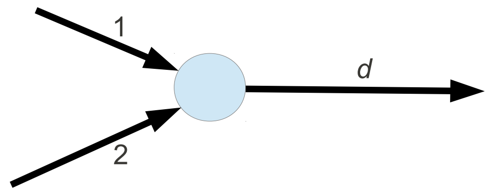


Figure 2.10: Prototype merge intersection.

flow from the upstream links to the downstream one, that is, the flows y_{1d} and y_{2d}. The main quantities of interest are the upstream sending flows S_1 and S_2, and the downstream receiving flow R_d. Unlike diverges, there is no route choice here, so there are no p values.

There are three possibilities, one corresponding to free flow conditions at the merge, one corresponding to congestion with queues growing on both upstream links, and one corresponding to congestion on only one upstream link. For the merge to be freely flowing, both upstream links must be able to transmit all of the flow which seeks to leave them, and the downstream link must be able to accommodate all of this flow. Mathematically, we need $S_1 + S_2 \leq R_d$, and if this is true then $q_{1d} = S_1$, and $q_{2d} = S_2$.

In the second case, there is congestion (so $S_1 + S_2 > R_d$), and furthermore, flow is arriving fast enough on both upstream links for a queue to form at each of them. Empirically, in such cases the flow rate from the upstream links is proportional to the capacity on these links, that is,

$$\frac{y_{1d}}{y_{2d}} = \frac{Q^1_{max}}{Q^2_{max}} \tag{2.35}$$

Furthermore, in the congested case, all of the available downstream capacity will be used, so

$$y_{1d} + y_{2d} = R_d \tag{2.36}$$

Substituting (2.35) into (2.36) and solving, we obtain

$$y_{id} = \frac{Q_i^{max}}{Q_1^{max} + Q_2^{max}} R_d \tag{2.37}$$

for $i \in \{1, 2\}$. Note that this method does not explicitly refer relative priorities, as might be given by a yield sign or roundabout control. The reason is that repeated field data show that vehicles tend to "take turns" merging during congestion, regardless of the rules of the road.

The third case is perhaps a bit unusual. The merge is congested ($S_1 + S_2 > R_d$), but a queue is only forming on one of the upstream links. This may happen if the flow on one of the upstream links is much less than the flow on the other. In this case, the proportionality rule allows all of the sending flow from one link to enter the downstream link, with room to spare. This "spare capacity" can then be consumed by the other approach. Mathematically, if one link cannot send enough flow to meet the proportionality condition, then for exactly one $i \in \{1, 2\}$ we have

$$y_{id} < \frac{Q_i^{max}}{Q_1^{max} + Q_2^{max}} R_d$$

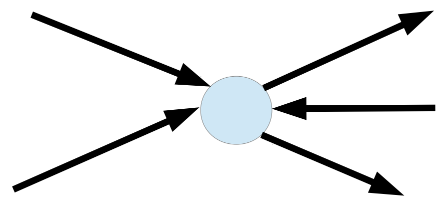


Figure 2.11: General intersection with multiple approaches and exits.

So, if j is the "other" link ($j \in \{1, 2\}$ but $j \neq i$), the two flow rates are $y_{id} = S_i$ and $y_{jd} = R_d - S_i$: one link sends all of the flow it can, and the other link consumes the remaining capacity.

The case of more than two upstream links is handled in a similar fashion. The objective is to determine which approaches (if any) will have queues; the approaches without queues can move their entire sending flow, while the approaches with queues can move sending flow in proportion to their capacity. While it is cumbersome to write analytical formulae for these cases, an iterative algorithm can determine these approaches quite efficiently, as described in Chapter 3.

General intersections General intersections consist of any number of incoming and outgoing links, and the procedure for determining flows for these are now described. Figure 2.11 shows an intersection with several incoming and outgoing links. Note that a *turning movement* cell has been created for every possible movement in this intersection. (If a turning movement does not connect an incoming link to an outgoing link, then that movement is considered forbidden, as with a turn prohibition).

Turning movement cells are slightly different than the ordinary cells which comprise roadway links, in several ways. First, the capacity of a turning movement cell can potentially change based on intersection conditions, to represent turning movements which must yield to another. Second, each turning movement has a *target delay*, that is, an average amount of time that vehicles are delayed when making this turning movement, due to intersection control (signal or stop). Note that this delay does *not* account for additional delay due to downstream congestion; it *only* measures delay due to the intersection control itself.

Each turning movement cell has its own sending flow — the number of vehicles in that cell that have been there longer than the target delay — and its own receiving flow, equal to the minimum of the turning movement's saturation flow, and that movement's share of the downstream roadway link's receiving flow. The intersection algorithm proceeds as follows:

1. Calculate the sending flow for each turning movement cell.
2. For each outgoing link j:
   (a) Calculate the receiving flow for link j.
   (b) Treat all turning movements ij with j as downstream link as a merge, calculate temporary flows y'_{ij}.

(c) Set the receiving flow of turning movement cell ij to y'_{ij}.

3. Load vehicles onto turn movement cells. Repeat the following steps for each incoming link i:
   (a) Calculate the sending flow for link i.
   (b) Treating link i as a diverge, move vehicles onto all turning movements ij with i as upstream link.
4. Move vehicles out of turn movement cells:
   (a) Recalculate the sending flow for each turning movement cell.
   (b) For each outgoing link j:
      i. Calculate the receiving flow for link j.
      ii. Treat all turning movements ij with j as downstream link as a merge, and move vehicles.

2.3 User behavior model

In contrast to the traffic flow model, the user behavior model is relatively simple, and is based on the principle of user equilibrium: if every driver is choosing a route that minimizes their own travel time, then all used routes connecting the same origin and destination will have equal and minimal travel time. Otherwise, one or more drivers would switch from a route with higher travel time to a route with a lower travel time, and the travel times on those two routes would tend to equalize.

Let Z denote the set of *centroids*, special intersections where all trips begin and end. (In software such as VISSIM, they are called “parking lots.”) Vehicles cannot pass through a centroid unless they are starting or ending there. Let r and s be centroids. The notation d^{rs} denotes the total number of vehicles departing centroid r for destination s during the analysis period. This value is provided as an input, and can be generated from a travel demand model such as that used by metropolitan planning organizations.

For mesoscopic simulation, this total demand must be *profiled*, or distributed, across the analysis period. Several different profiles are possible; see Figure 2.12 for a few possibilities. In the absence of further information, a uniform profile is a reasonable choice. In practice, field data can be used to calibrate the profile to match observed traffic counts or travel times. This profiling process produces values $d^{rs}(t)$, that is, the number of trips departing r for destination s *during time period* t; naturally $\sum_t d^{rs}(t) = d^{rs}$.

The user behavior model must now assign each of these $d^{rs}(t)$ vehicles to some route connecting centroids r and s, and do this for each time interval t. Using the notation from the macroscopic equilibrium formulation, let Π^{rs} denote the set of all routes connecting r to s, and let $h^\pi(t)$ denote the number of trips departing on path π at time t. Let $\tau^\pi(t)$ denote the travel time on route π for

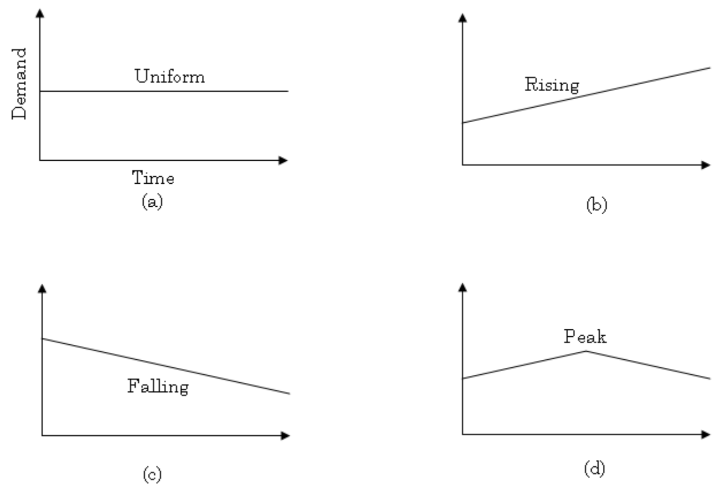


Figure 2.12: Four possible demand profiles.

travelers leaving at time t. Additionally define $\mathbf{h}$ and $\boldsymbol{\tau}$ to be the vectors of route flows and travel times across all r, s, and t.

Given any value of $\mathbf{h}$, the traffic flow model can determine $\boldsymbol{\tau}(\mathbf{h})$ — if the number of people leaving each route at each time is determined, calculating travel times is a simple matter of simulating the resulting traffic patterns (including any congestion) with the cell transmission model. The question now is how to update $\mathbf{h}$ to move towards an equilibrium solution. Since all drivers want to choose the route with the lowest travel time, we can define a "target" route flow vector $\mathbf{h}^*(\boldsymbol{\tau})$ in which every driver chooses the fastest route available to him or her based on travel times $\boldsymbol{\tau}$.

Of course, if all drivers switch to these target routes, the travel times will likely change and those routes will no longer have the fastest travel time. Instead, the solution is to switch only some drivers from their current routes onto the target routes. For any $\xi \in [0, 1]$, the vector $\xi\mathbf{h}^* + (1 - \xi)\mathbf{h}$ represents a set of route choice decisions intermediate between the current choices and the target choices, with ξ representing the proportion of drivers switching. (If $\xi = 0$, no drivers switch from their current routes; if $\xi = 1$, all drivers switch; if $\xi = 1/2$, half of drivers switch, and so forth.) In this way, we can avoid "overcorrecting" by moving too many people onto the same path at the same time, before recalculating travel times using the traffic flow model and making another route choice adjustment.

There are several compromises which must be made in choosing the right value of ξ. If you pick a value which is too small, very few drivers will switch paths, and the travel times will be very similar to what they were before. Unless the solution is already very close to an equilibrium, it will take many steps to find one, and each step will be spent simulating traffic flows very similar to those in the previous iteration. On the other hand, a value of ξ which is too large is potentially even worse: the solution could oscillate back and forth between two non-equilibrium states, each time overcorrecting and never settling to a stable case. The *method of successive averages* helps overcome these twin difficulties. In the method of successive averages, ξ changes according to the iteration number. During the first iteration, $\xi = 1/2$. During the second iteration, $\xi = 1/3$. During the third, $\xi = 1/4$, and so on. In general, during iteration k, $\xi = 1/(1+k)$. In this way, ξ starts relatively large, and becomes progressively smaller. From a behavioral standpoint, initially many drivers are switching routes, while over time fewer and fewer drivers switch routes. This strategy seeks to find an equilibrium within relatively few iterations by initially taking large steps, then reducing the step size as the equilibrium is approached.

Chapter 3

Implementation

This chapter describes how the methodology presented in Chapter 2 has been implemented in a mesoscopic simulation tool. This description is made in terms of the overall control flow of the simulation program, referencing each step in the simulator with an appropriate section from the methodology. This chapter does not contain a technical description of the C source code itself; programmers interested in a guide to the source code are referred to Appendix B.

Section 3.1 describes how the simulator works at a high level, and Section 3.2 provides additional details on the three major modules: the traffic flow (cell transmission) model; the fastest route model, which identifies the route with the least travel time between any points in the network at any departure time; and the route switching model, using the method of successive averages. Section 3.3 explains how the origin-destination table is profiled across different departure times. Section 3.4 introduces the input and output file formats; all of these are plain text files which can be edited using a number of free or commercial text editors. Sections 3.5 and 3.6 respectively describe the optional graphics modules (for creating image files based on simulation results). Finally, Section 3.7 describes the Microsoft Excel interface which allows the data files to be created, the simulation to be run, and output files to be read through a familiar user interface.

3.1 Overview

The simulator can be run either from the spreadsheet interface (by clicking on the `Run` button), or from the command line, by typing

```
wydot_dta parameters.txt
```

where the actual name of the parameters file is substituted for `parameters.txt`.

Figure 3.1 shows the basic workflow for the simulator. When first run, the program initializes by reading the following data files in sequence:

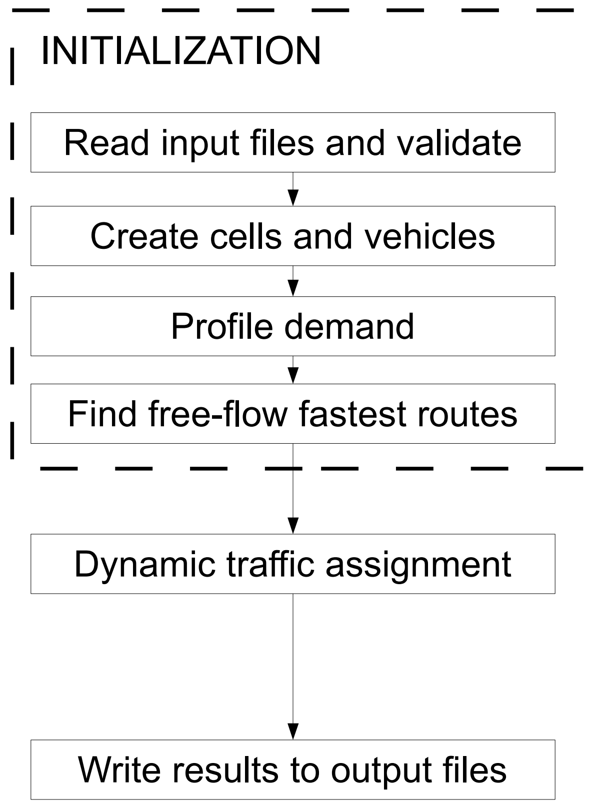


Figure 3.1: Overall simulation workflow.

1. The *parameters file*, which provides general information about the simulation run which is to be performed (such as the time step Δt and analysis period length), along with the names and locations of all the other input files.

2. The *network file*, which contains a list of every roadway link in the network and its characteristics (length, free-flow speed, etc.).

3. The *node coordinates file*, which contains a list of every intersection and its X and Y coordinates (which may be latitude and longitude, or another coordinate system).

4. The *intersection control file*, which provides detailed information on each intersection, including how it is controlled (signal, stop, etc.), which turning movements are allowed, and any additional information needed by the traffic flow model.

5. The *origin-destination (OD) matrix file*, which shows the total number of trips departing from every origin to every destination during the analysis period.

Next, the program prepares the internal structures needed for simulation: each link is divided into an appropriate number of cells, as is the list of vehicles which will eventually be assigned to the network. The OD matrix read from the input data is then profiled in a manner specified in the parameters file, converting it to a time-dependent ODT (origin-destination-departure time) array.

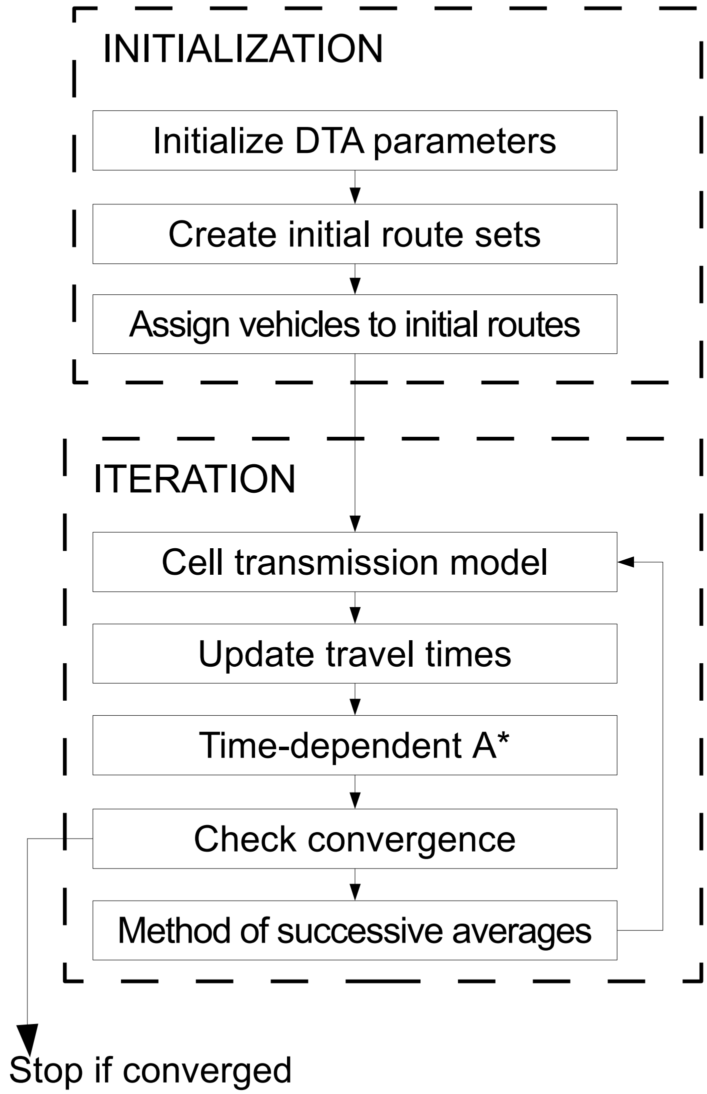


Figure 3.2: Dynamic traffic assignment workflow.

(Section 3.3 describes this process in greater detail.) As the final initialization step, the fastest route is calculated between every intersection and every destination, assuming free-flow conditions everywhere — precomputing this allows the simulation to run faster, even when network conditions are not free-flow.

After completing initialization, control switches to the main mesoscopic simulation model (termed *dynamic traffic assignment*). This is the core analysis routine which finds a mutually consistent equilibrium solution between the traffic flow and driver behavior models. The workflow for the dynamic traffic assignment process is shown in Figure 3.2. This process includes its own initialization procedures — setting initial travel conditions to free-flow everywhere, generating the initial route sets for each ODT based on the fastest free-flow routes, and assigning vehicles to these initial paths.

The dynamic traffic assignment process then begins in earnest, iterating through three primary modules — the cell transmission model, finding the fastest routes under updated conditions, and shifting some vehicles onto these routes. Each of these modules is described in detail in Section 3.2. Two additional steps are required: (1) after the cell transmission model is run, the travel time on each route must be recalculated, and (2) after each iteration of these three steps, a convergence check is performed.

Travel times for a route are calculated by adding the travel times of the constituent roadway links and turning movements. To calculate the travel time of a roadway link, the simulator makes use of the special cumulative counts $N^{\uparrow}(t)$ and $N^{\downarrow}(t)$, which respectively give the total number of vehicles which have passed the upstream and downstream ends of a link (or turn movement) by time t. To calculate the travel time for a vehicle entering at time t, we find that vehicle's exit time by finding the least integer t' such that $N^{\downarrow}(t') \geq N^{\uparrow}(t)$, and then set the travel time to the difference between t' and t (ensuring that this value is at least equal to the free-flow travel time).

The simulator uses three different convergence criteria, any or all of which can be set in the parameters file. (If multiple convergence criteria are satisfied, the simulation terminates as soon as any of them are reached.) These are:

Maximum time: If the simulation runs longer than a pre-specified time limit, it will terminate.

Maximum iterations: If the simulation has iterated among the three primary components more than a pre-specified number of times, it will terminate.

Consistency: If the simulation has achieved a certain consistency level between the traffic flow and user behavior models, it will terminate.

The first two are self-explanatory, while consistency requires further definition. Consistency is best measured against the equilibrium principle itself: at a perfectly consistent solution, all drivers are using the fastest routes available to them. So, if the solution is not perfectly consistent, one can measure the degree of consistency by comparing drivers' *actual* travel times to the travel times on the *fastest* routes they could choose from. This simulator measures this using the *average excess cost*, a metric defined in (Boyce et al., 2004) as the average difference between these values, across all vehicles.

3.2 Primary Modules

The three major modules are: (a) the traffic flow (cell transmission) model; (b) the fastest route model, which identifies the route with the least travel time between any points in the network at any departure time; and (c) the route switching model, using the method of successive averages. This section describes each of them in turn.

3.2.1 Cell transmission model

Sections 2.2.3–2.2.5 provided the mathematical formulation of the cell transmission model, link propagation, and intersection propagation models. By contrast, this section shows how these mathematical components are implemented in a computer model.

The following steps are performed in sequence, as shown in Figure 3.3:

1. Initially locate all vehicles on an artificial "origin" cell.
2. Start simulating at the initial time interval $t = 0$.
3. Identify vehicles departing at time interval t, and move them from the origin cell onto the first cell in their paths.
4. Calculate each link's sending and receiving flow at time t.
5. Move vehicles within link cells using the procedure in Section 2.2.4.
6. Process each intersection using the procedure in Section 2.2.5, moving vehicles into and out of turning movement cells.
7. If this is the last time interval, stop; otherwise increment t and return to step 2.

Although the same general procedure is used to process each intersection, there are several distinct intersection types which differ in how the target delay and capacity for each turning movement are calculated. These intersection types are as follows: internal, diverge, merge, four-way stop, two-way stop, signal, and interchange. Each is described as follows:

Internal: Internal intersections only have one upstream and one downstream link, are thus are not "true" intersections. However, they can be used to divide one roadway link into two, as in Figure 3.4. This may be useful to model a lane drop or other change in a link which does not occur at an intersection — each roadway link must be homogeneous in terms of capacity and jam density, so internal nodes can be used to mark internal points. Additional information is calculated at intersections, and so internal intersections can be inserted wherever this information would be needed. Internal intersections only have a single turning movement, with zero target delay, and their capacity is not binding[1].

Diverge: A diverge has one upstream link and multiple downstream links. Diverge turning movements have zero target delay and their capacity is not binding.

Merge: A merge has one downstream link and multiple upstream links. Diverge turning movements have zero target delay and their capacity is not binding.

[1]That is, flow can be restricted either by the capacity of the upstream or downstream links, but the turn movement does not impose any further restrictions

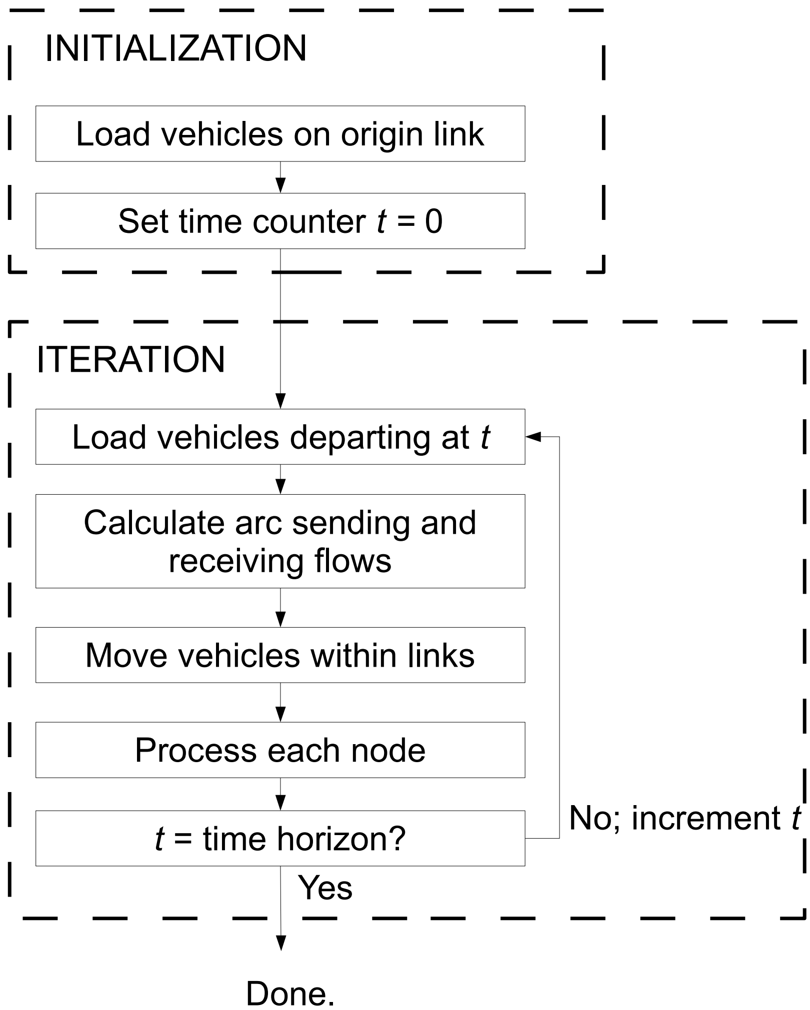


Figure 3.3: Cell transmission model workflow.

Four-way stop: Four-way stops can have multiple upstream and downstream links. They have a target delay specified as a simulation parameter (the default value is four seconds, representing time lost due to deceleration and stopping, not time spent in queue), and the capacity is specified in the intersection control file.

Two-way stop: For a two-way stop, each turn movement has two pieces of information stored with it: a saturation flow, and a priority value, indicating its relative priority in the intersection. Priority 1 movements do not need to yield to any other movement (for instance, through and right turns on the major approaches). Priority 2 movements only need to yield to Priority 1 movements (typically left turns on the major approaches); Priority 3 movements need to yield to Priority 1 and 2 movements (such as right turns on the minor approach), and so forth. Additionally, the "minimum stop priority" and "intersection capacity" must be specified. The minimum stop priority is the lowest priority movement which has a stop sign. Lower priority movements have zero target delay, while higher priority movements have the same target delay as the four way stop. When this intersection is processed, vehicles begin moving from the Priority 1 movements. If there is intersection capacity remaining, vehicles from Priority 2 movements can enter the intersection, and so on. Once the intersection capacity is exhausted, no more vehicles can move during this time interval.

Signal: For a signalized intersection, each turn movement has two pieces of information stored with it: a saturation flow, and an effective green value. Furthermore, the cycle length must be specified in the intersection control file. Target delays are calculated using the uniform delay formula in the Highway Capacity Manual (Transportation Research Board, 2010):

$$d = \frac{C}{2}\left(\frac{(1 - G/C)^2}{1 - \min\{X, 1\}G/C}\right) \tag{3.1}$$

where G is the effective green, C is the cycle length, and X is the degree of saturation (number of vehicles in the turning movement cell divided by the saturation flow and proportion of green time). No incremental delay component is needed because the dynamic traffic simulation accounts for fluctuations in arrival rate during the analysis period.

Interchange: An interchange has zero target delay for any of the turning movements, which are assumed to be grade separated. Capacities can still be specified in the intersection control file.

3.2.2 Fastest route model

Each ODT has a set of "working routes" that its vehicles are assigned to. Initially, this set only consists of the fastest route under free-flow conditions. At each iteration, the simulator finds the new fastest route for this ODT, accounting for updated travel conditions. This route is then added to the set of working routes (unless it was already there, in which case the set of working routes remains unchanged).

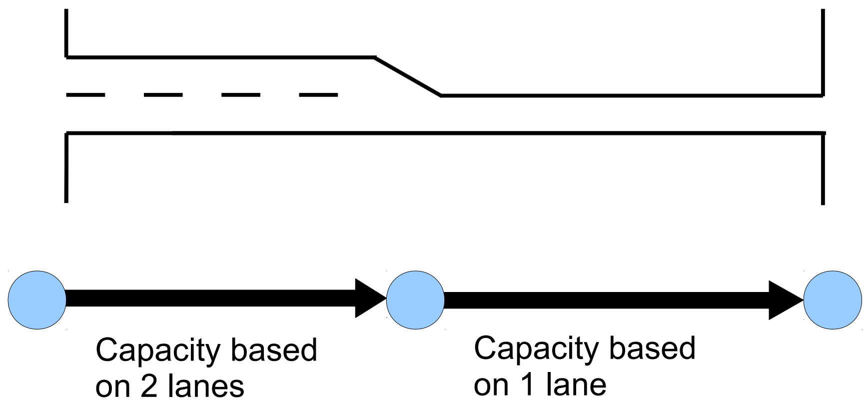


Figure 3.4: Dividing a link with a nonhomogeneous intersection.

To find these fastest routes, the simulator uses a time-dependent version of the A* shortest path algorithm, which was first developed by Hart et al. (1968) as a faster version of Dijkstra's classical shortest path algorithm. Intuitively, Dijkstra's algorithm searches in all directions from the origin r until the destination s is found, while A* uses a more directed search towards the destination. This directed search relies on having a lower bound on the travel time between any intersection and the destination. An efficient choice for this lower bound is the free-flow travel time between that intersection and the destination, which was calculated in one of the initialization steps.

This method proceeds using the following steps, and two additional data structures: a binary heap $\mathcal{H}$ of intersections to scan, and a set $\mathcal{C}$ of intersections for which the fastest route from r has already been found.

1. Initialize $\mathcal{H}$ by inserting r, and give it a label of zero.
2. Let i be the intersection in $\mathcal{H}$ with the least label; remove it from $\mathcal{H}$.
3. Scan i by performing the following steps for each intersection j directly connected to i by a single roadway link:
   (a) Calculate a temporary label $temp$ by adding (a) the travel time on the fastest route from r to i; (b) the travel time from i to j; and (c) the free-flow travel time from j to s. (Figure 3.5)
   (b) If j is not in $\mathcal{H}$, insert it into $\mathcal{H}$ with the label $temp$.
   (c) If j is already in $\mathcal{H}$ but its label is greater than $temp$, update the label of j to $temp$.
4. Add i to $\mathcal{C}$
5. If $\mathcal{H}$ is empty or s was just added to $\mathcal{C}$, end. Otherwise, return to step 2.

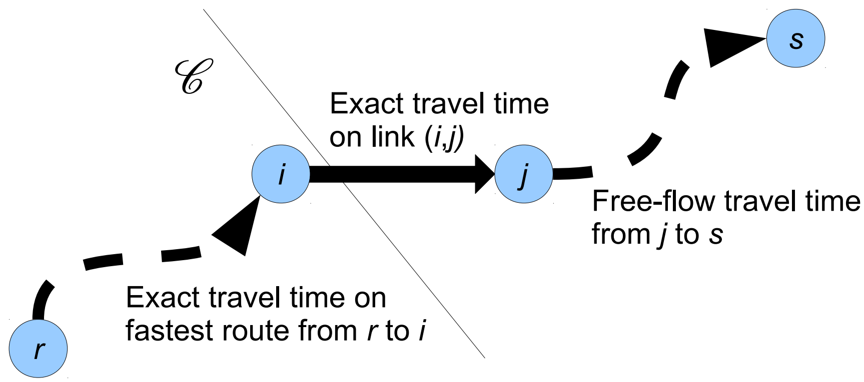


Figure 3.5: Label calculation in time-dependent A*.

3.2.3 Route switching model

As stated in Section 2.3, the method of successive averages moves a fraction of vehicles from their current paths onto the fastest path identified by the time-dependent A* algorithm. However, the total number of vehicles in an ODT can be somewhat small, and this fractional limit can be binding. For instance, at the fourth iteration, 1/5 of the vehicles should be moved onto the new fastest path. If there are only 3 vehicles corresponding to this ODT, it is unclear how this move should be performed. Therefore, the simulator uses a stochastic formula. Each vehicle moves to its ODT's fastest route with a probability of ξ, with ξ given by the formulas in Section 2.3, independent of any other vehicle's move.

3.3 Demand Profiling

Demand profiling is the process of converting a "static" or aggregate OD matrix into a time-dependent ODT array. That is, for every origin and destination r and s, we seek time-dependent departure rates $d^{rs}(t)$ such that $\sum_t d^{rs}(t) = d^{rs}$. One important simulator parameter is t_L, the time at which the last vehicle enters the network. After t_L, no additional vehicles are loaded, but the simulator completes the trips of any vehicles which have been loaded in earlier time intervals.

The simulator can profile demand automatically, using two predefined "shapes" specified in the parameters file. The predefined shapes are as follows:

Uniform: This is the simplest profile; demand is distributed evenly between the start of simulation and t_L; there is no peak, rise, or fall in the demand.

Triangle: This forms a "triangular" profile that can represent increasing, decreasing, or peaking behavior. The triangle profile consists of two linear pieces, and requires three parameters

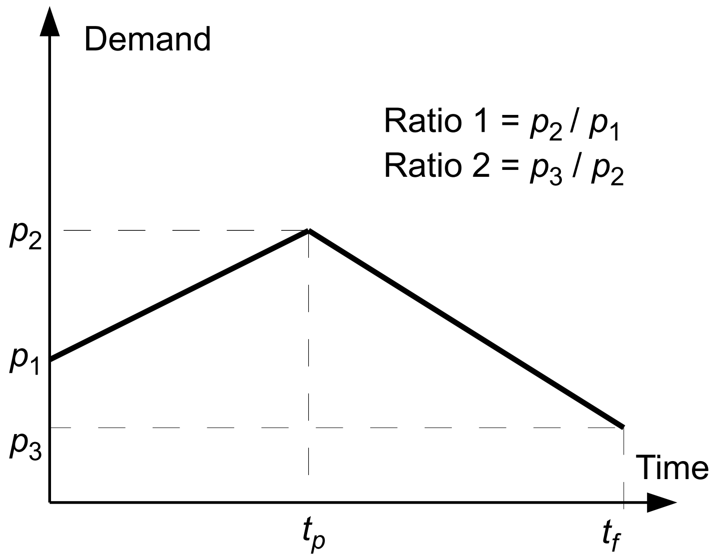


Figure 3.6: Parameter values for triangle profile.

(Figure 3.6): the *peak time* t_p; the *first ratio* R_1, and the *second ratio* R_2. The slope switches from the initial slope to the terminal slope at the peak time. Furthermore, the ratio of the demand during the peak interval during the demand at the first interval is given by R_1, and the ratio between the demand during the peak interval and the final loading interval t_f is given by R_2. This profile is flexible: by specifying a peak time of either 0 or t_f, a linear profile can be created with slope given by R_2 or R_1, respectively.

Alternatively, a "raw ODT" file can be specified which lists ODTs and flow rates explicitly. This option is more data-intensive, but can produce more reliable results if a precomputed profile is already available (perhaps from an activity-based or departure-time model).

3.4 File Formats

This section documents the formats for each input and output file used by the simulator. All of these files are plain text files to maximize ease of editing and portability. While plain text files require more disk space than binary formats, all of these files are amenable to compression and can be greatly reduced in size when not in use.

Many of these files use a standard metadata format; for instance, a typical line in the parameters file is

```
<NETWORK FILE NAME>  my_network.txt
```

The text enclosed in angle brackets is the *metadata tag* (in this case, `NETWORK FILE NAME`), and the remainder of the line is the *metadata value* (`my_network.txt`). The metadata tag is case insensitive, but the metadata value is case sensitive. All whitespace between the metadata tag and metadata value is stripped, but whitespace after the metadata value is retained. Metadata can be presented in any order in the files. In files which contain both metadata and other forms of data (such as the network file), the metadata must be presented first, and ended with a `<END OF METADATA>` tag.

A tilde (~) denotes a comment; any text after a tilde on a line is ignored.

3.4.1 Parameters file

The parameters file contains general values passed to the simulation, including the names of all of the other input and output files. The name of the parameters file is passed to the simulator as a command-line argument. This file only consists of metadata. Table 3.1 shows all of the metadata tags, noting which ones are required and which are optional, along with their functions. An example of a metadata file is as follows:

```
<NETWORK FILE>          braess.net
<DEMAND FILE>           braess.ods
<NODE COORDINATE FILE>  braess.nxy
<NODE CONTROL FILE>     braess.icf
<COUNTS FILE>           braess.sum
<TIME HORIZON>          3000
<LAST VEHICLE ON>       100
<AEC TOLERANCE>         0.1
<MAX RUN TIME>          60
<VERBOSITY LEVEL>       4
<TICK LENGTH>           10
<DEMAND PROFILE>        UNIFORM
<BACKWARD WAVE RATIO>   0.5
<RANDOM SEED>           0
```

Table 3.1: Metadata fields for the parameters file.

Metadata tag	Required?	Function
`NETWORK FILE`	Yes	Provides the name (and optional path) to the network file
`DEMAND FILE`	Yes	Provides the name (and optional path) to the OD matrix file, or to the raw ODT file if the demand profile is `RAW`
`NODE COORDINATE FILE`	Yes	Provides the name (and optional path) to the node coordinate file
`NODE CONTROL FILE`	Yes	Provides the name (and optional path) to the file containing intersection data

Continued on next page.

Metadata fields for the parameters file (continued)

Metadata tag	Required?	Function
`COUNTS FILE`	No	Provides the name (and optional path) for the file where complete link and turning movement cumulative counts will be reported. (This field can also be referenced with the `SUMMARY FILE` metadata tag, but this usage is not recommended.)
`LINK SUMMARY FILE`	No	Provides the name (and optional path) for the file where summary information for each link will be recorded.
`NODE SUMMARY FILE`	No	Provides the name (and optional path) for the file where summary information for each turning movement will be recorded.
`TIME HORIZON`	Yes	Provides the duration of time which will be simulated, in seconds.
`TICK LENGTH`	No	Provides the length of the simulation time step Δt, in seconds. If not provided, will default to 6 seconds.
`LAST VEHICLE ON`	Yes	Provides the time t_f at which the last vehicle will be loaded, in seconds.
`WARM UP PERIOD`	No	Provides the duration of the warm-up period, in seconds from the start of simulation; results in this period will not be counted in the link or node summary files. (This field is required if either a link or node summary file is present).
`COOL DOWN PERIOD`	No	Provides the duration of the cool-down period, in seconds before the time horizon; results in this period will not be counted in the link or node summary files. (This field is required if either a link or node summary file is present).
`AEC TOLERANCE`	No†	Average excess cost which will be used to declare the simulation sufficiently consistent (Section 3.1).
`MAX RUN TIME`	No†	Maximum run time before the simulation terminates, in seconds (Section 3.1).
`MAX ITERATIONS`	No†	Maximum number of dynamic traffic assignment iterations before the simulation terminates (Section 3.1).
`DEMAND MULTIPLIER`	No	Allows the user to scale all travel demand levels by the factor specified here. Useful for sensitivity analyses.
`VERBOSITY LEVEL`	No	Controls the amount of on-screen notifications provided by specifying an integer from 0 (nothing) to 5 (complete logging). The default verbosity level is 3. **Warning:** Specifying a verbosity level of 4 or 5 will create a number of additional log files. These files may be very large (over 1 GB) and greatly slow the simulation. A verbosity level of 3 or lower is recommended.

Continued on next page.

Metadata fields for the parameters file (continued)

Metadata tag	Required?	Function
VEHICLE LENGTH	No	Average effective length of a vehicle, in feet. Default value is 20 ft.
BACKWARD WAVE RATIO	No	The ratio δ between the backward and forward wave speeds (Section 3.2.1). Default value is 0.5.
RANDOM SEED	No	Provides a seed value for the random number generator, useful for replicating runs exactly. Default value is the current system time (which will generally produce slightly different results at each run).
DEMAND PROFILE	Yes	Either UNIFORM, TRIANGLE, or RAW. (Section 3.3)
PEAK DEMAND TIME	No	The peak time t_p for triangle profiles (Section 3.3). Required if the demand profile is TRIANGLE
RATIO 1	No	The first ratio R_1 for triangle profiles (Section 3.3). Required if the demand profile is TRIANGLE
RATIO 2	No	The second ratio R_2 for triangle profiles (Section 3.3). Required if the demand profile is TRIANGLE

† At least one of the three convergence criteria must be specified, or the simulator will run forever.

3.4.2 Network file

The network file consists of two components: a set of metadata fields which describe the size of the network (number of links, intersections, and zones), followed by a list of records, one per roadway link, specifying its parameters. Table 3.2 describes these metadata fields. A sample network file looks like this:

```
<NUMBER OF ZONES>    2
<NUMBER OF NODES>    6
<NUMBER OF LINKS>    7
<END OF METADATA>

~ Init node Term node Capacity Length (ft) u_f (mph) k_j(veh/mi);
   1         3          500      1500        30        200        ;
   3         4          50       1500        30        200        ;
   3         5          50       1500        30        200        ;
   4         5          50       1500        30        200        ;
   4         6          50       1500        30        200        ;
   5         6          50       1500        30        200        ;
   6         2          50       1500        30        200        ;
```

Following the metadata section, each roadway link record consists of one line of text, with six numbers and a semicolon, all separated by whitespace. These six numbers indicate (1) the

Table 3.2: Metadata fields for the network file.

Metadata tag	Required?	Function
`NUMBER OF NODES`	Yes	Number of intersections in the network
`NUMBER OF LINKS`	Yes	Number of roadway links in the network
`NUMBER OF ZONES`	Yes	Number of origin and destination centroids

intersection where the link starts; (2) the intersection where the link ends; (3) the capacity of the roadway link, in vehicles per hour; (4) the length of the link, in feet; (5) the free-flow speed u_f on the link, in miles per hour; and (6) the jam density k_j in vehicles per mile. Roadway links do not have to be listed in any particular order.

The intersections are numbered from 1 to `NUMBER OF NODES`, and textbfcentroid intersections must be listed first. That is, intersections 1–`NUMBER OF ZONES` correspond to centroids, and the remaining intersections are regular ones.

3.4.3 Node coordinate file

The node coordinate file has no metadata, and contains a list of coordinates for each intersection. Each intersection has one row, consisting of three numbers and a semicolon, all separated by whitespace. These three numbers are (1) the intersection ID; (2) . Intersections do not need to be listed in order of their ID.

```
~Node    X     Y  ;
1        0     0  ;
2        100   0  ;
3        10    0  ;
4        20    -10;
5        20    10 ;
6        30    0  ;
```

3.4.4 Demand matrix

The demand matrix file specifies the d^{rs} values, that is, the total number of vehicles departing each origin r for each destination s during the analysis period. This file contains three optional metadata (shown in Table 3.3), followed by `<END OF METADATA>`. Even if none of the optional metadata are used, the `<END OF METADATA>` line must still be included prior to the beginning of the OD matrix. This file has the following format:

```
<NUMBER OF ZONES>    2
<TOTAL OD FLOW>      100.0
```

Table 3.3: Metadata fields for the demand matrix file.

Metadata tag	Required?	Function
NUMBER OF ZONES	No	Number of origin and destination centroids. If included, it will be checked against the value in the network file for consistency. Including this optional field can be useful to catch errors when running simulations, in case the chosen OD matrix and network are not from the same simulation run.
DEMAND MULTIPLIER	No	Allows the user to scale all travel demand levels by the factor specified here. Useful for sensitivity analyses. (This will override any value provided in the parameters file.)
TOTAL OD FLOW	No	Lists the total number of vehicles to be assigned to the network. Providing this optional field can be useful for checking that the OD matrix has been correctly specified. If the simulation VERBOSITY is at least 3, it will report the TOTAL OD FLOW value compared to the actual number of vehicles shown in the OD matrix.

```
<DEMAND MULTIPLIER> 6
<END OF METADATA>

Origin   1
    1 :      0.0;     2 :     100.0;

Origin   2
    1 :      5.0;
```

The demand matrix is organized by origin; all of the vehicles leaving a particular origin X are listed in the section following `Origin X`. Each destination corresponding to that origin is listed, followed by colon, the number of vehicles, and a semicolon. So, in the example provided above, 0 vehicles are departing origin 1 for destination 1; 100 vehicles are departing origin 1 for destination 2; and 5 vehicles are departing origin 2 for destination 1. Not all destinations need to be provided; the simulator will assume a zero value for any origin-destination pairs not listed in this file. Fractional values can be used (these values can be thought of as “average” flow rates).

An alternative specification is the raw demand file, which is provided instead of the demand matrix file if the demand profile has `RAW` type. This file contains the following metadata fields, followed by one row for each ODT:

```
<NUMBER OF ODTS> 3
<NUMBER OF ZONES> 7
<TOTAL OD FLOW> 5000.000000
<DEMAND MULTIPLIER> 1
```

Table 3.4: Metadata fields for the raw demand file.

Metadata tag	Required?	Function
NUMBER OF ODTS	Yes	Provides the total number of ODTs (origin-destination-departure time combinations) in the network.
NUMBER OF ZONES	No	Number of origin and destination centroids. If included, it will be checked against the value in the network file for consistency. Including this optional field can be useful to catch errors when running simulations, in case the chosen OD matrix and network are not from the same simulation run.
DEMAND MULTIPLIER	No	Allows the user to scale all travel demand levels by the factor specified here. Useful for sensitivity analyses. (This will override any value provided in the parameters file.)
TOTAL OD FLOW	No	Lists the total number of vehicles to be assigned to the network. Providing this optional field can be useful for checking that the OD matrix has been correctly specified. If the simulation VERBOSITY is at least 3, it will report the TOTAL OD FLOW value compared to the actual number of vehicles shown in the OD matrix.

```
<END OF METADATA>

~ Origin Destination DepartureTime Demand
  1      5           1             100
  2      6           5             100
  3      7           10            100
```

Each row contains four numbers, separated by whitespace: the origin, the destination, the departure time, and the number of vehicles which are departing. The relevant metadata fields are specified in Table 3.4. Note that using this file would raise a warning, because the TOTAL OD FLOW metadata shows 5000 vehicles, which does not match the 300 vehicles which are actually listed in the remainder of the file. This allows errors to be caught before a simulation is run.

3.4.5 Intersection control file

The intersection control file provides detailed information on each intersection, the form of traffic control there, and an explicit list of the turning movements associated with that intersection. An example of such a file is below:

```
Node 1 : CENTROID
Node 2 : CENTROID
```

```
Node 3 : DIVERGE
   1 -> 3 -> 4       9999
   1 -> 3 -> 5       9999
Node 4 : DIVERGE
   3 -> 4 -> 5       9999
   3 -> 4 -> 6       9999
Node 5 : MERGE
   3 -> 5 -> 6       9999
   4 -> 5 -> 6       9999
Node 6: MERGE
   4 -> 6 -> 2       9999
   5 -> 6 -> 2       9999
```

Each intersection is introduced as `Node X : CONTROL` where `CONTROL` is the control type for this intersection. The allowable control types are: `FOUR-WAY-STOP`, `INTERCHANGE`, `TWO-WAY-STOP`, `BASIC-SIGNAL`, `CENTROID`, `MERGE`, `DIVERGE`, tttNONHOMOGENEOUS, and `UNKNOWN`. The `UNKNOWN` control should only be used in conjunction with the warrant analysis module (Section 3.6). Following this is a list of all allowable turning movements at this intersection. Every allowable movement must be listed — any movement *not* listed will not be included in the simulation. Each turning movement is signified using the notation `X -> Y -> Z`, which reflects turning from the link connecting intersection `X` to intersection `Y`, onto the link connecting `Y` to intersection `Z`. The specific format for each node type is slightly different.

For most forms of control, the name of each turning movement is followed simply by its saturation flow in vehicles per hour. This holds true for all forms of control except for `TWO-WAY-STOP` and `BASIC-SIGNAL`. For these latter two, additional information is needed on the intersection control and on each turning movement. A sample entry for a `TWO-WAY-STOP` intersection is as follows:

```
Node 8 :  TWO-WAY-STOP
   Intersection saturation flow 4999.999809
   Minimum stop priority 3
   1 -> 8 -> 9    1  4999.999809
   1 -> 8 -> 11   1  4999.999809
   7 -> 8 -> 9    2  4999.999809
   7 -> 8 -> 11   2  4999.999809
   9 -> 8 -> 11   3  4999.999809
   11 -> 8 -> 9   3  4999.999809
```

Two pieces of intersection data are required: the `Intersection saturation flow`, giving the maximum rate at which vehicles can pass through the intersection as a whole, and the `Minimum stop priority`, the least priority movement which has to stop at the sign. For each listed movement, two pieces of data are required: the stop priority corresponding to that movement, and the saturation flow.

A sample entry for a `BASIC-SIGNAL` intersection is as follows:

```
Node 11 :  BASIC-SIGNAL
   Cycle length 20
   6 -> 11 -> 5   3  4999.999809
   6 -> 11 -> 8   3  4999.999809
   6 -> 11 -> 10  3  4999.999809
   8 -> 11 -> 5   17 4999.999809
   8 -> 11 -> 6   17 4999.999809
   8 -> 11 -> 10  17 4999.999809
   10 -> 11 -> 5  17 4999.999809
   10 -> 11 -> 6  17 4999.999809
   10 -> 11 -> 8  17 4999.999809
```

Once piece of intersection data is required: the `Cycle length` for this signal. For each listed movement, two pieces of data are required: the effective green time, and the saturation flow.

3.4.6 Counts file

The counts file is an output file which reports the $N^{\uparrow}(t)$ and $N^{\downarrow}(t)$ cumulative count values and travel times for each roadway link and turning movement at all times t. **This file can be very large, and is not intended to be read directly.** Instead, this file stores all of the simulation results which will be used later, either as summarized information in the node and link summary files (Sections 3.4.8 and 3.4.7), to generate graphics, or to perform a warrant analysis.

The first half of the file reports cumulative counts and travel times for each link, and the second half reports cumulative counts and travel times for each turning movement. This information is presented in tabular form, with rows corresponding to a particular time interval and columns corresponding to roadway links (in the order given in the network file). An excerpt of such a file is shown below:

```
LINK CUMULATIVE COUNTS
----------------------
t   (1,3) Downstream  Time  (3,4) Downstream  Time  ...
10 2     0            50   0     0            40    ...
20 4     0            60   0     0            40    ...
30 6     0            70   0     0            40    ...
...
2990 343 295          3010 0     0            3010  ...
3000 344 296          3010 0     0            3010  ...

TURN MOVEMENT CUMULATIVE COUNTS
-------------------------------
t  1->3->5  Downstream  Time  1->3->4  Downstream  Time ...
10 0        0           0     0        0           0    ...
20 0        0           0     0        0           0    ...
...
```

Note that links are indicated by their upstream and downstream intersections (so `(1,3))` is the link connecting intersection 1 to intersection 3, in that direction), and that turning movements are indicated by three intersections (`1 -> 3 -> 5` is the turning movement from link `(1,3)` to link `3,5`). There are three columns corresponding to each link or turning movement: the first shows the upstream count, the second the downstream count, and the third the travel time for vehicles entering at the time corresponding to that row (which is shown in the very first column labeled `t`).

3.4.7 Link summary file

This output file summarizes the simulation results for each roadway link, in the order given in the network file. For each link, the average travel time is reported in seconds, along with the average delay in seconds (that is, the travel time in excess of the free-flow time), along with the average density (in vehicles per mile), the average volume (in vehicles per hour) and the peak-hour factor. These can be used as input for an operational analysis, such as that in the Highway Capacity Manual. An excerpt of this file is shown below:

```
LINK SUMMARY (ALL VALUES TIME AVERAGES)
---------------------------------------
Link  Travel time (s)   Delay (s)   Density (veh/mi)  Volume (veh/hr)   PHF
(1,8)    100            0           13                86                0.53
(2,18)   346            246         66                89                0.62
(3,28)   565            465         91                89                0.76
(4,38)   575            475         86                89                0.80
(187,5)  100            0           11                73                0.47
(197,6)  100            0           8                 57                0.43
```

3.4.8 Node summary file

This output file summarizes the simulation results for each turning movement, organized by the intersection that the turning movements are associated with. For each movement, the average delay is reported (in seconds), along with the average volume (in vehicles per hour) and the peak-hour factor. These can be used as input for an operational analysis, such as that in the Highway Capacity Manual, or for signal retiming. An excerpt of this file is shown below:

```
NODE SUMMARY FILE
-----------------
   Movement          Delay (s)   Volume (vph)   PHF
Node 1 summary
   0 -> 1 -> 8       242         82             0.50
...
Node 18 summary
   108 -> 18 -> 118  0           33             0.50
```

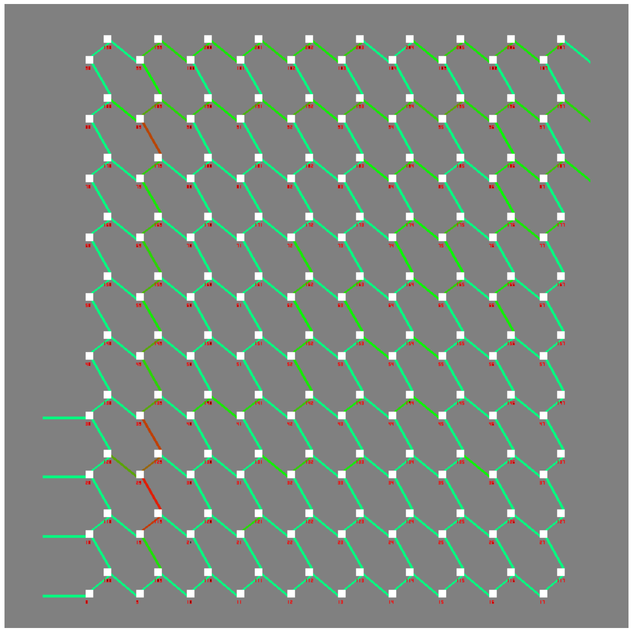


Figure 3.7: Example image for a grid network.

```
   2 -> 18 -> 118    0           89             0.62
Node 19 summary
   118 -> 19 -> 119  0           63             0.66
   109 -> 19 -> 119  0           50             0.53
Node 20 summary
   119 -> 20 -> 120  0           20             0.36
   110 -> 20 -> 120  0           4              0.25
...
```

3.5 Graphics Module

The graphics module produces image files corresponding to a simulation run, as shown in Figure 3.7. Graphics can be produced either based on the *average* level of congestion on each roadway link, or based on a *snapshot* level of congestion at each simulation tick. **These graphics files are made in the Portable Network Graphics (PNG) format, and to use this module you must have the `libpng3.dll` and `zlib.dll` libraries installed on your computer.** This format stores images in a space-efficient manner using lossless compression.

To produce graphics, the graphics module must be run separately, either from the spreadsheet or from the command line, by typing

```
wydot_graphics parameters.txt
```

where the actual name of the parameters file is substituted for `parameters.txt`. Furthermore,

Table 3.5: Metadata fields for the graphics parameters file.

Metadata tag	Required?	Function
IMAGE WIDTH	No	The width of the PNG file to create, in pixels. Default value is 500.
IMAGE HEIGHT	No	The height of the PNG file to create, in pixels. Default value is 500.
BORDER WIDTH	No	Width of the black border on all sides of the image, in pixels. Default value is 50.
NODE RADIUS	No	Each intersection is marked with a square, whose width (in pixels) is twice the value given for this metadata. Default value is 5.
LINK WIDTH	No	Each link is drawn as a straight line, whose width (in pixels) is the value given for this metadata. Default value is 2.
PNG ROOT	Yes	The start of the filename for the PNG file; the program will append `final.png` to this label.
SNAPSHOT	No	If this metadata tag is included, the code will generate a snapshot for each time interval, rather than a single "average" graphic. (Metadata value is ignored.)
EXCLUDE ZONES	No	If this metadata tag is included, the code will not plot centroid intersections or connectors. (Metadata value is ignored).

to produce graphics, the parameters file must contain the metadata `<GRAPHICS PARAMETERS FILE>`, whose corresponding metadata value is the name of the graphics parameters file, an additional text file. This file only consists of metadata; Table 3.5 gives a complete listing of the fields. An example of the graphics parameters file is as follows:

```
<IMAGE WIDTH>  800
<IMAGE HEIGHT> 800
<BORDER WIDTH> 50
<NODE RADIUS>  5
<LINK WIDTH>   3
<PNG ROOT>     gridgen2_
```

Each image uses color to depict the level of congestion on a link. Red indicates the most congested conditions, yellow indicates moderate congestion, and green indicates free-flow conditions. Continuous shading is used to reflect conditions in between. Congestion is based on the average density on each link, with jam density corresponding to red and zero density corresponding to green. This information is obtained by reading the counts file indicated in the parameters file.

3.6 Warrants Module

The warrants module uses the simulation results to determine what form of intersection control is warranted based on predicted traffic volumes. It can be used in one of two ways: (1) to generate an initial intersection control file when no data is available; and (2) as a post-processing step, to recommend an updated control scheme for re-optimizing the network based on predicted flows.

This module can only be called from the command line. The two modes of operation for this module are distinguished by the number of arguments given on the command line. For the first usage, to perform a comprehensive warrant analysis, the following command line pattern is given with four arguments:

```
wydot_warrant parametersFile networkFile initialICF finalICF
```

where `parametersFile` is the simulation parameters file, `networkFile` is the network file which will be used for executing the calibration run, `initialICF` is the file identifying which nodes will have the warrant analysis conducted, and `finalICF` is the name of the intersection control file which will include the warrant analysis. Only those intersections which are given `UNKNOWN` control in the `initialICF` file will have a warrant analysis conducted. In this case, an initial simulation is made with `INTERCHANGE` control at all nodes, determining the flow pattern if there were no delay at the nodes. These volumes are then used to perform a warrant analysis.

For the second usage, the following command line pattern is used with three arguments:

```
wydot_warrant parametersFile nodeNumber outputFile
```

where `parametersFile` is the simulation parameters file (whose `<COUNTS FILE>` has the simulation results), `nodeNumber` is the ID number for the intersection to analyze, and `outputFile` is the name of the file into which the new control data will be written.

In both cases, the warrants in the Manual for Uniform Traffic Control Devices (Federal Highway Administration, 2009) are used to classify each node's control. The first step is to identify the primary and secondary approaches to the node; this determination is made on the basis of the flow rates from the simulation counts file. Note that the primary approaches need not correspond to a through movement, but can correspond to a right or left turn (Figure 3.8). The appropriate form of signal control is then determined using tables from the Manual for Uniform Traffic Control Devices, to determine whether a signal, two-way stop, or four-way stop control is warranted. If none of these warrants are satisfied, four-way stop control is used as default.

For signalized intersections, a basic, two-phase signal timing is created using Webster's formula Webster (1958):

- Calculate the degree of saturation X_M for the major approach, based on the ratio of average volume to saturation flow.

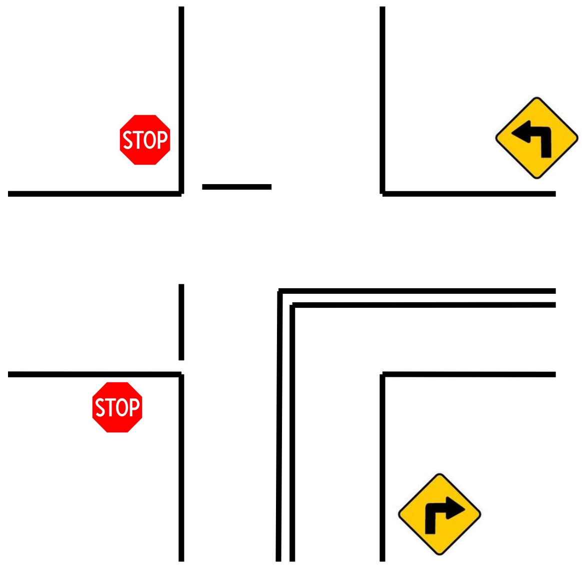


Figure 3.8: An intersection where the primary movements correspond to a turn.

- Calculate the degree of saturation X_m for the minor approach.
- Calculate the cycle length using the formula

$$C = \min\left\{\frac{5}{1 - X_M - X_m}, C_{max}\right\} \tag{3.2}$$

  where C is the calculated cycle length, and C_{max} is the maximum allowable cycle length. If $X_M + X_m \geq 1$, then $C = C_{max}$.

- Allocate green time to the major and minor approaches in proportion to X_m and X_M.

This procedure assumes no lost time, makes no saturation flow adjustments based on local conditions (such as grade or the presence of parking), and does not account for pedestrian crossing times. If this information is available, then it should be entered into the intersection control file directly, because it cannot be calculated by the warrant module.

For two-way stops, each turn movement is classified based on whether it is a right turn or left turn, and whether its upstream and downstream roadway links are major and/or minor. The node coordinates are used to determine the angle for the turning movement, and to assign the correct priority.

For four-way stops, no additional calculations are needed.

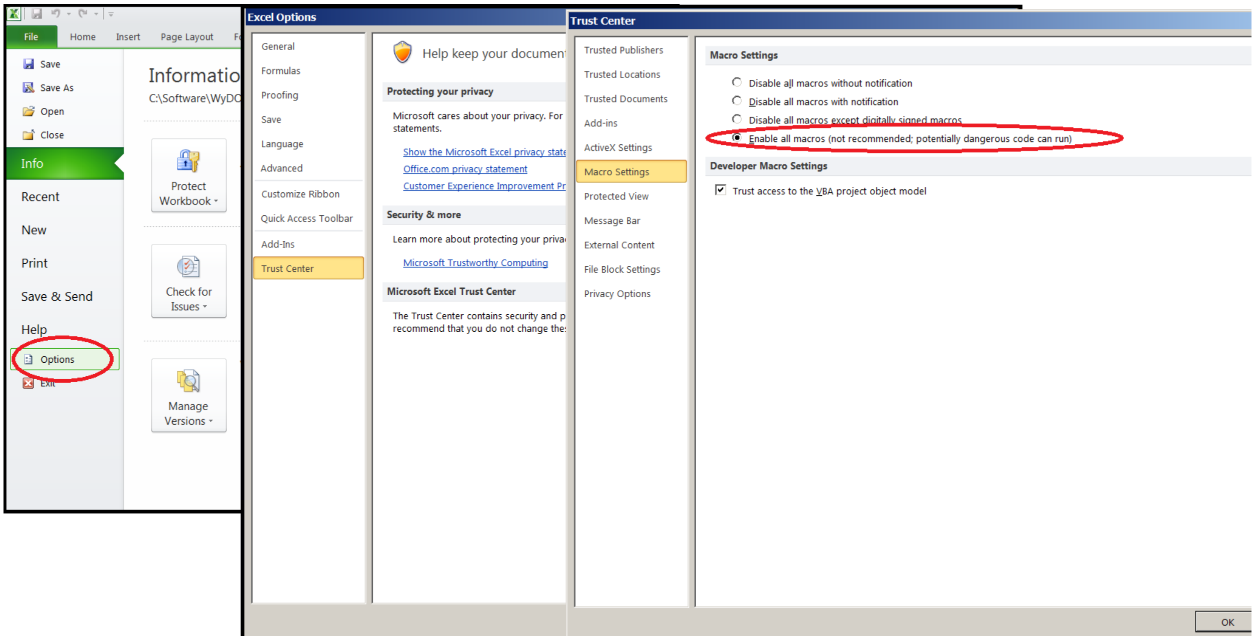


Figure 3.9: Changing macro settings in Excel.

3.7 Spreadsheet Interface

A spreadsheet interface to the simulator has been created, using Microsoft Excel and its VBA scripting language. **To use this spreadsheet, macros must be enabled in Excel.** To change the security settings to allow macros to run, click on the File tab at the upper left of the ribbon, and click "Excel Options." Click on "Trust Center" at the bottom of the left panel, then choose "Macro Settings." Select "Enable all macros." See Figure 3.9 (You should switch your macro settings back to their initial settings after you have finished using the interface, by performing the same steps and choosing a different security level.)

The interface contains a number of different worksheets. Upon first opening the interface, you will be on the Dashboard worksheet. (Figure 3.10). The buttons on this worksheet walk you through the steps of preparing the necessary input data for the simulator, running the simulator itself, and then importing the simulation summary files and graphics. You can always return to the Dashboard from any worksheet by pressing a button at the upper left.

The Project Summary worksheet is the first step in creating a new network. (Figure 3.11). On this sheet, there is room to provide information about the analysis project, as well as the engineer performing the analysis. The network data is specified below, including the number of nodes (intersections), links, and zones. Simulation parameters are listed below; these will eventually be entered into the appropriate parameters file for a simulation run. To the right is a listing of all of the input and output files which can be created by the simulator. You do not need to enter these manually; when you export the data, the VBA code will prompt you to enter the name of each file in turn. Once all of the parameters have been entered, clicking the "Prepare for manual input" button will then format the remaining sheets based on the network parameters that have been

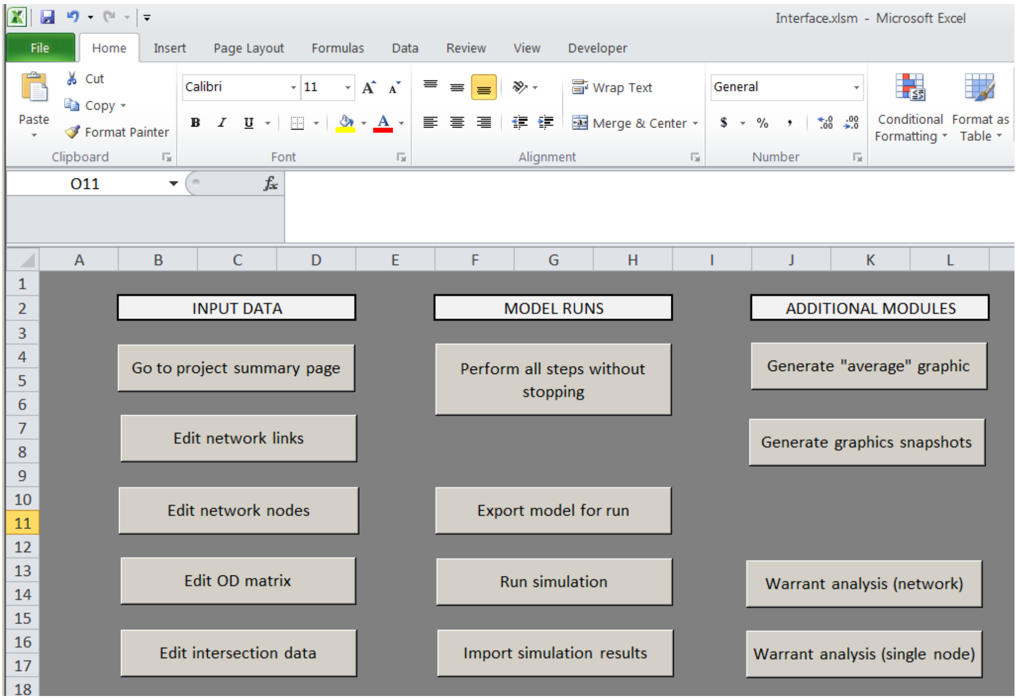


Figure 3.10: Dashboard spreadsheet.

entered.

The Link Data spreadsheet allows you to enter information for each roadway link in the network. (Figure 3.12). This file will eventually be exported to form the network file (Section 3.4.2), and each column corresponds to the descriptions in that section. Upon entering this data, click on “Proceed to node data.”

The Node Data spreadsheet allows you to enter the coordinates for each intersection in the network. (Figure 3.13). This file will eventually be exported to form the node coordinates file (Section 3.4.3), and each column corresponds to the descriptions in that section. Upon entering this data, click on “Proceed to OD Matrix.”

The OD Matrix spreadsheet allows you to enter the coordinates for each intersection in the network. (Figure 3.14). This file will eventually be exported to form the demand matrix file (Section 3.4.4). Upon entering this data, click on “Proceed to intersection data.”

The Intersection Data spreadsheet allows you to enter information on each intersection’s control type. (Figure 3.15). This spreadsheet is slightly more involved than the others. The intersection type can be selected from a drop-down box next to each node’s ID. By default, the spreadsheet will allow all turning movements at an intersection, except for U-turns. The columns labeled “Forbidden movements” and “Permitted movements” allow you to override these defaults: prohibited turn movements can be entered in the “Forbidden movements” column, and if U-turns are allowed, they can be added to the “Permitted movements” column. These movements are indicated using the standard `X -> Y -> Z` notation, and a list of movements can be separated by commas in these cells. Clicking on “Edit node Y details” takes you to a secondary sheet which

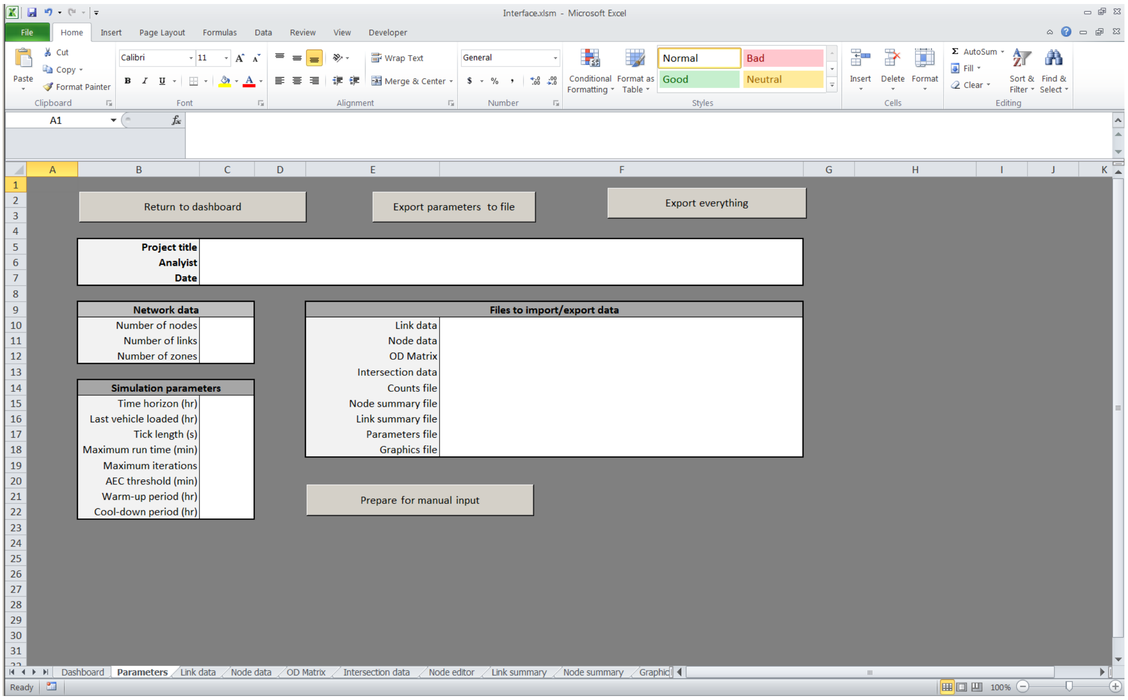
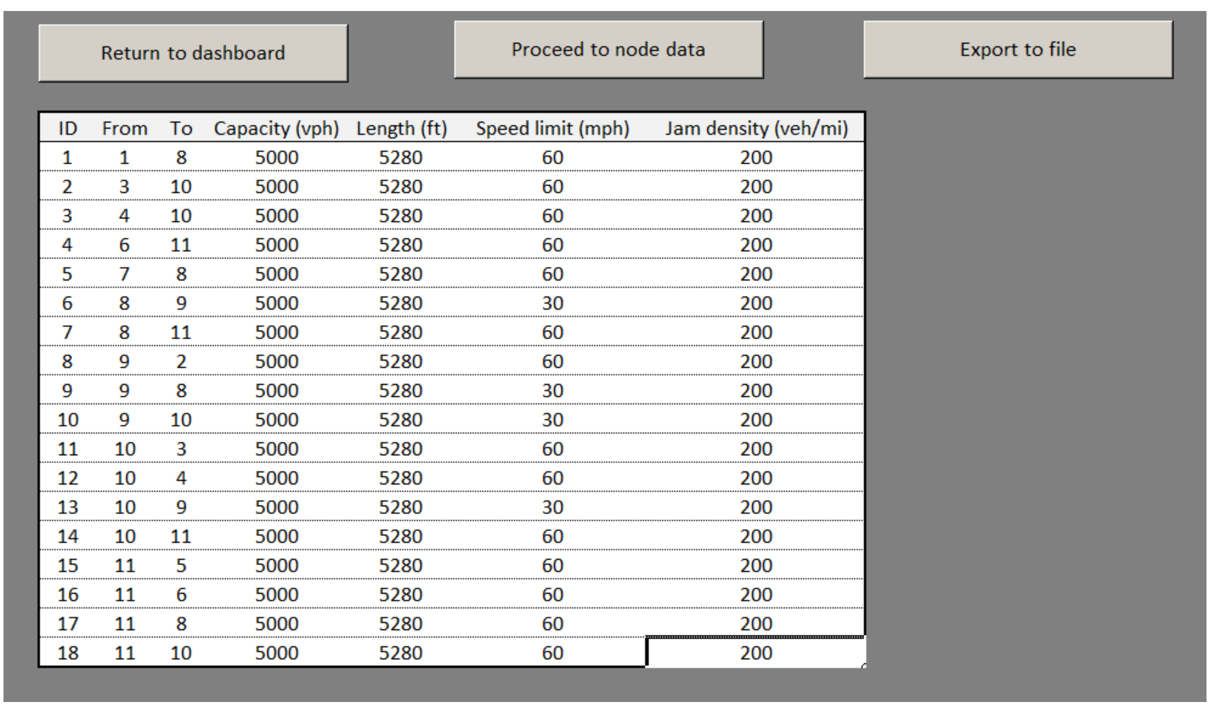


Figure 3.11: Project Summary spreadsheet.



ID	From	To	Capacity (vph)	Length (ft)	Speed limit (mph)	Jam density (veh/mi)
1	1	8	5000	5280	60	200
2	3	10	5000	5280	60	200
3	4	10	5000	5280	60	200
4	6	11	5000	5280	60	200
5	7	8	5000	5280	60	200
6	8	9	5000	5280	30	200
7	8	11	5000	5280	60	200
8	9	2	5000	5280	60	200
9	9	8	5000	5280	30	200
10	9	10	5000	5280	30	200
11	10	3	5000	5280	60	200
12	10	4	5000	5280	60	200
13	10	9	5000	5280	30	200
14	10	11	5000	5280	60	200
15	11	5	5000	5280	60	200
16	11	6	5000	5280	60	200
17	11	8	5000	5280	60	200
18	11	10	5000	5280	60	200

Figure 3.12: Links spreadsheet.

Return to dashboard

Proceed to OD Matrix

Node	Longitude	Latitude
1	10	0
2	20	0
3	30	30
4	20	30
5	10	30
6	0	20
7	0	10
8	10	10
9	20	10
10	20	20
11	10	20

Figure 3.13: Node Data spreadsheet.

Return to dashboard

Proceed to intersection data

Orig | Dest	1	2	3	4	5	6	7
1			3000				
2							
3						500	
4		100					
5							
6			500				
7		200					

Figure 3.14: OD Matrix spreadsheet.

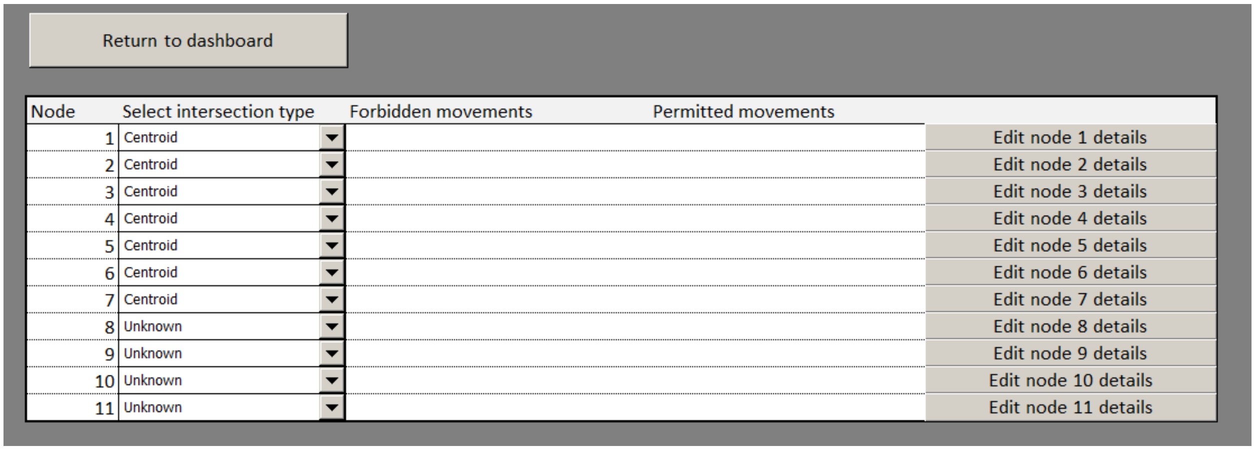


Figure 3.15: Intersection Data spreadsheet.

allows you to enter additional information on each turn movement and intersection (such as cycle lengths, movement saturation flows, green times, and priorities). (Figure 3.16). This latter sheet also allows you to toggle movements between permitted and prohibited. Upon completion of editing, you must click “Save information and return to intersection data” for this information to be properly saved.

After entering all of this data, return to the Dashboard and click “Export model for run.” You will be prompted for all the names of the files to export all of the information which has been entered into the spreadsheets. At this point, you can click “Run simulation” to perform mesoscopic simulation — the program will run in the same way as if it had been called from the command line. After simulation is complete, clicking on “Import simulation results” will enter the link and intersection summary data into their respective sheets (Figures 3.18 and 3.17), along with a graphical illustration (Figure 3.19).

Save information and return to intersection data

Node	11	details

Cycle length	30

Movement	Permitted?	Saturation flow	Effective green
6 -> 11 -> 5	Yes	5000	17
6 -> 11 -> 6	No	0	0
6 -> 11 -> 8	Yes	5000	17
6 -> 11 -> 10	Yes	5000	17
8 -> 11 -> 5	Yes	5000	17
8 -> 11 -> 6	Yes	5000	17
8 -> 11 -> 8	No	0	0
8 -> 11 -> 10	Yes	5000	17
10 -> 11 -> 5	Yes	5000	13
10 -> 11 -> 6	Yes	5000	13
10 -> 11 -> 8	Yes	5000	13
10 -> 11 -> 10	No	0	0

Figure 3.16: Node Data spreadsheet.

ID	From	To	Travel time (s)	Delay (s)	Density (veh/mi)	Volume (vph)	PHF
1	1	8	60	0	25	1533	0.75
2	3	10	60	0	4	255	0.73
3	4	10	60	0	1	44	0.53
4	6	11	60	0	4	251	0.72
5	7	8	60	0	2	104	0.7
6	8	9	120	0	4	108	0.72
7	8	11	60	0	26	1571	0.77
8	9	2	60	0	3	161	0.72
9	9	8	120	0	0	0	---
10	9	10	120	0	0	0	---
11	10	3	60	0	32	1910	0.8
12	10	4	60	0	0	0	---
13	10	9	120	0	2	49	0.59
14	10	11	60	0	4	261	0.74
15	11	5	60	0	0	0	---
16	11	6	60	0	4	267	0.76
17	11	8	60	0	0	0	---
18	11	10	61	1	31	1862	0.78

Figure 3.17: Link Summary spreadsheet.

From	->	Node	->	To	Delay (s)	Volume (vph)	PHF
0	->	1	->	8	0	1499	0.74
9	->	2	->	0	0	161	0.72
10	->	3	->	0	0	1910	0.8
0	->	3	->	10	0	249	0.71
10	->	4	->	0	0	0	---
0	->	4	->	10	0	43	0.52
11	->	5	->	0	0	0	---
11	->	6	->	0	0	267	0.76
0	->	6	->	11	0	246	0.7
0	->	7	->	8	0	103	0.69
11	->	8	->	9	0	0	---
9	->	8	->	11	0	0	---

Figure 3.18: Node Summary spreadsheet.

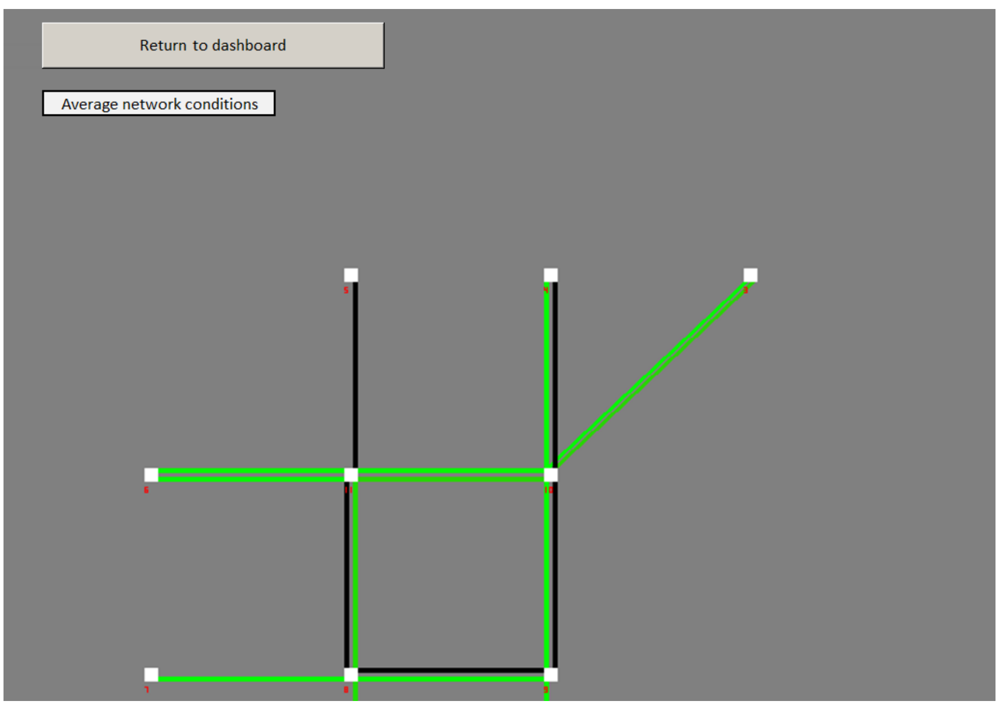


Figure 3.19: Graphics spreadsheet.

Chapter 4

Case Studies

This chapter presents three different networks as case studies for the mesoscopic simulator: a small "toy" network, the city of Casper, and the state of Wyoming. Each case study demonstrates a different aspect of the simulator and modules. The toy network is small enough for the impact of its results to be immediately apparent; this further serves as a demonstration of the warrant analysis. The Casper network represents one of Wyoming's largest cities, and demonstrates the applicability of the tool for quantifying traffic diversion due to a work zone closing a major thoroughfare. The Wyoming network represents the major freeways throughout the entire state, and forms the context for quantifying traffic diversion due to a hypothetical toll imposed on I-80.

In this chapter, the focus is mainly on the network constructions, and interpreting the result summary and graphics files. Detailed tutorials, with step-by-step instructions for performing these analysis, can be found in Appendix A.

4.1 Toy Network

Consider the network shown in Figure 4.1. Intersections 1–7 are centroids. Every link is 1 mile long and has a capacity of 5000 vph and jam density 200 veh/mi; the thick shaded links have a speed limit of 30 mph and all other links have a speed limit of 60 mph. 3000 vehicles depart from zone 1 to zone 3; 500 vehicles from zone to 3 to zone 6; 100 from zone 4 to zone 2; 500 from 6 to 3; and 200 from 7 to 2. Assume that the node controls at intersections 8, 9, 10, and 11 is currently unknown. This leads to the following incomplete intersection control file:

```
Node 1 : CENTROID
Node 2 : CENTROID
Node 3 : CENTROID
Node 4 : CENTROID
Node 5 : CENTROID
Node 6 : CENTROID
```

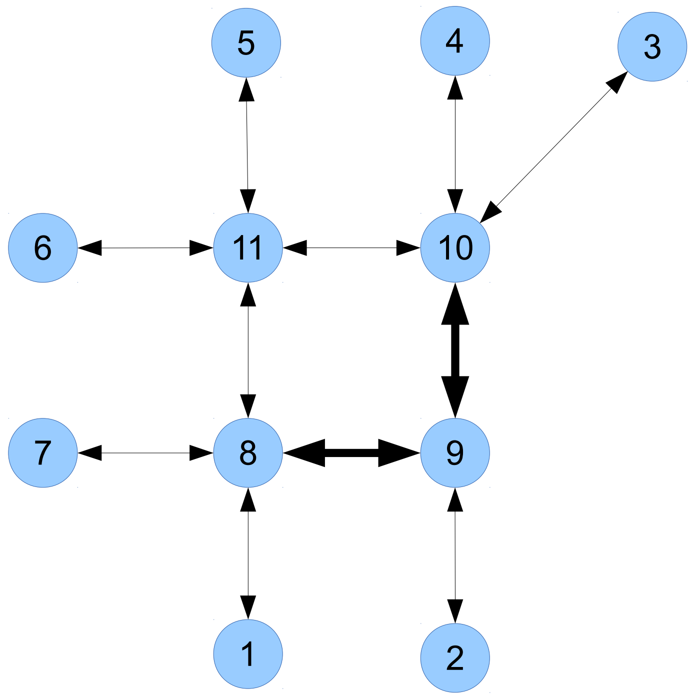


Figure 4.1: Toy network schematic.

```
Node 7 : CENTROID
Node 8 : UNKNOWN
Node 9 : UNKNOWN
Node 10 : UNKNOWN
Node 11 : UNKNOWN
```

Running the warrant analysis module, volumes are calculated on a temporary basis, assuming that each of nodes 8–11 has four-way stop control. Upon analyzing these volumes, the warrant analysis module generates the following, completed intersection control file:

```
Node 1 :  CENTROID
Node 2 :  CENTROID
Node 3 :  CENTROID
Node 4 :  CENTROID
Node 5 :  CENTROID
Node 6 :  CENTROID
Node 7 :  CENTROID
Node 8 :  TWO-WAY-STOP
   Intersection saturation flow 4999.999809
   Minimum stop priority 3
   1 -> 8 -> 9 1  4999.999809
   1 -> 8 -> 11   1  4999.999809
   7 -> 8 -> 9 2  4999.999809
```

```
   7 -> 8 -> 11   2  4999.999809
   9 -> 8 -> 11   3  4999.999809
   11 -> 8 -> 9   3  4999.999809
Node 9 :  BASIC-SIGNAL
   Cycle length 20
   8 -> 9 -> 2 10 4999.999809
   8 -> 9 -> 10   10 4999.999809
   10 -> 9 -> 2   10 4999.999809
   10 -> 9 -> 8   10 4999.999809
Node 10 :  BASIC-SIGNAL
   Cycle length 20
   3 -> 10 -> 4   16 4999.999809
   3 -> 10 -> 9   16 4999.999809
   3 -> 10 -> 11  16 4999.999809
   4 -> 10 -> 3   16 4999.999809
   4 -> 10 -> 9   16 4999.999809
   4 -> 10 -> 11  16 4999.999809
   9 -> 10 -> 3   4  4999.999809
   9 -> 10 -> 4   4  4999.999809
   9 -> 10 -> 11  4  4999.999809
   11 -> 10 -> 3  4  4999.999809
   11 -> 10 -> 4  4  4999.999809
   11 -> 10 -> 9  4  4999.999809
Node 11 :  BASIC-SIGNAL
   Cycle length 20
   6 -> 11 -> 5   15 4999.999809
   6 -> 11 -> 8   15 4999.999809
   6 -> 11 -> 10  15 4999.999809
   8 -> 11 -> 5   15 4999.999809
   8 -> 11 -> 6   15 4999.999809
   8 -> 11 -> 10  15 4999.999809
   10 -> 11 -> 5  5  4999.999809
   10 -> 11 -> 6  5  4999.999809
   10 -> 11 -> 8  5  4999.999809
```

Notice that a two-way stop is recommended at node 8, and signals at nodes 9, 10, and 11. Each signal has a short cycle length (20 seconds). At node 9, the signal time is divided between the two approaches, while nodes 10 and 11 use a less even distribution because of different approach volumes. Running the main simulator produces the following link flows:

```
LINK SUMMARY (ALL VALUES TIME AVERAGES)
---------------------------------------
Link   Time (s) Delay (s)   Density  Volume   PHF
(1,8)     60    0           39       2357     0.92
(3,10)    60    0           7        398      0.90
(4,10)    60    0           1        81       0.85
...
```

and turning movement summary:

```
NODE SUMMARY FILE
-----------------
   Movement Delay (s)   Volume (vph)   PHF
Node 1 summary
   0 -> 1 -> 8 0  2314  0.90
Node 2 summary
   9 -> 2 -> 0 0  261   0.92
Node 3 summary
   10 -> 3 -> 0   0  2894  0.96
   0 -> 3 -> 10   0  391   0.88
```

4.2 Casper

The Casper network was constructed from the TransCAD model used for long-range planning. After making the necessary conversions to mesoscopic simulation format, this network has 304 centroids, 1014 intersections, and 2760 roadway links. The entire network is shown in Figure 4.2 — in this and similar figures, the color indicates the level of congestion on a link, based on its density. Green indicates free-flow conditions, red congested conditions, and intermediate shades reflect traffic conditions in between. Black indicates links with very little flow. To simulate a work zone closure, the link representing Yellowstone Highway between Beverly St and C St is closed (represented in the simulator by deleting its link from the network file) (Figure 4.3).

Figure 4.4 shows how roadway volume has shifted before (left) and after (right) the work zone closure. For a more quantitative view, the node summary file shows how the flows around intersection 436 (Yellowstone Hwy & Beverly St) have changed, as compiled in Table 4.1. Note the substantial diversion of flow away from this intersection after the closure. This reflects travelers choosing alternate routes to avoid the work zone and closure.

4.3 Wyoming

The Wyoming network was constructed manually, including every interstate and federal highway in the State, along with several "external" intersections and links which represent diversion opportunities outside the state (Figure 4.5). More details on the construction process can be found in Saha et al. (2013). The specific policy studied here was levying a toll on I-80 between Rock Springs and Rawlins, as suggested by Wyoming Department of Transportation (2008).

While the simulator does not directly model tolls on links, a toll can be effectively modeled by adjusting the free-flow speed of a link, as follows: to simulate a $20 toll, we increase the free-flow travel time on this link by an equivalent amount. This equivalency depends on travelers' value of time; this demonstration assumes an average $10/hr value of time, a standard figure in the tolling

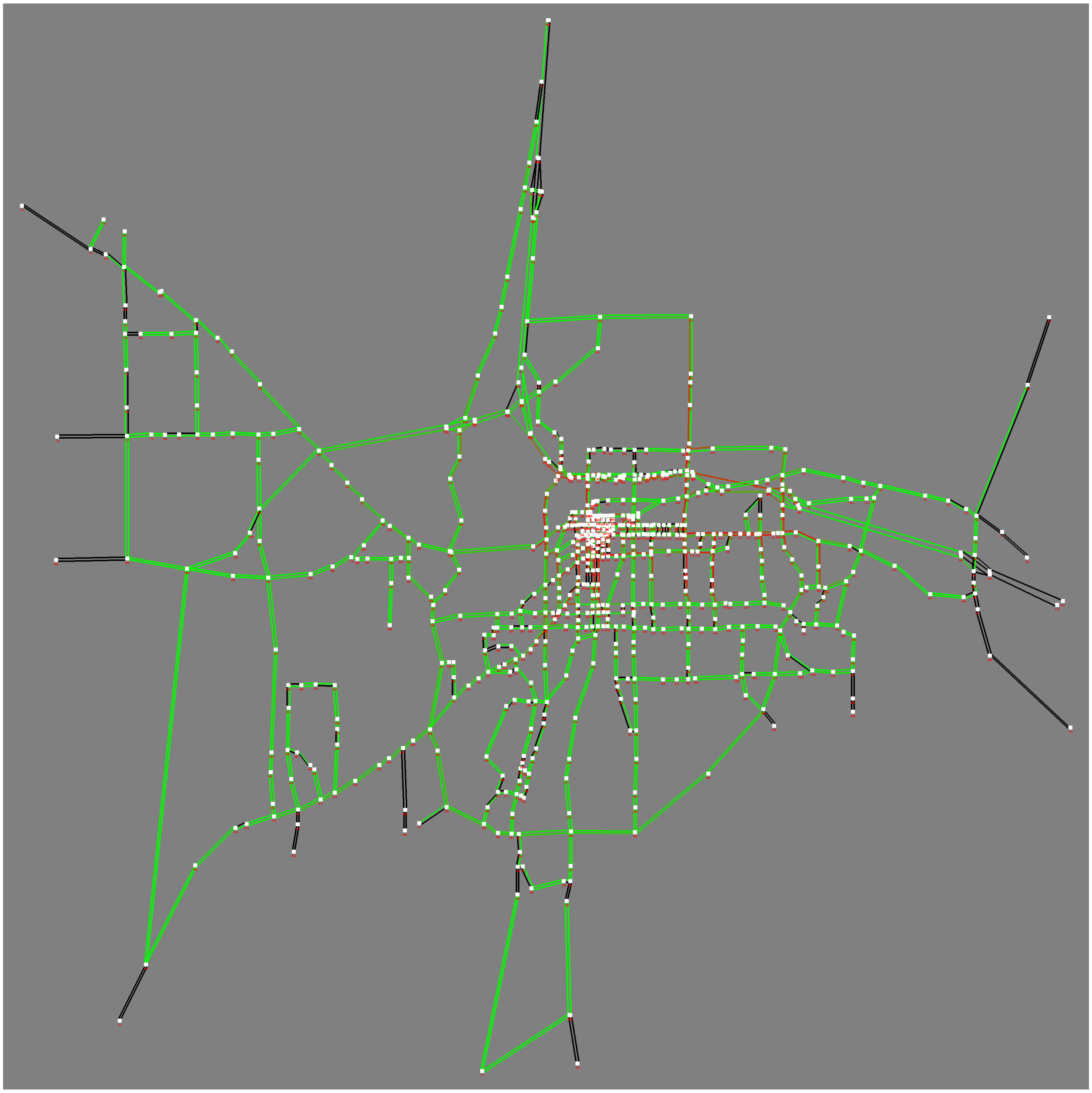


Figure 4.2: Map of the entire Casper network.

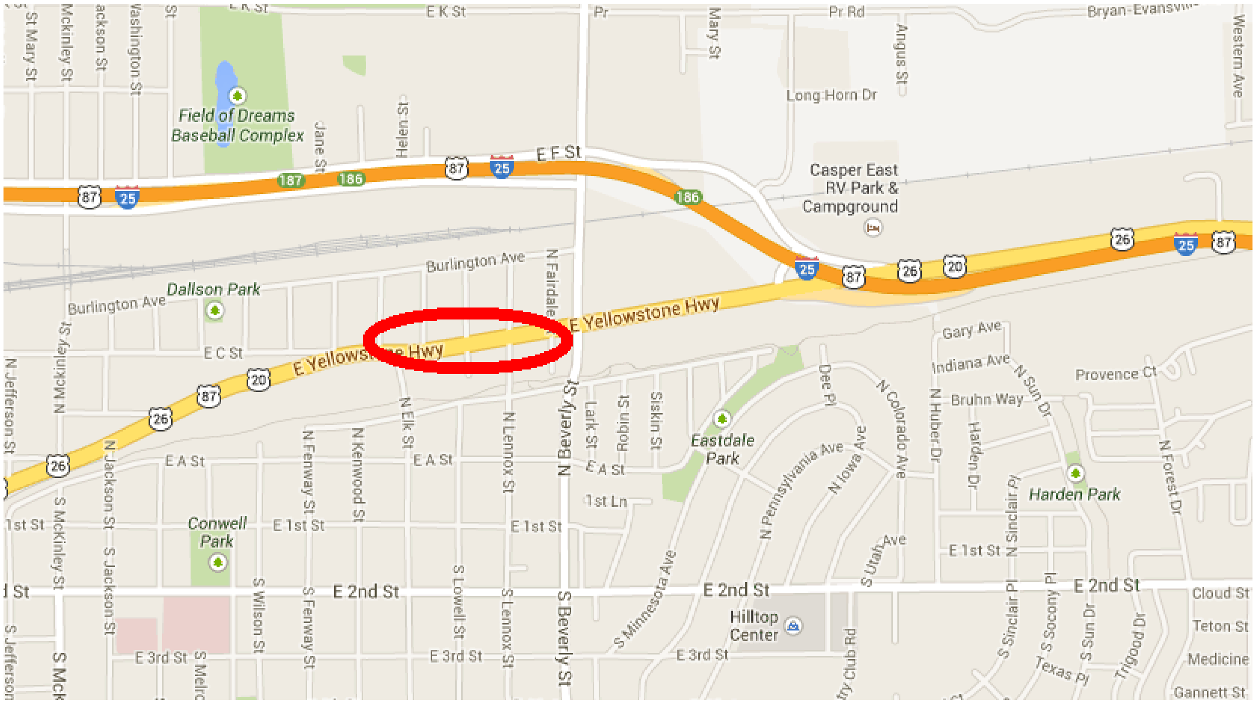


Figure 4.3: Link closed between Beverly St. & C St. for work zone.

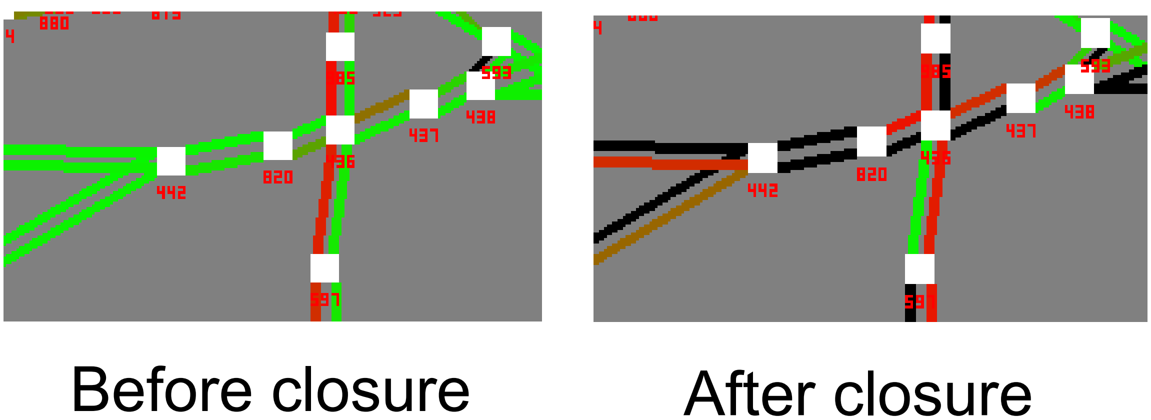


Figure 4.4: Link congestion between Beverly St. & C St. before and after work zone.

Table 4.1: Turning movement flows before and after closure of Yellowstone Hwy.

Movement	Flow before closure	Flow after closure
`820 -> 436 -> 985`	27	0
`820 -> 436 -> 597`	80	0
`820 -> 436 -> 437`	105	0
`597 -> 436 -> 985`	165	34
`597 -> 436 -> 820`	85	5
`597 -> 436 -> 437`	160	39
`437 -> 436 -> 985`	4	1
`437 -> 436 -> 820`	48	1
`437 -> 436 -> 597`	70	24
`985 -> 436 -> 820`	34	5
`985 -> 436 -> 597`	120	45
`985 -> 436 -> 437`	17	14

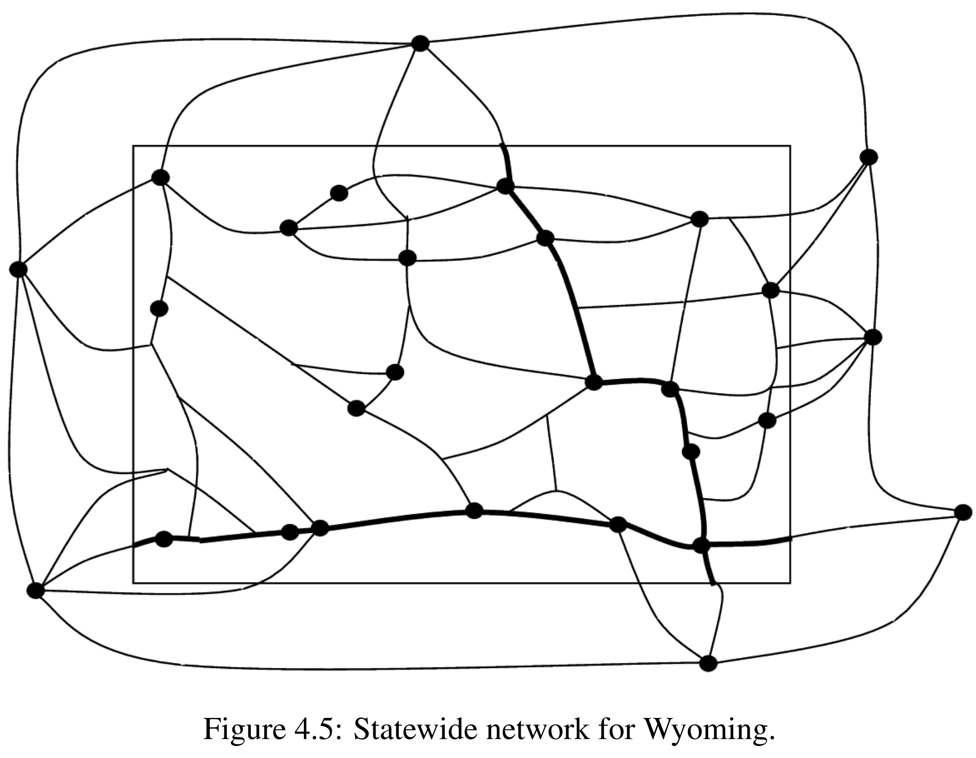


Figure 4.5: Statewide network for Wyoming.

literature. This segment is 111 miles long, and has a speed limit of 75 mph. At this speed, the link takes 1.48 hours to traverse. At $10/hr value of time, the $20 toll imposes the equivalent of two additional hours of time, or an equivalent of 3.48 hours. This, in turn, translates to a speed of 32 mph for the 111 mile distance. Therefore, in the network file, the speed limit for I-80 was reduced to 32 mph between Rock Springs and Rawlins to produce the same effect as a $20 toll.

Viewing the link summary file, we can quantify the amount of diversion: prior to imposing the toll, the average annual daily traffic (AADT) count was approximately 6650 at this location, while after the toll, the AADT count decreased to 2800. Thus, approximately 3800 vehicles diverted onto an alternate route, primarily out-of-state traffic. This new AADT count can also be used to forecast toll revenue.

Chapter 5

Summary & Conclusions

This project developed a mesoscopic simulation software capable of modeling both cities and statewide regions. The key concepts are (1) a traffic flow model, which is realistic enough to capture basic traffic dynamics, yet efficient and scalable to very large regions, and (2) a user behavior model where route choice and diversion are determined endogenously, based on simulated travel times. To this end, the cell transmission model was integrated with a user equilibrium principle to produce an efficient, dynamic traffic assignment simulator.

This simulation software was implemented in the C programming language, with a Microsoft Excel VBA frontend. The simulation itself proceeds in an iterative process, moving towards consistency between the traffic flow and user behavior models. The iteration proceeds through three primary submodules: the cell transmission model; the time-dependent A* algorithm to find the least-cost routes for each origin, destination, and departure time; and the method of successive averages to shift an appropriate number of vehicles from longer routes to shorter ones.

All input and output from the simulation takes the form of plain text files which are portable, human-readable, and easy to edit. A spreadsheet interface is also provided to facilitate data entry and analysis of results. Two additional modules were developed: a graphics module which produces network maps showing congestion patterns, and a warrant analysis module. The latter serves two purposes, and can overcome data limitations by developing a basic intersection control pattern throughout the network based on engineering warrants, or alternately serve as a postprocessing procedure to determine updated intersection control after simulation.

Case studies demonstrate this in both the Casper network, and a statewide network representing all of Wyoming. Appendices to the main document include tutorials, a programmer's guide to the source code, and the C and VBA code itself.

References

Beckmann, M. J., C. B. McGuire, and C. B. Winston (1956). *Studies in the Economics of Transportation*. New Haven, CT: Yale University Press.

Boyce, D., B. Ralevic-Dekic, and H. Bar-Gera (2004). Convergence of traffic assignments: how much is enough? *Journal of Transportation Engineering 130*(1), 49–55.

Boyles, S. D. and S. T. Waller (2010). Traffic network analysis and design. In J. J. Cochran (Ed.), *Wiley Encyclopedia of Operations Research*. Wiley.

Courant, R., K. Friedrichs, and H. Lewy (1928). Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen 100*(1), 32–74.

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

Federal Highway Administration (2009). *Manual on Uniform Traffic Control Devices*. Washington, DC: US Department of Transportation.

Godunov, S. K. (1959). A difference scheme for numerical solution of discontinuous solution of hydrodynamic equations. *Matematicheskii Sbornik 47*, 271–306.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4 4*(2), 100–107.

Lighthill, M. and G. Whitham (1955). On kinematic waves. II. A theory of traffic flow on long crowded roads. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 317–345.

Patriksson, M. (1994). *The Traffic Assignment Problem — Models and Methods*. Utrecht, Netherlands: VSP.

Pigou, A. C. (1920). *The Economics of Welfare*. London: Macmillan and Co.

Richards, P. (1956). Shock waves on the highway. *Operations Research 4*(1), 42–51.

Saha, P., R. Liu, C. Melson, and S. D. Boyles (2013). Network model for rural roadway tolling with pavement deterioration and repair. In review, *Computer-Aided Civil and Infrastructure Engineering*.

Transportation Research Board (2010). *Highway Capacity Manual*. Washington, DC: National Research Council.

Wardrop, J. (1952). Some theoretical aspects of road traffic research. *Proceedings of the Institute of Civil Engineers, Part II*, 325–378.

Webster, F. V. (1958). Traffic signal settings. Technical report, Her Majesty's Stationery Office, London. Road Research Technical Paper No. 30.

Wyoming Department of Transportation (2008). Interstate 80 tolling feasibility study. Prepared by Parsons Brinkerhoff in association with PB Consult.

Yperman, I. (2007). *The Link Transmission Model for Dynamic Newtork Loading*. Ph. D. thesis, Katholieke Universiteit Leuven, Belgium.

Appendix A

Tutorials

This appendix contains in-depth tutorials for the three case studies in Chapter 4. These include:

1. A small toy network which is made from scratch, showing how to use the interface to create and edit data files. This example also shows how to use the warrant analysis and graphics modules as well.
2. A network representing the city of Casper. This example shows how the simulator can be used to predict the influence of a work zone closure on traffic diversion, and on re-optimizing signal control.
3. A network representing the state of Wyoming. This example shows how the simulator can reflect large-scale diversion which can occur as a result of a toll on I-80. This reflects a less conventional use of the traffic simulator and shows the flexibility of the mesoscopic approach.

Specific instructions for the tutorial are printed in **boldface**, interspersed with additional description of how the program is working.

A.1 Toy network

We want to create the network shown schematically in Figure A.1. Note that the node numberings have been determined in advance, and are chosen so that the origin and destination centroids have the lowest numbers (1–7), while the regular intersections 8–11 have the higher ID numbers. Intersections 1–7 are centroids. Every link is 1 mile long and has a capacity of 5000 vph and jam density 200 veh/mi; the thick shaded links have a speed limit of 30 mph and all other links have a speed limit of 60 mph. According to a travel demand model which has been conducted earlier, 3000 vehicles depart from zone 1 to zone 3; 500 vehicles from zone to 3 to zone 6; 100 from zone

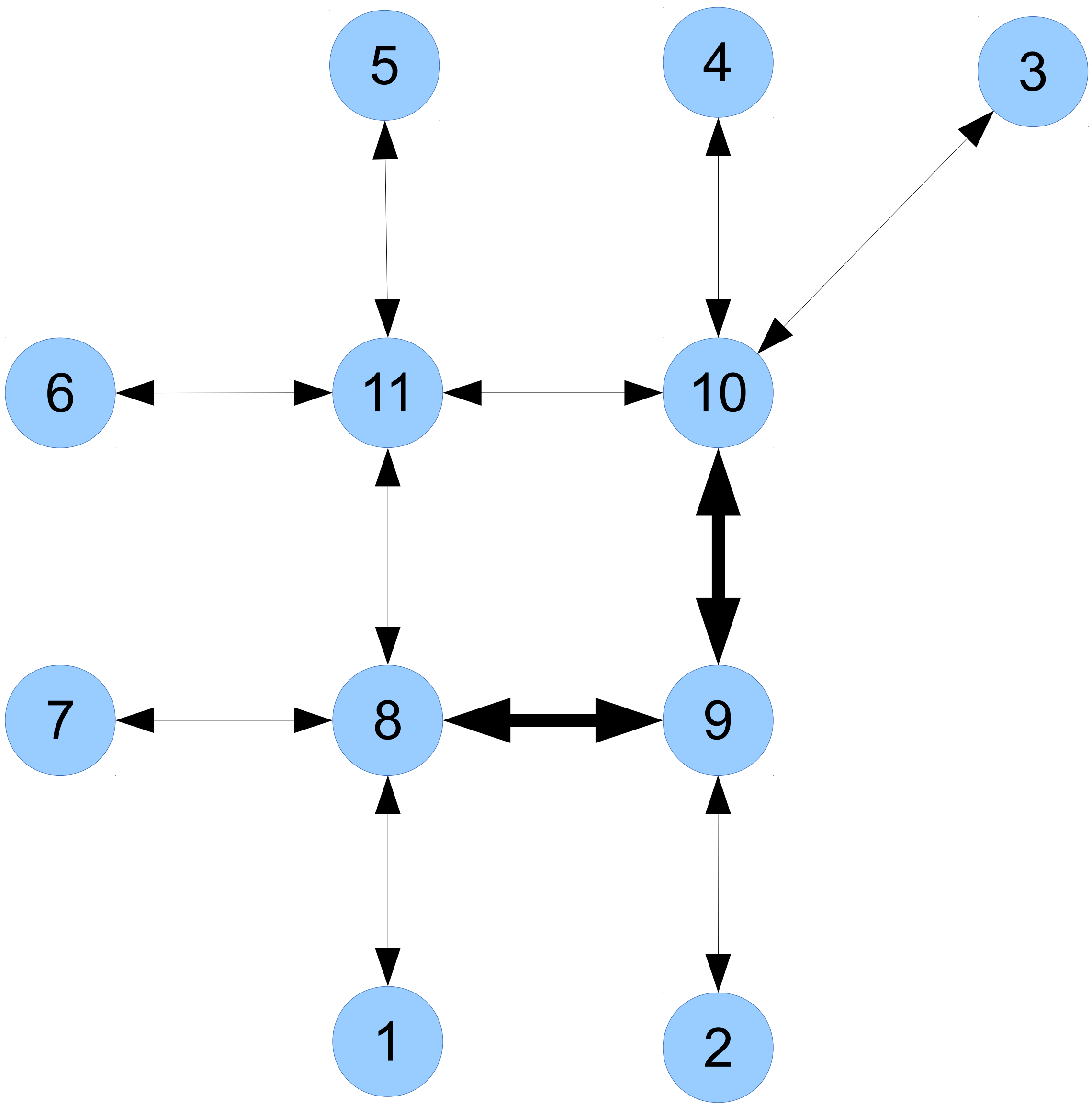


Figure A.1: Toy network for tutorial.

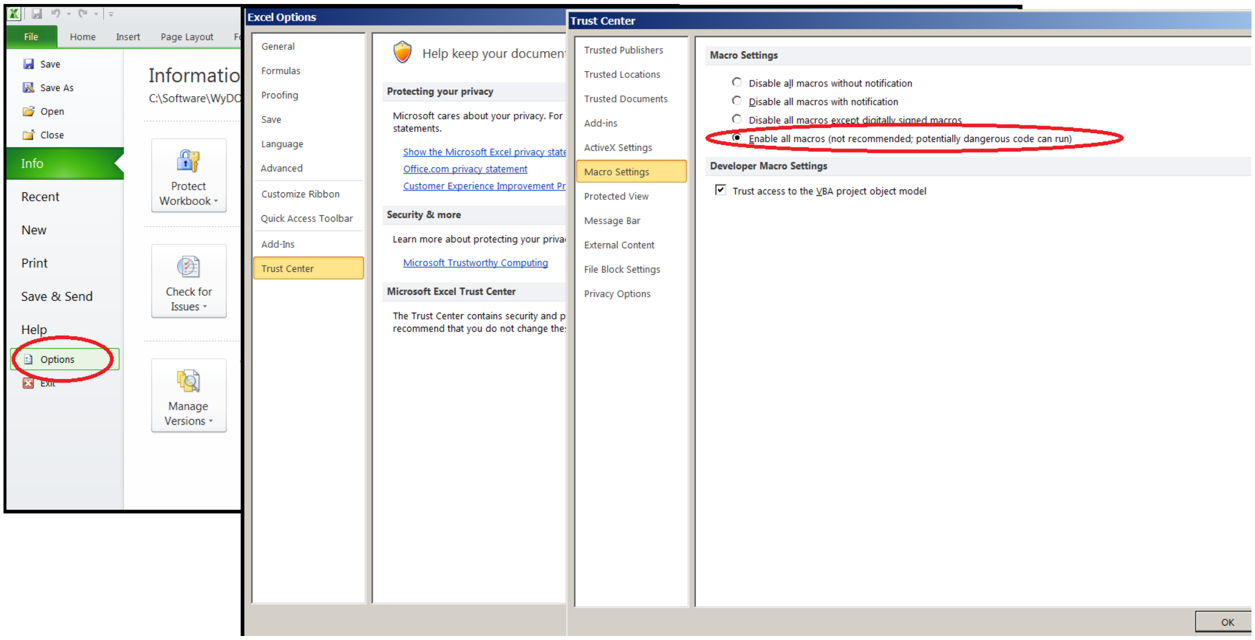


Figure A.2: Changing macro settings in Excel.

4 to zone 2; 500 from 6 to 3; and 200 from 7 to 2. Assume that the node controls at intersections 8, 9, 10, and 11 is currently unknown.

Open the spreadsheet `interface.xlsm` with Microsoft Excel. This interface relies extensively on Excel VBA macros, which must be enabled. If you receive a warning about macros being disabled because of your security settings, follow these instructions: click on the File tab at the upper left of the ribbon, and click "Excel Options." Click on "Trust Center" at the bottom of the left panel, then choose "Macro Settings." Select "Enable all macros." See Figure A.2 (You should switch your macro settings back to their initial settings after you have finished using the interface, by performing the same steps and choosing a different security level.)

You should see the dashboard shown in Figure A.3. **Click on "Go to project summary page"** underneath "Input Data." This will take you to the Parameters sheet, where you can enter in the basic project information the interface needs to construct the remainder of the sheets. (Figure A.4). **Enter in the following information in the cells marked "Network data" and "Simulation parameters":**

Number of nodes: 11

Number of links: 18

Number of zones: 7

Time horizon (hr): 2

Last vehicle loaded (hr): 1

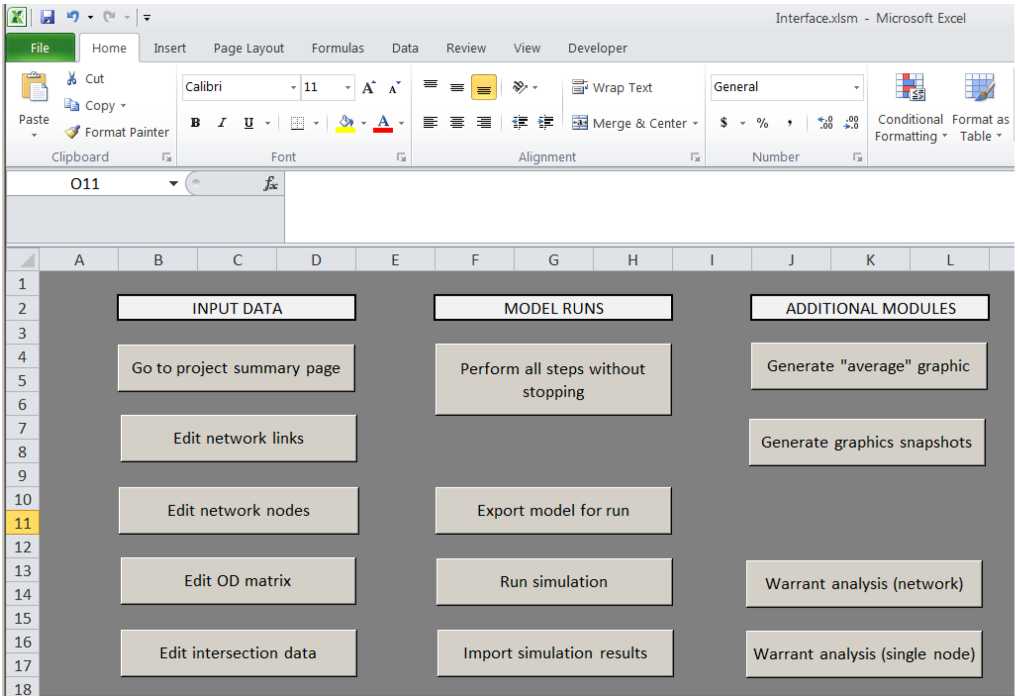


Figure A.3: Dashboard screenshot.

Tick length (s): 10

Maximum run time (min): 10

AEC threshold (min): 5

Warm-up period (hr): 0.25

Cool-down period (hr): 0.25

For more details on what each of these parameters does, consult the relevant sections in Chapter 3. After entering this information, the spreadsheet should appear as in Figure A.5.

Now, **press the "Prepare for manual input" button at the bottom of the sheet.** This calls a series of macros which format the link data, node data, OD matrix, and intersection data spreadsheets based on the parameters specified. Pressing this button will erase anything which is currently in these sheets, so the interface will prompt for confirmation. Since these sheets are blank at this time, **click "Yes" when prompted.**

After creating these sheets, the interface will load the Link Data sheet. Enter the link parameters shown in Figure A.6. **Click on the "Proceed to node data" button**, which will advance you to the Node Data sheet. Enter the node parameters shown in Figure A.7, then **click on the "Proceed to OD Matrix" button**. You should see a blank matrix with entries for each of the possible origin-destination pairs in the network. The rows index the origin of travel, and the columns index the destination. As was stated at the start of the section, assume that 3000 vehicles depart from

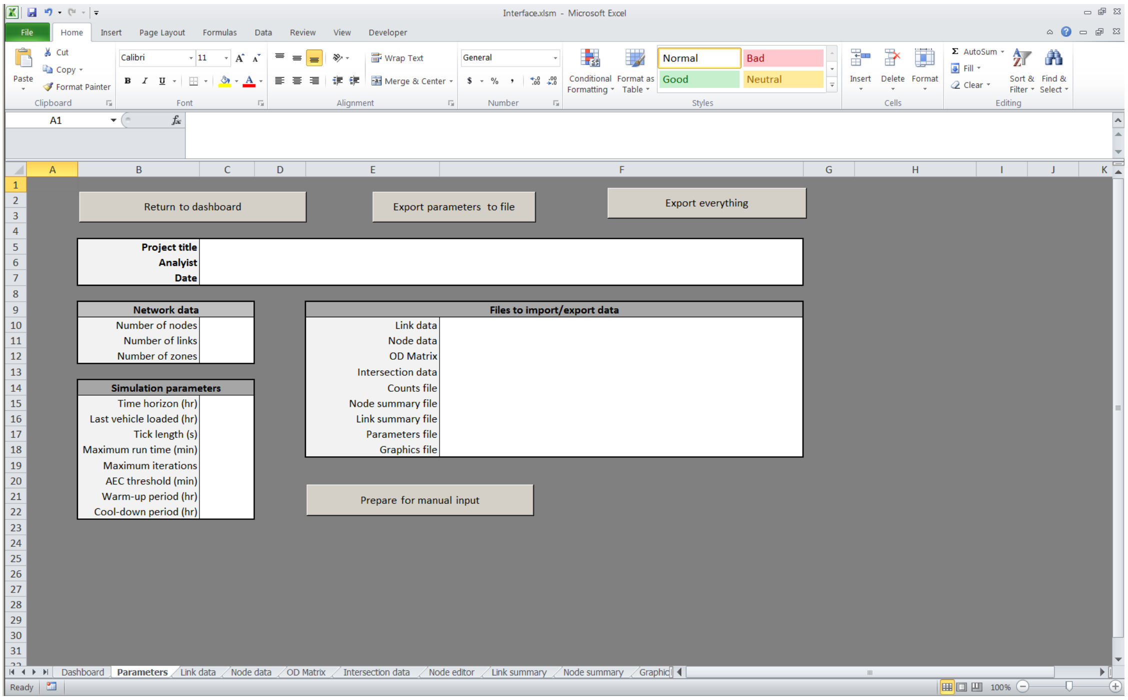


Figure A.4: Toy network parameters, blank.

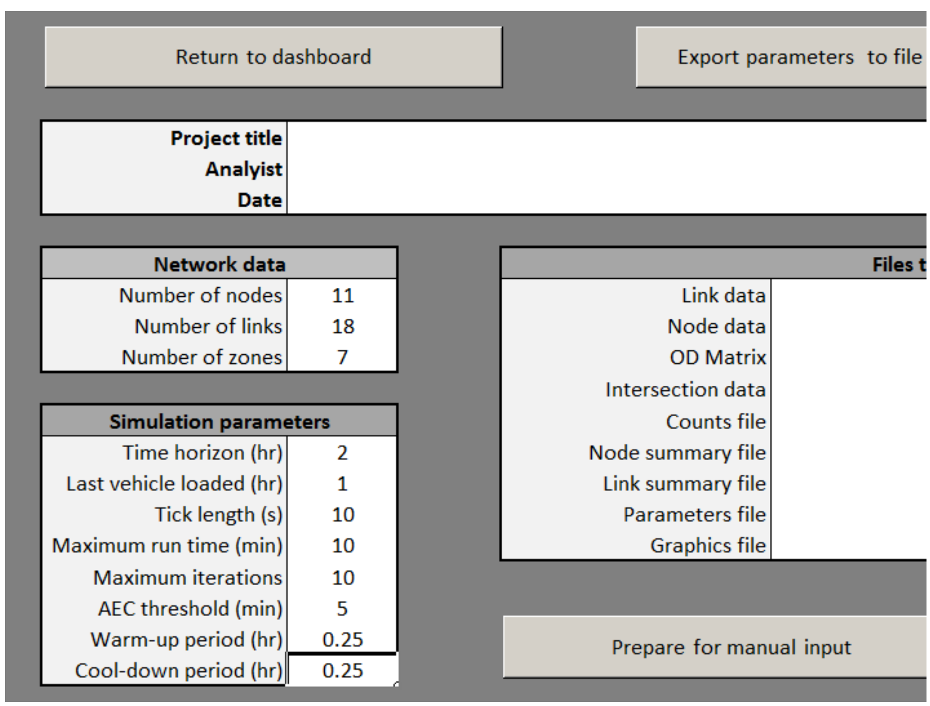


Figure A.5: Toy network parameters, completed.

Return to dashboard | Proceed to node data | Export to file

ID	From	To	Capacity (vph)	Length (ft)	Speed limit (mph)	Jam density (veh/mi)
1	1	8	5000	5280	60	200
2	3	10	5000	5280	60	200
3	4	10	5000	5280	60	200
4	6	11	5000	5280	60	200
5	7	8	5000	5280	60	200
6	8	9	5000	5280	30	200
7	8	11	5000	5280	60	200
8	9	2	5000	5280	60	200
9	9	8	5000	5280	30	200
10	9	10	5000	5280	30	200
11	10	3	5000	5280	60	200
12	10	4	5000	5280	60	200
13	10	9	5000	5280	30	200
14	10	11	5000	5280	60	200
15	11	5	5000	5280	60	200
16	11	6	5000	5280	60	200
17	11	8	5000	5280	60	200
18	11	10	5000	5280	60	200

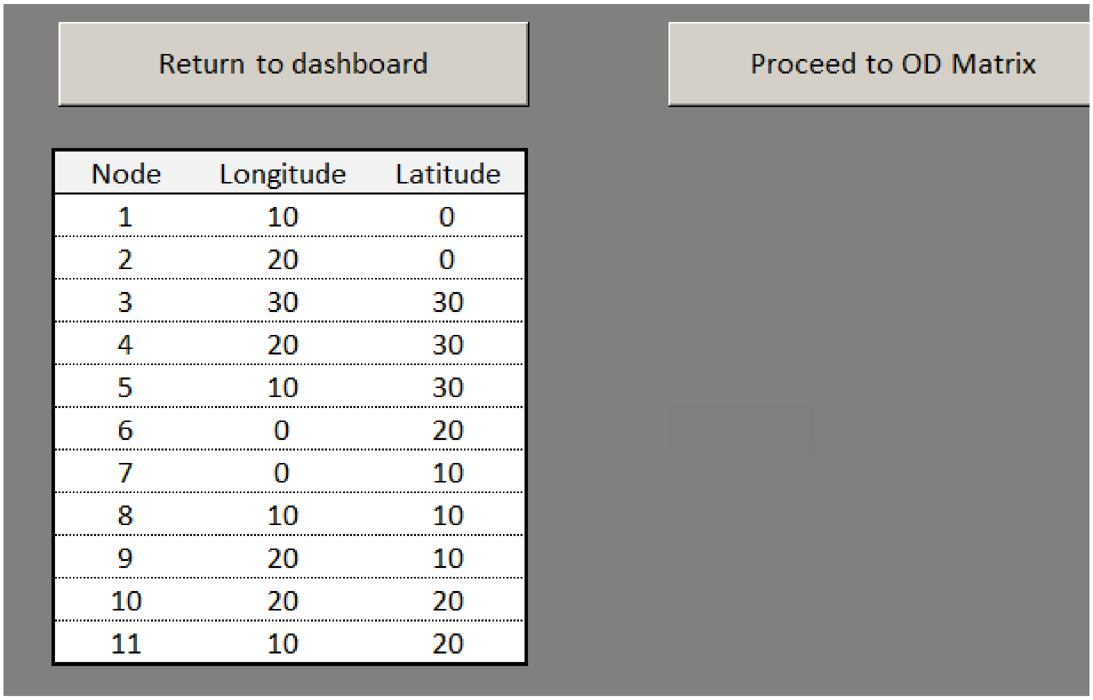
Figure A.6: Toy network link data, completed.

zone 1 to zone 3; 500 vehicles from zone to 3 to zone 6; 100 from zone 4 to zone 2; 500 from 6 to 3; and 200 from 7 to 2. **Enter these values into the sheet**, checking that your final result looks like Figure A.8, making sure that the rows and columns haven't been mixed up.

Click on "Proceed to intersection data," the final sheet for inputting data. The dropdown boxes in Column C indicate the different intersection control types for each intersection in the network. Nodes 1-7 are zones, and must have Centroid type. Since we don't know the control types for the other intersections, give them Unknown type. Before running a simulation, these will have to be filled in using the warrant analysis module. To do this, **click on "Return to dashboard."** Once on the dashboard, **click on "Warrant analysis (network)."**

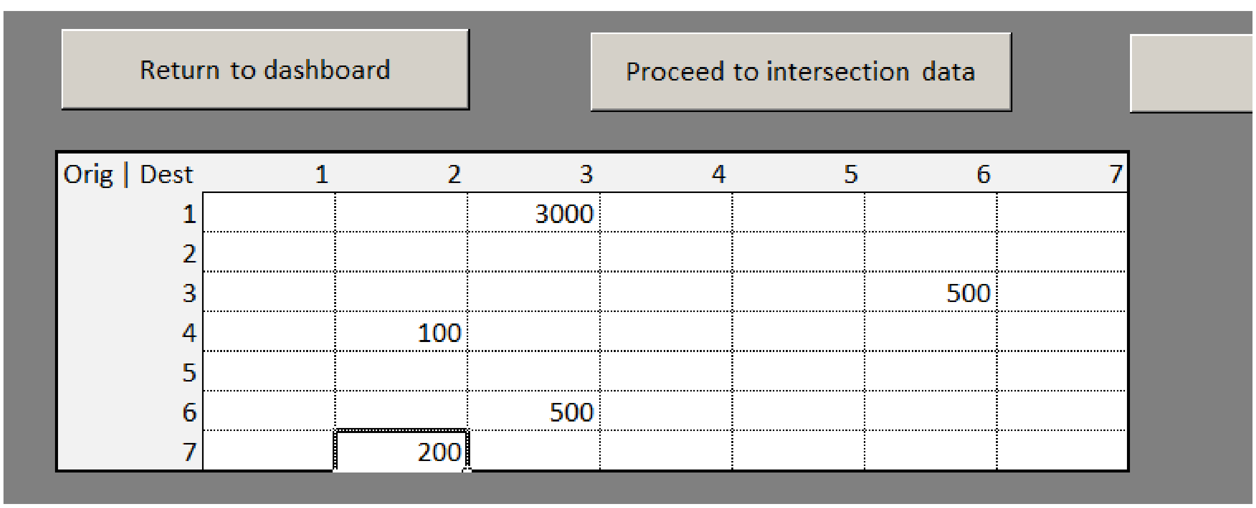
The first step in the warrant analysis is to export all of the information entered in the spreadsheet into the text files which the simulator will read. (You can also edit these text files directly, if desired.) Five dialog boxes will appear in sequence, asking you to enter the locations to save (a) the link data file, (b) the node data file, (c) the OD matrix, (d) the intersection control file, and (e) the parameters file. Enter the following information for each of these: (a) `toy_network.txt`, (b) `toy_nodes.txt`, (c) `toy_demand.txt`, (d) `toy_control.txt`, (e) `toy_parameters.txt`. The warrant analysis program will then run external to Excel, with control returning to the spreadsheet when it's finished.

At this point, if you explore the different spreadsheets in the file, you will notice a few changes. First, on the Parameters sheet, you will find the names of the filenames you typed (cf. Figure A.10, keeping in mind the exact path may differ depending on the folder you saved the files into). More importantly, on the Intersection Data sheet, you will see that nodes 8, 9, 10, and 11 no longer have



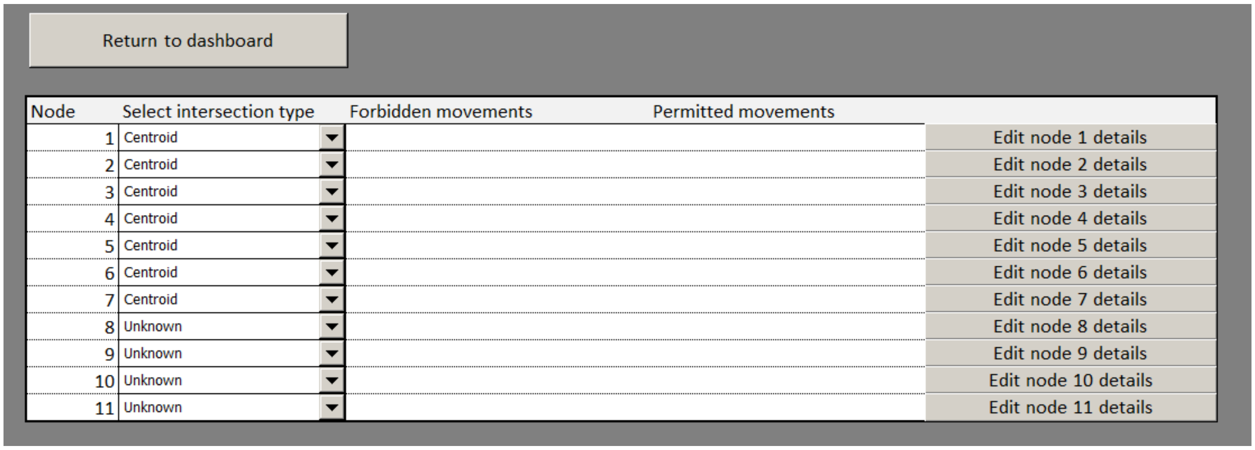
Node	Longitude	Latitude
1	10	0
2	20	0
3	30	30
4	20	30
5	10	30
6	0	20
7	0	10
8	10	10
9	20	10
10	20	20
11	10	20

Figure A.7: Toy network node data, completed.



Orig | Dest	1	2	3	4	5	6	7
1			3000				
2							
3						500	
4		100					
5							
6			500				
7		200					

Figure A.8: Toy OD matrix data, completed.


Return to dashboard

Node	Select intersection type	Forbidden movements	Permitted movements	
1	Centroid			Edit node 1 details
2	Centroid			Edit node 2 details
3	Centroid			Edit node 3 details
4	Centroid			Edit node 4 details
5	Centroid			Edit node 5 details
6	Centroid			Edit node 6 details
7	Centroid			Edit node 7 details
8	Unknown			Edit node 8 details
9	Unknown			Edit node 9 details
10	Unknown			Edit node 10 details
11	Unknown			Edit node 11 details

Figure A.9: Toy intersection data, completed.

Unknown type. Nodes 8 and 10 are now controlled by two-way stop, Node 9 by a four-way stop, and Node 11 by a signal, based on the warrant analysis which has been conducted. (Figure A.11).

To see more details on the signal timing chosen for node 11, **click on the "Edit node 11 details"** button. You should see the Node Editor sheet for node 11. Turning movement saturation flows are based on those of the upstream link, the cycle length was determined by Webster's Method, and the green times apportioned proportional to the degree of saturation for each approach. Now, let's edit this slightly: the green time for the approaches from link (10,11) seem very small. **Change the cycle length to 30, and the effective green times for movements `10 -> 11 -> 5`, `10 -> 11 -> 6`, and `10 -> 11 -> 8` to 13**, as shown in Figure A.13.

Now, we can run a full simulation to generate more detailed information. First, **click on "Save information and return to intersection data."** This step is very important; if you do not click this, your changes will not be saved. **Click on "Return to dashboard," and "Perform all steps without stopping."** This performs the following steps, in order:

1. Export the same 4 data files requested during the warrant analysis. (These files must be re-exported, because the warrant analysis changed the file configuration.) **use the same names as during the warrant analysis**.
2. Prompts you for the names of the three output files, the counts file, the node summary file, and the link summary file. **When prompted, enter `toy_counts.txt`, `toy_nodesummary.txt`, and `toy_linksummary.txt`.**
3. Export the parameters file; **use the same name as during the warrant analysis**.
4. Calls the mesoscopic simulation module, which run in a separate process.
5. Imports the link and node summary files.
6. Moves the user to the Link Summary spreadsheet.

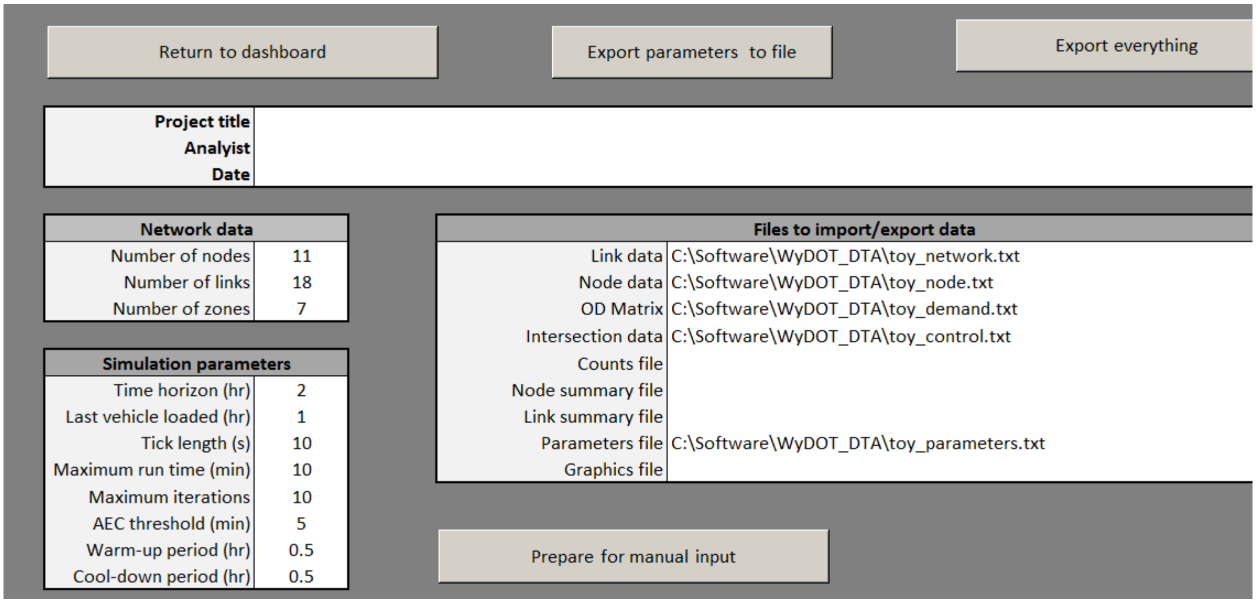


Figure A.10: File names have now appeared in the Parameters sheet.

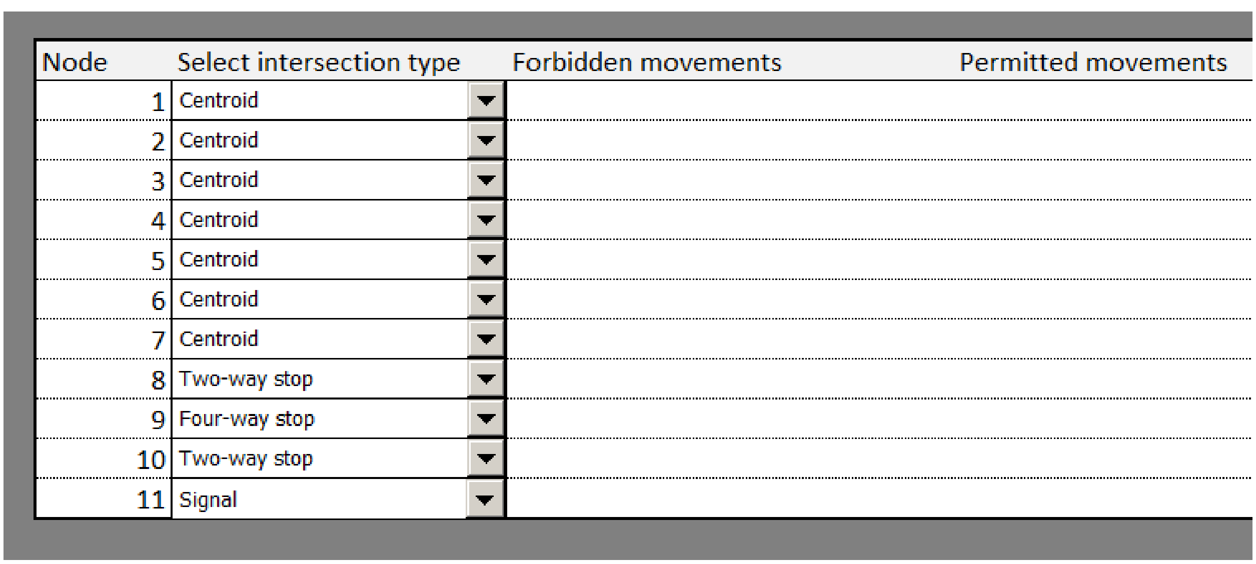
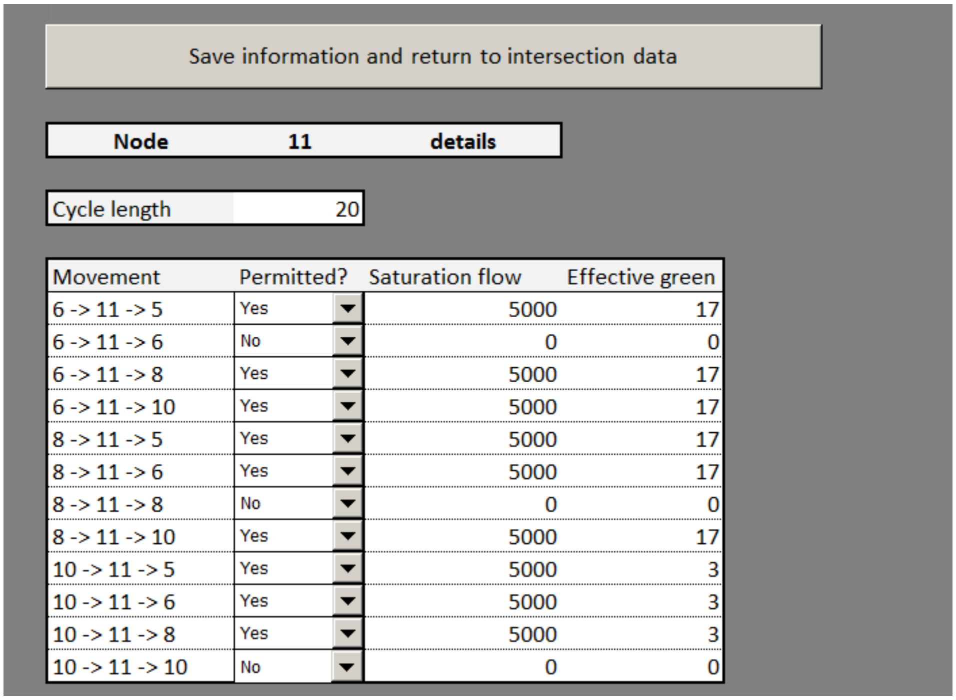


Figure A.11: Intersection types chosen by the warrant analysis.



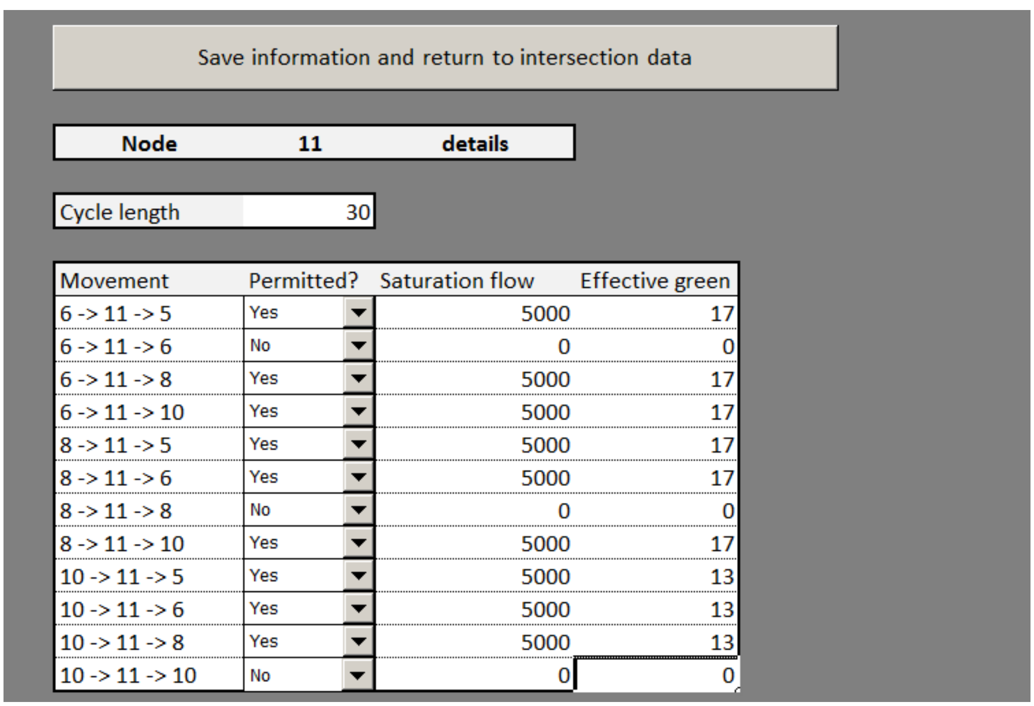
Save information and return to intersection data

Node	11	details

Cycle length	20

Movement	Permitted?	Saturation flow	Effective green
6 -> 11 -> 5	Yes	5000	17
6 -> 11 -> 6	No	0	0
6 -> 11 -> 8	Yes	5000	17
6 -> 11 -> 10	Yes	5000	17
8 -> 11 -> 5	Yes	5000	17
8 -> 11 -> 6	Yes	5000	17
8 -> 11 -> 8	No	0	0
8 -> 11 -> 10	Yes	5000	17
10 -> 11 -> 5	Yes	5000	3
10 -> 11 -> 6	Yes	5000	3
10 -> 11 -> 8	Yes	5000	3
10 -> 11 -> 10	No	0	0

Figure A.12: Node details for node 11 from warrant analysis.



Save information and return to intersection data

Node	11	details

Cycle length	30

Movement	Permitted?	Saturation flow	Effective green
6 -> 11 -> 5	Yes	5000	17
6 -> 11 -> 6	No	0	0
6 -> 11 -> 8	Yes	5000	17
6 -> 11 -> 10	Yes	5000	17
8 -> 11 -> 5	Yes	5000	17
8 -> 11 -> 6	Yes	5000	17
8 -> 11 -> 8	No	0	0
8 -> 11 -> 10	Yes	5000	17
10 -> 11 -> 5	Yes	5000	13
10 -> 11 -> 6	Yes	5000	13
10 -> 11 -> 8	Yes	5000	13
10 -> 11 -> 10	No	0	0

Figure A.13: Modified details for node 11.

ID	From	To	Travel time (s)	Delay (s)	Density (veh/mi)	Volume (vph)	PHF
1	1	8	60	0	25	1533	0.75
2	3	10	60	0	4	255	0.73
3	4	10	60	0	1	44	0.53
4	6	11	60	0	4	251	0.72
5	7	8	60	0	2	104	0.7
6	8	9	120	0	4	108	0.72
7	8	11	60	0	26	1571	0.77
8	9	2	60	0	3	161	0.72
9	9	8	120	0	0	0	---
10	9	10	120	0	0	0	---
11	10	3	60	0	32	1910	0.8
12	10	4	60	0	0	0	---
13	10	9	120	0	2	49	0.59
14	10	11	60	0	4	261	0.74
15	11	5	60	0	0	0	---
16	11	6	60	0	4	267	0.76
17	11	8	60	0	0	0	---
18	11	10	61	1	31	1862	0.78

Figure A.14: Link summary information for toy network.

The Link Summary spreadsheet should appear as in Figure A.14. Note that the average density, volumes, and peak hour factors have been reported for each link in the network. Similar information appears in the Node Summary spreadsheet for each turning movement, sorted by the associated intersection. Finally, let's generate a graphics file using the data already obtained. **Click on "Generate 'average' graphic," and enter `toy_final.png` when prompted for the image name.** You should see the image in Figure A.15. This concludes the tutorial for the toy network.

A.2 Casper

The Casper tutorial shows how the simulator can be used on the city scale to model diversion for a work zone closure. This network is considerably larger than the toy network in the previous section: it contains 304 centroids, 1014 intersections, and 2760 roadway links. However, the `casper.xlsm` spreadsheet has the network data pre-loaded. The network and data were constructed from the TransCAD model used for long-range planning. The entire network is shown in Figure A.16. To simulate a work zone closure, the link representing Yellowstone Highway between Beverly St and C St is closed (represented in the simulator by deleting its link from the network file) (Figure A.17). Because this street is bidirectional, two links must be deleted — link 893 (connecting intersections 442 and 820) and link 2202 (connecting 820 and 442).

To examine the traffic conditions before the work zone is present, export all data, run the simulation, and import the simulation results as was done for the toy network. Zooming in to

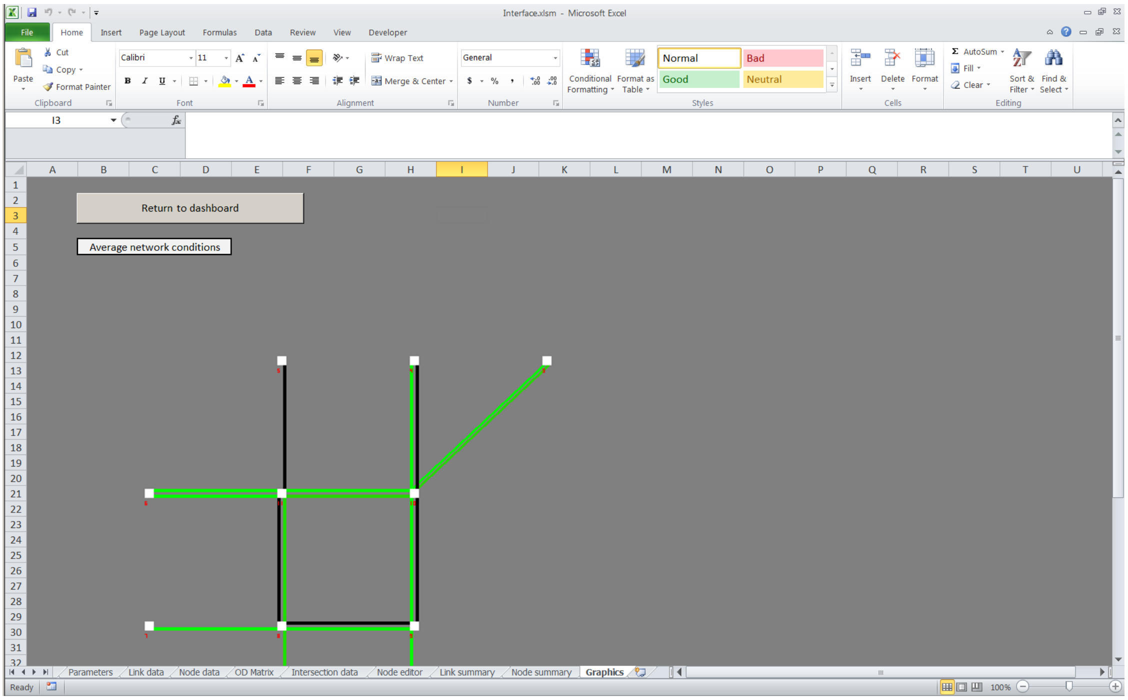


Figure A.15: Average graphics file for toy network.

node 436 (the intersection of Yellowstone Hwy & Beverly St), the flows for each turn movement can be recorded (and are shown in Table A.1).

To delete this link from the network, perform the following steps:

1. **On the Parameters sheet, decrease the number of links by 2.**
2. On the Link Data sheet, delete the rows for links 893 (connecting 442 and 820) and 2202 (connecting 820 and 442). To delete these, **select the entire row, and choose `Delete -> Delete Sheet Rows`** from the ribbon.
3. Export all data again, choosing different names to avoid overwriting the original results.
4. Run the simulation again.
5. Import the new results, and view the node summary file.

This obtains the updated flows shown in the right column of Table 4.1. Note the substantial diversion of flow away from this intersection after the closure. This reflects travelers choosing alternate routes to avoid the work zone and closure.

To find out how the traffic control should be revised to account for the change in flow, **click on "Warrant analysis (single node)"** on the dashboard. The single-node warrant analysis now looks at the revised flows, and makes the appropriate change on the Intersection Data worksheet. Clicking to this worksheet, one sees that the new recommended control at this intersection is a

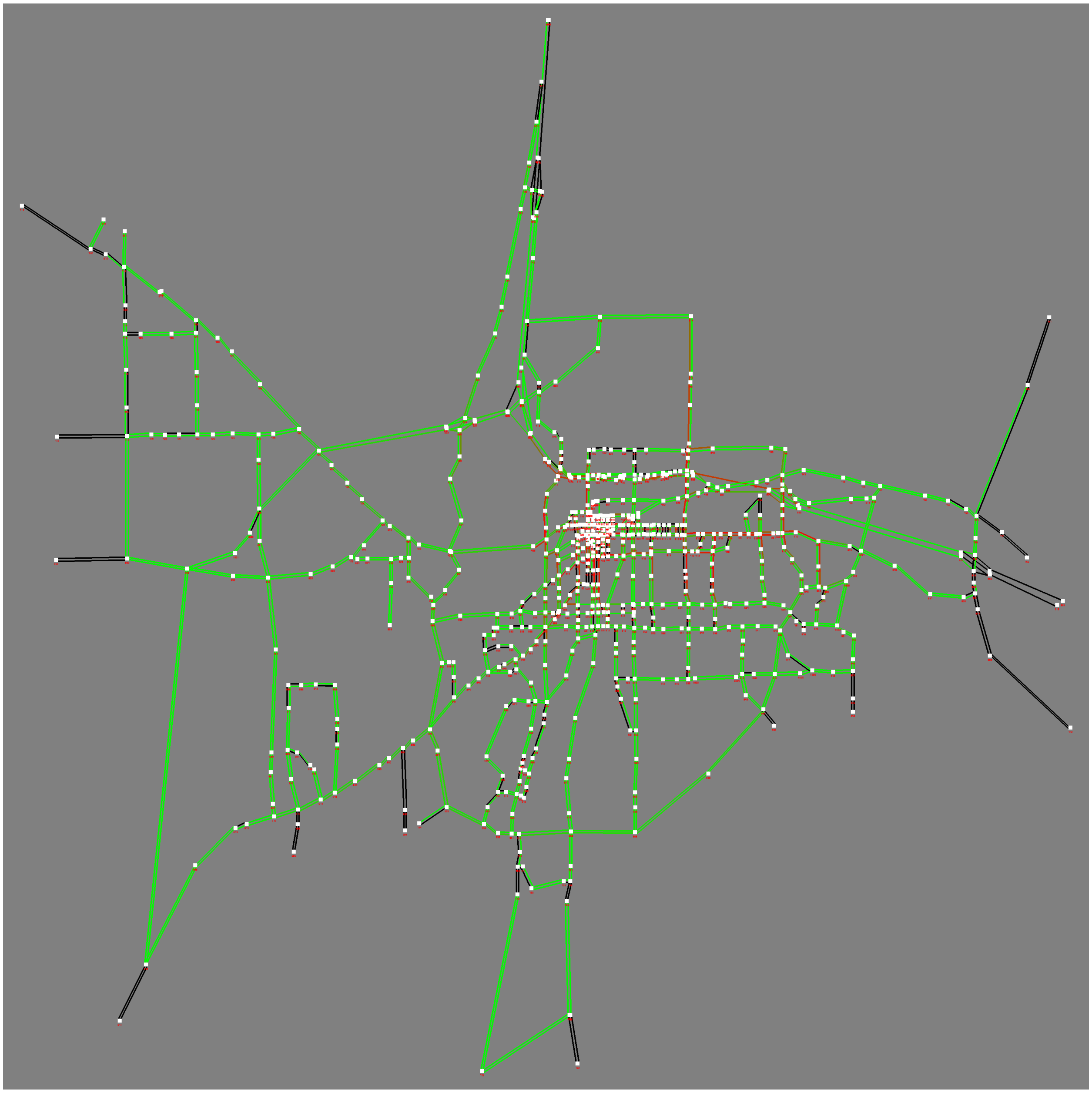


Figure A.16: Map of the entire Casper network.

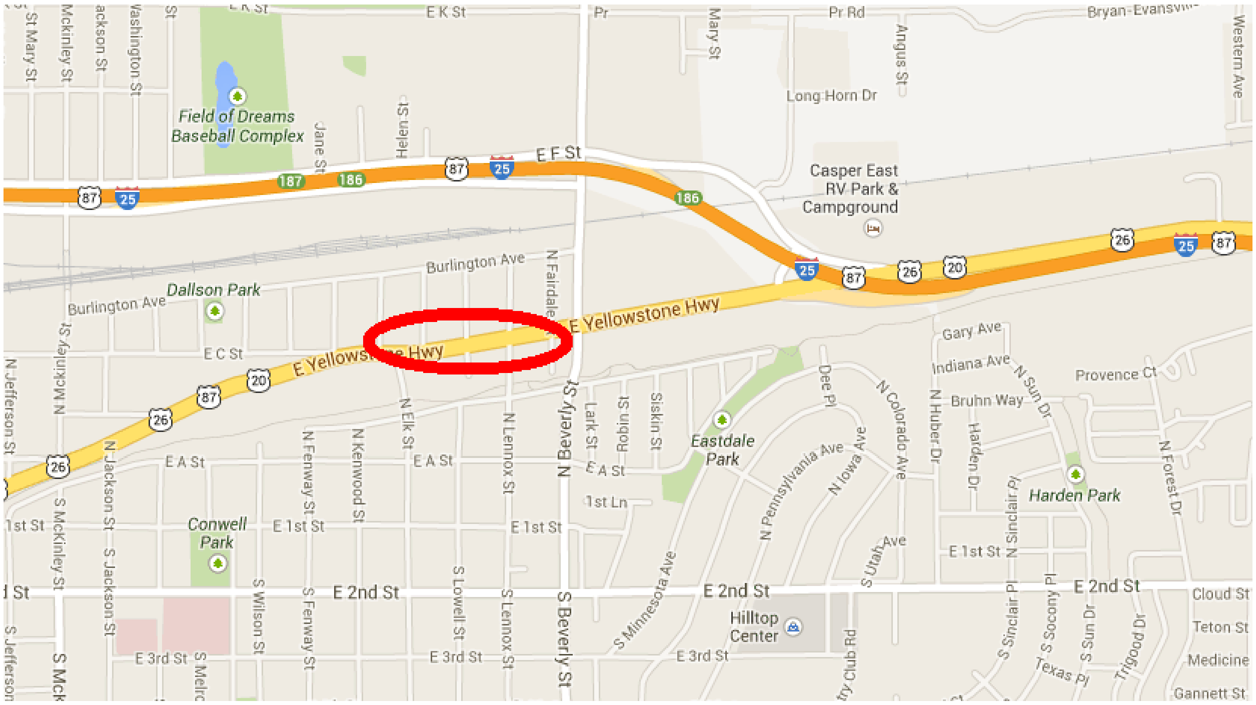


Figure A.17: Link closed for work zone.

Table A.1: Turning movement flows before and after closure of Yellowstone Hwy.

Movement	Flow before closure	Flow after closure
820 -> 436 -> 985	27	0
820 -> 436 -> 597	80	0
820 -> 436 -> 437	105	0
597 -> 436 -> 985	165	34
597 -> 436 -> 820	85	5
597 -> 436 -> 437	160	39
437 -> 436 -> 985	4	1
437 -> 436 -> 820	48	1
437 -> 436 -> 597	70	24
985 -> 436 -> 820	34	5
985 -> 436 -> 597	120	45
985 -> 436 -> 437	17	14

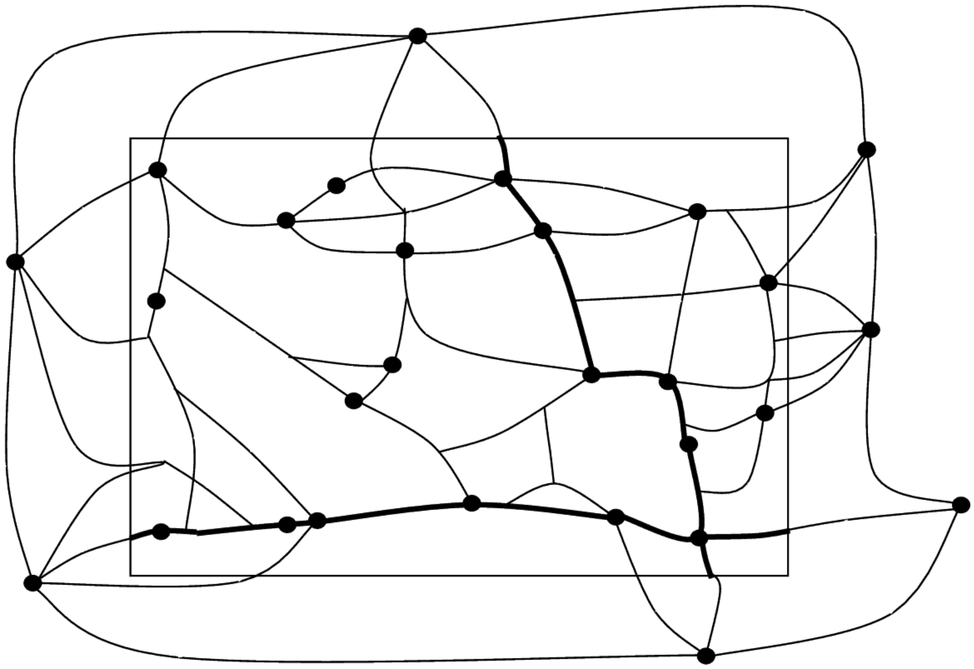


Figure A.18: Statewide network for Wyoming.

Four-Way Stop: due to diversion effects, this intersection sees much less traffic than before, and a four-way stop is sufficient to handle the traffic volumes. (This analysis only takes into account the traffic volume warrants. Particularly around work zones, safety factors may dictate another form of control.)

A.3 Wyoming

The last tutorial involves the Wyoming network. Like the Casper network, a "preloaded" version can be found in the spreadsheet `wyoming.xlsx`. The Wyoming network was constructed manually, including every interstate and federal highway in the State, along with several "external" intersections and links which represent diversion opportunities outside the state (Figure A.18). More details on the construction process can be found in Saha et al. (2013). The specific policy studied here was levying a toll on I-80 between Rock Springs and Rawlins, as suggested by Wyoming Department of Transportation (2008).

While the simulator does not directly model tolls on links, a toll can be effectively modeled by adjusting the free-flow speed of a link, as follows: to simulate a \$20 toll, we increase the free-flow travel time on this link by an equivalent amount. This equivalency depends on travelers' value of time; this demonstration assumes an average \$10/hr value of time, a standard figure in the tolling literature. This segment is 111 miles long, and has a speed limit of 75 mph. At this speed, the link takes 1.48 hours to traverse. At \$10/hr value of time, the \$20 toll imposes the equivalent of two additional hours of time, or an equivalent of 3.48 hours. This, in turn, translates to a speed of 32 mph for the 111 mile distance. Therefore, in the network file, the speed limit for I-80 was reduced to 32 mph between Rock Springs and Rawlins to produce the same effect as a $20 toll. **This change is made on the Link Data sheet, to link 7 (connecting intersections 38 and 45) and link 117 (connecting intersections 45 and 38)**.

Viewing the link summary file, we can quantify the amount of diversion: prior to imposing the toll, the average annual daily traffic (AADT) count was approximately 6650 at this location, while after the toll, the AADT count decreased to 2800. Thus, approximately 3800 vehicles diverted onto an alternate route, primarily out-of-state traffic. This new AADT count can also be used to forecast toll revenue.

Appendix B

Source Code Guide

This appendix describes how the source code for the simulator is organized. As the complete source code has been provided to WYDOT, in the future WYDOT may wish to extend or modify the simulator in a variety of ways, and this appendix helps orient programmers to the structure of the code. Efforts have been made to make the code modular and easy to understand, using principles of structured programming. This appendix does not provide details on the methodology or general flow of the algorithm, which are provided in Chapters 2 and 3, respectively.

The simulator was written in C, and is compatible with the ANSI C89 standard. (Virtually any modern C compiler should be able to compile to this standard.) The simulator code is spread across ten source files and ten header files. These files are generally organized in a hierarchical manner, as shown in Figure B.1, in the sense that source files call on functions or use data structures of equal or lower hierarchy, but not functions or data structures from higher files in the hierarchy. There are a few exceptions to this rule, where strict obedience would lead to convoluted or confusing solutions, but in the large majority of instances the code respects this organization. This facilitates compilation order and makefile creation.

Each file is briefly described as follows, starting from the bottom of the hierarchy and working up:

1. `utils.c` and `utils.h`: These files define basic macros and functions used throughout the code. These include basic mathematical functions not included in the standard libraries (e.g., maximization, minimization, and rounding), definition of a boolean data type (if not being compiled in C++ mode), defining the verbosity levels for status messages, and providing logging functions to display messages and warnings (based on the verbosity level), and fatal errors (regardless of the verbosity level).

2. `sampling.c` and `sampling.h`: These files contain the code used to generate random numbers from different distributions. The most important for the simulator are `roundStochastic`, which employs stochastic rounding to convert floating-point numbers to integers (e.g., since vehicles are modeled in a discrete fashion, fractional flows cannot be created) and the function `roundStochasticMatrix`, which stochastically rounds a

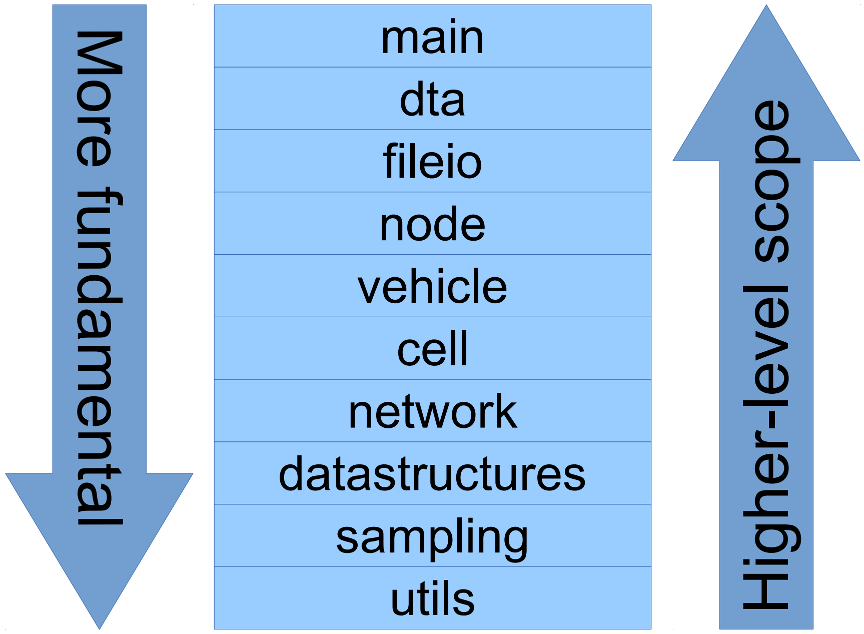


Figure B.1: Hierarchy of source files.

floating-point matrix to integers while preserving row and column sums to the extent possible.

3. `datastructures.c` and `datastructures.h`: These files define basic data structure templates for singly and doubly linked lists, queues, and binary heaps. These are general-purpose data structures, and more specific implementations are described in files higher in the hierarchy. Additionally, custom memory allocation and deallocation routines are provided, with optional memory-leak checking. These routines are discussed in more detail below.

4. `network.c` and `network.h`: These files establish the data structures for general network modeling, along with standard network algorithms. **This file contains the core module `TDAStar`, which implements the time-dependent A* algorithm.** The data structures in this file are extremely important, and are discussed in more detail below.

5. `cell.c` and `cell.h`: These files contain the data structures and routines related to the vehicles themselves, and their propagation from one cell to another, from roadway links to turning movements, and vice versa. Gap calculations for the convergence criteria are also found in these files.

6. `vehicle.c` and `vehicle.h`: These files contain the data structures and routines related to the vehicles themselves, and their propagation from one cell to another, from roadway links to turning movements, and vice versa. Gap calculations for the convergence criteria are also found in these files.

7. `node.c` and `node.h`: These files contain the routines for the node processing algorithm described in Sections 2.2.5 and 3.2.1. If users wish to modify the algorithms used for intersection processing (or to introduce a new intersection control type), these files will be the primary place these modifications are made.

8. `fileio.c` and `fileio.h`: These files contain the routines for reading input data files, and writing output data files. Additional string processing routines are included here to parse metadata tags and values, identify comments in the input line, and so forth.

9. `dta.c` and `dta.h`: These files contain the general control loop for dynamic traffic assignment. In particular, **these files contain the core modules `simulateCTM` and `shiftMSA`, which implement the cell transmission model and method of successive averages.** Additional functions provide for allocation and deallocation of the traffic assignment run structures.

10. `main.c` and `main.h`: These brief files handle control when the program is first run, checking that the necessary number of parameters has been provided, and then transferring control to the DTA processing code.

The following features of the source code warrant further documentation: the units system; the memory allocation and deallocation system; the implementation of linked lists; and network algorithms which function in the dual graph.

The code uses a consistent internal system of units, measuring all times in seconds and all distances in feet. However, it is often more convenient to provide input or output in different units. The follow macros provided in `utils.h` facilitate units conversion:

```
#define HOURS      3600.0
#define MINUTES    60.0
#define SECONDS    1.0
#define MILES      5280.0
#define KILOMETERS 3280.839895
#define METERS     3.280839895
#define FEET       1.0
#define INCHES     0.083333333
```

Multiplying by one of these macros converts a quantity into standard units, while dividing converts it out of standard units. For instance, if `t = 3` and `t` is measured in hours, then `t * HOURS` will give the proper number of seconds. If `u` is measured in seconds (the standard unit), but we want to report the answer in hours, then `t / HOURS` will give the correct number of hours. Caution should be used when units appear in the denominator. As an example, if `v` is measured in miles per hour, then converting to standard units would require `v * MILES / HOURS`, while converting `v` from internal units to miles per hour would require `v / (MILES / HOURS)` or `v * HOURS / MILES`. If a programmer wishes to change the internal system of units, all that needs to be changed are the relative values in this file, and the code will seamlessly change. (The internal units are those corresponding to `1.0` values in this list.)

Memory allocation and deallocation is facilitated by the following macros, which are defined in `datastructures.h`:

```
#define newScalar(y) (y *)allocateScalar(sizeof(y))
#define newVector(u,y) (y *)allocateVector(u,sizeof(y))
#define newMatrix(u1,u2,y) (y **)allocateMatrix(u1,u2,sizeof(y))
#define new3DArray(u1,u2,u3,y) (y ***)allocate3DArray(u1,u2,u3,sizeof(y))

#define declareScalar(y,S) y *S = newScalar(y)
#define declareVector(y,V,u) y *V = newVector(u,y)
#define declareMatrix(y,M,u1,u2) y **M = newMatrix(u1,u2,y)
#define declare3DArray(y,A,u1,u2,u3) y ***A = new3DArray(u1,u2,u3,y)

#define deleteScalar(y) killScalar(y)
#define deleteVector(y) killVector(y)
#define deleteMatrix(y,u1) killMatrix((void **)y,u1)
#define delete3DArray(y,u1,u2) kill3DArray((void ***)y,u1,u2)
```

These routines allow users to allocate and deallocate memory for scalars (single variables), vectors (one-dimensional arrays), matrices (two-dimensional arrays), and three-dimensional arrays. To use the `newScalar`, `newVector`, `newMatrix`, and `new3DArray` macros, the relevant dimensions are specified, followed by the data type. These are used as right-hand expressions; the left-hand side should be a pointer of the appropriate type. For instance, to allocate a vector of 5 `double`s, the code

```
double x* = newVector(5, double)
```

can be used. For convenience, a more compact shorthand is available when simultaneously declaring and allocating an array, the macros `declareScalar`, `declareVector`, `declareMatrix`, and `declare3DArray` which are used as follows:

```
declareVector(double, x, 5)
```

which is identical to the previous code snippet.

To deallocate memory, the relevant `deleteX` macro can be called. For multidimensional arrays, you must specify all dimensions except the last. All of these macros reference routines in `datastructures.c` which include error checking and input validation.

The network data structure is extremely important, and forms the backbone of the cell transmission model. As shown schematically in Figure B.2, the `network_type` data structure contains the following elements:

- `arc`, an array storing every roadway link, of type `arc_type` (see description below).

- `node`, an array storing every intersection, of type `node_type` (see description below).
- `ODT`, an array storing every origin-destination-time combination, of type `ODT_type` (see description below).
- `paths`, a linked list storing every route which has been generated during the simulation run.
- `origin`, an artificial arc where vehicles are placed before they depart on trips.
- `destination`, an artificial arc where vehicles are placed after they complete their trips.
- `sink`, an artificial node corresponding to the `origin` and `destination` arcs.
- `staticOD`, a two-dimensional array storing the demand matrix read from the file.
- A number of parameters specific to the network, such as the total number of vehicles assigned, simulation tick length, sizes of the above arrays, and so forth.

Arcs, nodes, and ODTs form their own structures. An arc contains the following elements:

- `cells`, a linked list representing each of the cells the link has been divided into.
- `turnMovements`, a linked list containing pointers to each turning movement at the downstream end of the link.
- `upstreamMovements`, a linked list containing pointers to each turning movement at the upstream end of the link.
- `tail`, a pointer to the node at the upstream end of the intersection.
- `head`, a pointer to the node at the downstream end of the intersection.
- `travelTime`, an array giving the integer travel time for a vehicle entering at each time interval.
- `upstreamCount`, an array giving the cumulative count at the upstream end of the arc at each time interval.
- `downstreamCount`, an array giving the cumulative count at the downstream end of the arc at each time interval.
- `freeFlowToDest`, an array giving the free-flow travel time to each destination (used in time-dependent A*).
- `freeFlowMovement`, an array storing the fastest routes at free-flow (used if the time horizon will be exceeded).
- A number of parameters specific to the link, such as the jam density, free-flow travel time, and so forth.

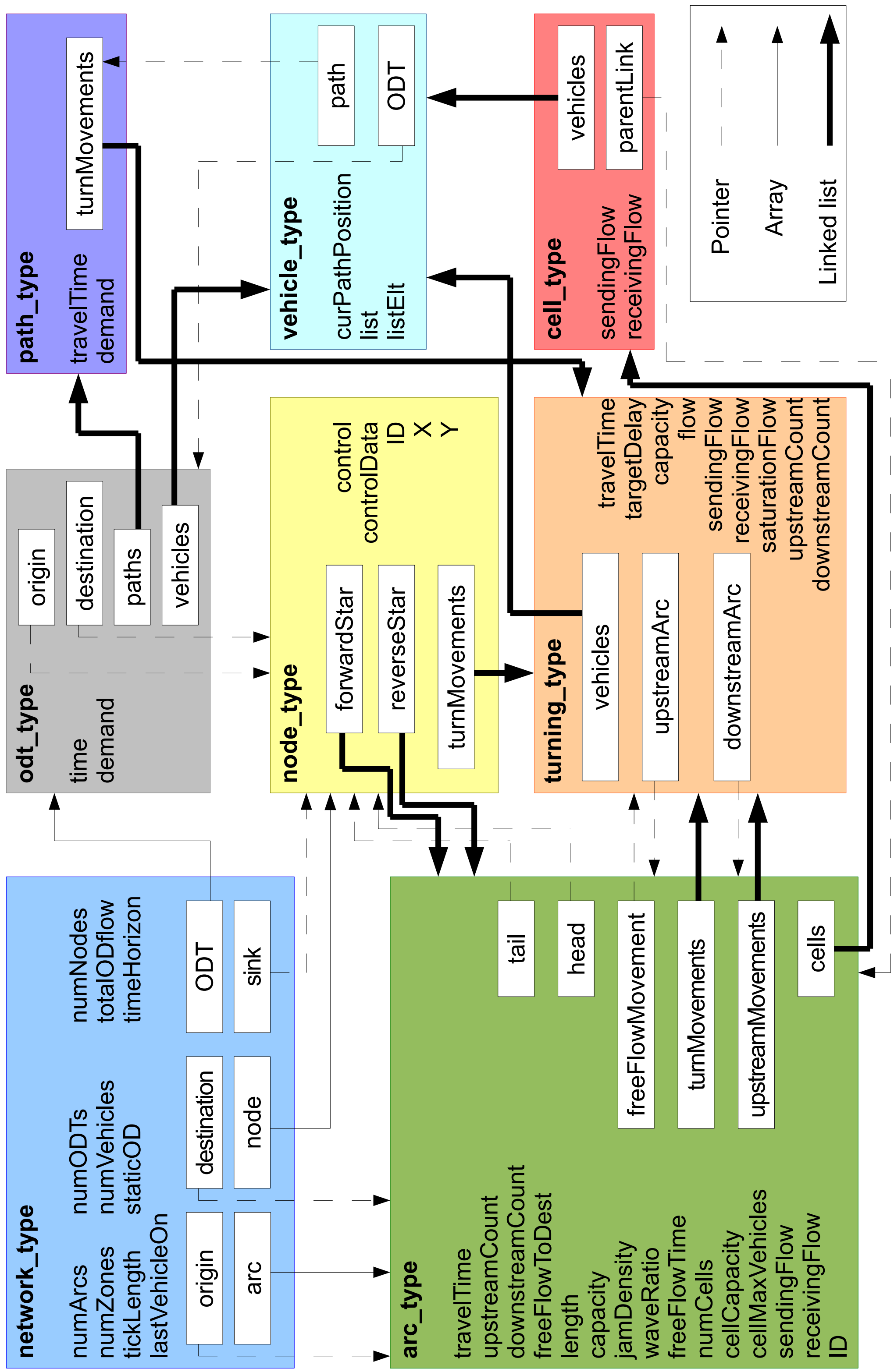


Figure B.2: Schematic of network-related data structures.

A node contains the following elements:

- `forwardStar`, a linked list of the arcs emanating from this node.
- `reverseStar`, a linked list of the arcs terminating at this node.
- `turnMovements`, a linked list of all turning movements corresponding to this intersection.
- `intersectionControl`, an `enum` specifying the traffic control method at the intersection.
- `controlData`, a data structure which may be created for particular traffic control types, and which takes different forms based on the control type (see `node.h` for details).

An ODT contains the following elements:

- `origin`, a pointer to the origin centroid.
- `destination`, a pointer to the destination centroid.
- `departureTime`, the departure time corresponding to this ODT.
- `demand`, the number of vehicles corresponding to this ODT.
- `vehicles`, a doubly linked list containing pointers to each vehicle associated with this ODT.
- `paths`, a linked list of routes which are used by vehicles from this ODT.

A turning movement contains the following elements:

- `vehicles`, a doubly linked list containing pointers to each vehicle in the turning movement cell.
- `upstreamArc`, a pointer to the arc that the turning movement is coming from.
- `downstreamArc`, a pointer to the arc that the turning movement is heading to.
- `travelTime`, an array listing the travel time for vehicles entering the turning movement at any time.
- `upstreamCount`, an array listing the upstream cumulative count $N^{\uparrow}$ at any time.
- `downstreamCount`, an array listing the upstream cumulative count $N^{\downarrow}$ at any time.
- A number of parameters specific to the movement, such as its capacity and target delay.

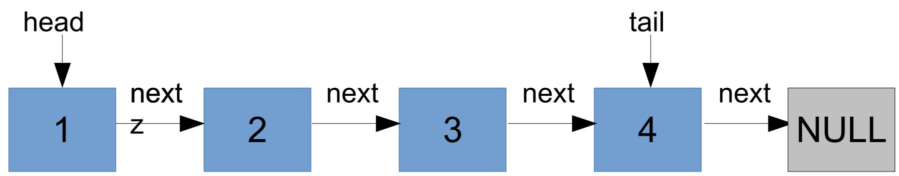


Figure B.3: Schematic of linked list data structures.

Paths, vehicles, and cells are important secondary data structures. A path primarily consists of a linked list of turning movements — it isn't necessary to store the specific links in the path, because a sequence of turning movements implies the roadway links themselves. The vehicle data structure includes the following elements:

- `path`, a pointer to the path the driver of the vehicle is using.
- `curPathPosition`, a pointer into the path's list of turn movements, to show the current position of the vehicle.
- `list`, a pointer to the linked list the vehicle is stored in. Every vehicle is always stored in a vehicleDoublyLinkedList, either on the artificial origin arc, artificial destination arc, or a link or turning movement vehicle list. Storing this pointer greatly speeds up the process of moving vehicles.
- `listElt`, a pointer to the specific position within the linked list the vehicle is stored in. Storing this pointer greatly speeds up the process of moving vehicles.

A cell includes the following elements:

- `vehicles`, a doubly linked list pointing to the vehicles currently in the cell.
- `parentLink`, a pointer to the roadway link the cell belongs to (this avoids having to duplicate all link parameters in each cell).
- `sendingFlow` and `receivingFlow`, used in the flow propagation model (cf. Section 2.2.4).

Many types of linked lists are used in the code, to store variable-length arrays (such as lists of paths, vehicles, cells, and so forth). While slightly different information is stored in each type of linked list, they all have a common form (Figure B.3). The arc linked list provides a good example for demonstration:

```
typedef struct arcLinkedListElt_s {
arc_type *arc;
struct arcLinkedListElt_s *next;
} arcLinkedListElt;

typedef struct arcLinkedList_s {
arcLinkedListElt *head;
arcLinkedListElt *tail;
int size;
} arcLinkedList;
```

where the head and tail are initialized to the `NULL` pointer. Notice that the linked list does not store the arc itself, but a pointer to it. This saves memory and increases computation time. Very frequently, the code must iterate over every element in a linked list. This is commonly done using a `for` loop using the following syntax:

```
for (curArc = network->node[i].forwardStar->head;
curArc != NULL;
curArc = curArc->next) {
...
}
```

Here, `network->node[i].forwardStar` is an `arcLinkedList` storing all the arcs emanating from intersection `i`. `curArc` iterates in turn over every element in the linked list. Notice that `curArc` is *not* the arc element itself, but rather the list element. To access the corresponding arc, you need to use `curArc->arc`.

Appendix C

License agreement

This software is copyrighted by The University of Texas at Austin and the Wyoming Department of Transportation (2013), and distributed under the GPLv2 open-source license, reproduced below:

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and ”any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM ”AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH

YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Appendix D

Simulator code

D.1 Mesoscopic simulation module

D.1.1 `main.c`

```
#include "main.h"

/*
main -- Primary file for running mesoscopic simulation
*/
int main(int numArgs, char *args[]) {

  parameters_type run;
   verbosity = FULL_NOTIFICATIONS;

   #ifdef DEBUG_MODE /* Debug mode enables extra logging.  Define this macro in utils.h */
      debugFile = openFile("logfile.txt", "w");
   #endif

   if (numArgs != 2) displayUsage();

  initializeDTARun(&run, args[1]);
   DTA(&run);

   if (strlen(run.countsFileName)) writeCumulativeCounts(run.network, run.countsFileName);
   if (strlen(run.linkSummaryName)) writeLinkSummary(&run, run.linkSummaryName);
   if (strlen(run.nodeSummaryName)) writeNodeSummary(&run, run.nodeSummaryName);
```

```
  cleanUpDTARun(&run);

   #ifdef DEBUG_MODE
      fclose(debugFile);
   #endif

  return (EXIT_SUCCESS);
}

/*
displayUsage -- provides instructions to the user if the incorrect number of command-line arguments is given.
*/
void displayUsage() {
  fatalError("Must provide exactly one argument (parameters file).");
}
```

D.1.2 **main.h**

```
/*
 Hierarchy of header files (bottom-up):

 utils.h
 datastructures.h
 sampling.h
 network.h
 cell.h
 vehicle.h
 node.h
 fileio.h
 dta.h
 Optional modules can be included here: warrant.h, graphics.h
 main.h

 Declarations referring to lower-level headers can use typedefs; declarations referring to higher-level headers must use structs
*/


#include <stdlib.h>
#include "cell.h"
#include "fileio.h"
#include "utils.h"
#ifdef GRAPHICS_MODE
   #include "graphics.h"
#endif
```

```c
#include "warrant.h"

void displayUsage();
```

D.1.3 `dta.c`

```c
#include "dta.h"

/*
DTA -- controls the main DTA loop.
Arguments:
   run -- pointer to a parameters_type containing all the DTA run parameters
*/
void DTA(parameters_type *run) {
  network_type *network = run->network;
  bool isConverged = FALSE;
  int iteration = 0;
  clock_t startTime = clock();
  double elapsedTime = 0;
  float gap = INFINITY;

  initializeTravelTimes(network); /* Initializes travel times to free-flow */
  addShortestPaths(network);      /* Add initial SPs to path set for each ODT */
   initializeVehicles(network);
   updateElapsedTime(startTime, &elapsedTime);
   displayMessage(MEDIUM_NOTIFICATIONS, "Ready to start main loop.\n");


  while (isConverged == FALSE) {
    iteration++;
    startTime = clock();

    simulateCTM(network);
    updateAllTravelTimes(network);
    addShortestPaths(network);

    updateElapsedTime(startTime, &elapsedTime);
    gap = averageExcessCost(network);
    displayMessage(MEDIUM_NOTIFICATIONS, "Iteration %d: AEC %f in %f seconds (latest arrival %d)\n", iteration, gap, elapsedTime, latestArrivalTime(network));
    if (gap < run->AECtarget || iteration >= run->maxIterations || elapsedTime > run->maxRunTime) break;

```

```
    shiftMSA(network, 1.0 / (iteration + 1));
  }
}

/*
simulateCTM -- performs cell transmission model loading when all vehicles have already been assigned paths.
Arguments:
   network -- pointer to a network_type containing the network and all relevant parameters

Simulation works as follows:
  In *increasing order of time*
    1. Load vehicles onto origin movements based on *paths*
    2. Identify each cell's sending and receiving flow
    3. Process links:
        Transfer vehicles for *interior cells.*  Find the # of vehicles, move that many from the end of upstream queue to back of downstream queue
    4. Process nodes:
        Go to a splitting function based on the node type.  Works based on sending/receiving flows, also updates cumulative counts
*/
#define CTM_REPORTING_INTERVAL 100
void simulateCTM(network_type *network) {
  int i, ij, t;
  int odt = 0;
  vehicleDoublyLinkedListElt *curVehicle;

  prepareAllTrips(network); /* Load all vehicles on origin link */
  initializeCounts(network);
  initializeNodes(network);
  for (t = 0; t < network->timeHorizon; t++) {

    /* Initialize new cumulative counts to old ones */
    if (t > 0) copyCounts(network, t-1, t);

    /* Load flows at origin nodes.  This implementation exploits the temporal ordering of the ODT array */
    if (odt < network->numODTs) {
      while (network->ODT[odt].departureTime == t) {
        for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
          transferVehicleToMovement(curVehicle->vehicle, &(network->origin), curVehicle->vehicle->curPathPosition->movement, t);
        }
```

```
        if (++odt >= network->numODTs) break;
      }
    }

    /* Calculate sending and receiving flows for all cells, and shift flows within an arc */
    for (ij = 0; ij < network->numArcs; ij++) {
      calculateSendingFlows(&network->arc[ij]);
      calculateReceivingFlows(&network->arc[ij]);
      moveIntralinkVehicles(&network->arc[ij]);
    }

    /* Now process each node, according to its control type */
    for (i = 0; i < network->numNodes; i++) {
      processNode(network, &(network->node[i]), t);
    }

    if (t % CTM_REPORTING_INTERVAL == 0) displayMessage(FULL_NOTIFICATIONS, "Simulated %d of %d ticks (%d%%)\r", t, network->timeHorizon, 100 * t / network->timeHorizon);
  }
  displayMessage(FULL_NOTIFICATIONS, "Simulated %d of %d ticks (%d%%)\n", network->timeHorizon, network->timeHorizon, 100);

  terminateAllTrips(network); /* Clean up any vehicles still left on the network by moving to destination and warn about incomplete trips */
}

/*
shiftMSA -- Changes path choices based on method of successive averages, using stochastic selection.
Arguments:
   network -- pointer to a network_type containing the network and all relevant parameters
   movingFraction -- probability to move vehicles onto the shortest path.  Generally 1/(iteration+1), but this function can be called with any fraction
*/
void shiftMSA(network_type *network, float movingFraction) {
  int odt;
  double minPathCost;
  vehicleDoublyLinkedListElt *curVehicle;
  pathLinkedListElt *curPath;
  path_type *targetPath;

  for (odt = 0; odt < network->numODTs; odt++) {
```

```
    /* Find min-cost path */
    minPathCost = network->timeHorizon + 1;
    for (curPath = network->ODT[odt].paths->head; curPath != NULL; curPath = curPath->next) {
      if (curPath->path->travelTime < minPathCost) {
        minPathCost = curPath->path->travelTime;
        targetPath = curPath->path;
      }
    }
    if (targetPath == NULL) fatalError("No paths exist for odt %d -> %d @ %d\n", network->ODT[odt].origin->ID, network->ODT[odt].destination->ID, network->ODT[odt].departureTime);
    for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
      if (randUniform(0, 1) < movingFraction) {
        curVehicle->vehicle->path->demand--;
        curVehicle->vehicle->path = targetPath;
        curVehicle->vehicle->path->demand++;
      }
    }
  }
}

/*
cleanUpDTARun -- Deallocates memory associated with a DTA run.
Splits into deleteSchedule and deleteNetwork so the latter can be called separately
when memory has only been partially allocated (as in warrant analysis runs)
Arguments:
   run -- pointer to a parameters_type containing all the DTA run parameters
*/
void cleanUpDTARun(parameters_type *run) {
   deleteSchedule(run->network);
   deleteNetwork(run->network);
   if (run->demandProfile == TRIANGLE) deleteScalar(run->profileParameters);
}

/*
deleteSchedule -- Deallocates memory associated with the time-dependent data structures of a network
Arguments:
   network -- pointer to a network_type containing all relevant parameters
*/
```

```
void deleteSchedule(network_type *network) {
  int i, ij, odt;
  cellDoublyLinkedListElt *curCell;
  turningLinkedListElt *curMovement;
  vehicleDoublyLinkedListElt *curVehicle;

  for (ij = 0; ij < network->numArcs; ij++) {
    deleteVector(network->arc[ij].travelTime);
    deleteVector(network->arc[ij].upstreamCount);
    deleteVector(network->arc[ij].downstreamCount);
    for (curCell = network->arc[ij].cells->tail; curCell != NULL;
        curCell = curCell->next) {
      deleteVehicleDoublyLinkedList(curCell->cell->vehicles);
      deleteScalar(curCell->cell);
    }
    deleteCellDoublyLinkedList(network->arc[ij].cells);
    deleteVector(network->arc[ij].freeFlowToDest);
    deleteVector(network->arc[ij].freeFlowMovement);
  }
  deleteVector(network->origin.travelTime);
  deleteVector(network->origin.upstreamCount);
  deleteVector(network->origin.downstreamCount);
  deleteVector(network->destination.travelTime);
  deleteVector(network->destination.upstreamCount);
  deleteVector(network->destination.downstreamCount);

   for (i = 0; i < network->numNodes; i++) {
    for (curMovement = network->node[i].turnMovements->head;
        curMovement != NULL; curMovement = curMovement->next) {
      deleteVector(curMovement->movement->travelTime);
      deleteVector(curMovement->movement->upstreamCount);
      deleteVector(curMovement->movement->downstreamCount);
    }
   }

  for (odt = 0; odt < network->numODTs; odt++) {
    for (curVehicle = network->ODT[odt].vehicles->head; curVehicle !=
        NULL; curVehicle = curVehicle->next) {
      deleteScalar(curVehicle->vehicle);
    }
    deleteVehicleDoublyLinkedList(network->ODT[odt].vehicles);
    deletePathLinkedList(network->ODT[odt].paths);
  }
  deleteVehicleDoublyLinkedList(network->origin.cells->head->cell->
      vehicles);
  deleteVehicleDoublyLinkedList(network->destination.cells->head->cell
      ->vehicles);
```

```
  deleteScalar(network->origin.cells->head->cell);
  deleteScalar(network->destination.cells->head->cell);
  deleteCellDoublyLinkedList(network->origin.cells);
  deleteCellDoublyLinkedList(network->destination.cells);
  deleteVector(network->ODT);
}

/*
deleteNetwork -- Deallocates memory associated with the core data structures of a network
Arguments:
   network -- pointer to a network_type containing all relevant parameters
*/
void deleteNetwork(network_type *network) {
  int i, ij;
  turningLinkedListElt *curMovement;
  pathLinkedListElt *curPath;

  for (ij = 0; ij < network->numArcs; ij++) {
    deleteTurningLinkedList(network->arc[ij].turnMovements);
    deleteTurningLinkedList(network->arc[ij].upstreamMovements);
  }

  for (i = 0; i < network->numNodes; i++) {
    switch (network->node[i].control) {
      case TWO_WAY_STOP:
         deletePriorityLinkedList(((twoWayStop_type *)(network->node[i].controlData))->priorityList);
         deleteScalar(network->node[i].controlData);
         break;
      case BASIC_SIGNAL:
         deleteLinkedList(((basicSignal_type *)(network->node[i].controlData))->greenTime);
         deleteScalar(network->node[i].controlData);
         break;
      case CENTROID:
      case NONHOMOGENEOUS:
      case DIVERGE:
      case MERGE:
      case FOUR_WAY_STOP:
      case FANCY_SIGNAL:
      case INTERCHANGE:
      default: /* Simple intersection types do not require additional cleanup */
         break;
    }
```

```
    deleteArcLinkedList(network->node[i].forwardStar);
    deleteArcLinkedList(network->node[i].reverseStar);
    for (curMovement = network->node[i].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
      deleteVehicleDoublyLinkedList(curMovement->movement->vehicles);
      deleteScalar(curMovement->movement);
    }
    deleteTurningLinkedList(network->node[i].turnMovements);
  }

  for (curPath = network->paths->head; curPath != NULL; curPath = curPath->next) {
    deletePath(curPath->path);
  }
  deleteTurningLinkedList(network->origin.turnMovements);
  deleteTurningLinkedList(network->destination.upstreamMovements);
  deleteVector(network->arc);
  deleteVector(network->node);
  deletePathLinkedList(network->paths);
  deleteMatrix(network->staticOD, network->numZones);
  deleteScalar(network);
}

/*
initializeDTARun -- Read all files and allocate all memory needed for a run
Arguments:
   run -- pointer to a parameters_type which will be initialized during this function
   parametersFileName -- string with the name of the parameters file to read
*/
void initializeDTARun(parameters_type *run,char *parametersFileName) {
  int destination;
  network_type *network = newScalar(network_type);
  run->network = network;

  readParametersFile(run, parametersFileName);

  verbosity = run->verbosity;

  initializeNetwork(run);
  readNodeCoordinateFile(network, run->coordinateFileName);
  readNodeControlFile(network, run->nodeControlFileName);

  validateNetwork(network);
```

```
  generateSchedule(network);
  readDemandFile(run);

  generateCells(network);
  generateVehicles(network);
  initializeCounts(network);

  for (destination = 0; destination < network->numZones; destination++) {
    calculateFreeFlowSPLabels(network, destination);
  }

  displayMessage(FULL_NOTIFICATIONS, "Finished initializing run.\n");
}

/*
initializeNetwork -- Read data from a network file, and initialize the relevnat network data structuer
Arguments:
   run -- pointer to a parameters_type containing the network which will be initialized during this function.
          This parameters_type contains the filename of the network file.
*/
void initializeNetwork(parameters_type *run) {
  int ij;
  network_type *network = run->network;

  network->numZones = IS_MISSING;
  network->numArcs = IS_MISSING;
  network->numNodes = IS_MISSING;
  network->paths = createPathLinkedList();

  network->sink.ID = 0;
  network->origin.tail = &(network->sink);
  network->origin.head = &(network->sink);
  network->destination.head = &(network->sink);
  network->destination.tail = &(network->sink);

   readNetworkFile(network, run->networkFileName, run->backwardWaveRatio);
   createStarLists(network);

    network->tickLength = run->tickLength;
  network->timeHorizon = ceil(run->timeHorizon / run->tickLength); /* Convert time horizon in seconds to clock ticks */
  network->lastVehicleOn = ceil(run->lastVehicleOn / run->tickLength);
```

```
  for (ij = 0; ij < network->numArcs; ij++) {
    network->arc[ij].freeFlowToDest = newVector(network->numZones, int);
    network->arc[ij].freeFlowMovement = newVector(network->numZones, turning_type *);
  }

}

/**** Generate cell data structures ****/

/*
generateCells -- Create and allocate data structures related to cells in a network, partitioning each link into the
                 right number of cells.
Arguments:
   network -- pointer to a network_type to create cells for
*/
void generateCells(network_type *network) {
  int ij, c;
  cell_type *newCell;

  displayMessage(FULL_NOTIFICATIONS, "Generating cells...\n");

  /* Create cells for artificial origin and destination links */
  network->origin.cells = createCellDoublyLinkedList();
  network->origin.numCells = 1;
  network->origin.cellCapacity = INT_MAX;
  network->origin.cellMaxVehicles = INT_MAX;
  network->origin.waveRatio = 1;
  newCell = newScalar(cell_type);
  newCell->parentLink = &(network->origin);
  newCell->vehicles = createVehicleDoublyLinkedList();
  newCell->sendingFlow = 0;
  newCell->receivingFlow = 0;
  insertCellDoublyLinkedList(network->origin.cells, newCell, NULL);
  network->destination.cells = createCellDoublyLinkedList();
  network->destination.numCells = 1;
  network->destination.cellCapacity = INT_MAX;
  network->destination.cellMaxVehicles = INT_MAX;
  network->destination.waveRatio = 1;
  newCell = newScalar(cell_type);
  newCell->parentLink = &(network->destination);
  newCell->vehicles = createVehicleDoublyLinkedList();
  newCell->sendingFlow = 0;
  newCell->receivingFlow = 0;
```

```
  insertCellDoublyLinkedList(network->destination.cells, newCell, NULL)
     ;

  /* Create cells for all other links */
  for (ij = 0; ij < network->numArcs; ij++) {
    network->arc[ij].numCells = ceil(network->arc[ij].freeFlowTime /
       network->tickLength);
    network->arc[ij].cellCapacity = ceil(network->arc[ij].capacity *
       network->tickLength);
    network->arc[ij].cellMaxVehicles = ceil(network->arc[ij].length *
       network->arc[ij].jamDensity / network->arc[ij].numCells);
    network->arc[ij].cells = createCellDoublyLinkedList();
    for (c = 0; c < network->arc[ij].numCells; c++) {
      newCell = newScalar(cell_type);
      newCell->parentLink = &(network->arc[ij]);
      newCell->vehicles = createVehicleDoublyLinkedList();
      newCell->sendingFlow = 0;
      newCell->receivingFlow = 0;
      insertCellDoublyLinkedList(network->arc[ij].cells, newCell,
         network->arc[ij].cells->tail);
    }
  }

  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

/**** Network validation ****/

/*
validateNetwork -- Ensure that the network is valid: node control is
   appropriate based on number of
                   incoming and outgoing links; no duplicate turn
                      movements listed; network is
                   properly connected, each destination is reachable
                      from each origin.
Arguments:
   network -- pointer to a network_type for validation
*/
void validateNetwork(network_type *network) {
  displayMessage(MEDIUM_NOTIFICATIONS, "Validating network data...\n");

  validateNodeControl(network);
  checkDuplicateTurnMovements(network);
  checkNetworkConnectivity(network);

  displayMessage(MEDIUM_NOTIFICATIONS, "...validation complete.\n");
}
```

```c

/*
validateNodeControl -- Check that each node's assigned control is compatible with the number of
                       entering and leaving links.
Arguments:
   network -- pointer to a network_type for validation
*/
void validateNodeControl(network_type *network) {
  int i;

  displayMessage(FULL_NOTIFICATIONS, "Checking node control...\n");

  /* Confirm all nodes have a relevant control file entry and finite tick length */
  for (i = 0; i < network->numNodes; i++) {
    if (network->node[i].control == UNKNOWN_CONTROL) fatalError("Node %d either missing in node control file, or unknown control type.", i+1);
  }

  /* Check turning movement compatability with control types (not more than one into a diverge, not more than one out of a merge, etc.) */
  for (i = 0; i < network->numNodes; i++) {
    if (i < network->numZones && network->node[i].control != CENTROID) warning(FULL_NOTIFICATIONS, "Origin/destination %d does not have CENTROID control.", i+1);
    switch (network->node[i].control) {
    case CENTROID:
      if (i > network->numZones) warning(FULL_NOTIFICATIONS, "Centroid node type found for node %d which is neither an origin nor a destination.", i+1
      );
      if (network->node[i].reverseStar->size == 0) warning(FULL_NOTIFICATIONS, "No incoming links for centroid %d\n", i+1);
      if (network->node[i].forwardStar->size == 0) warning(FULL_NOTIFICATIONS, "No outgoing links for centroid %d\n", i+1);
      break;
    case NONHOMOGENEOUS:
      if (network->node[i].reverseStar->size > 1) fatalError("Nonhomogeneous node %d has more than one incoming link!", i+1);
      if (network->node[i].forwardStar->size > 1) fatalError("Nonhomogeneous node %d has more than one outgoing link!", i+1)
```

```
;
      if (network->node[i].turnMovements->size == 0) fatalError("No turning movements listed for non-centroid node %d", i+1);
      break;
    case MERGE:
      if (network->node[i].reverseStar->size <= 1) fatalError("Merge node %d needs at least two incoming links!", i+1);
      if (network->node[i].forwardStar->size != 1) fatalError("Merge node %d has more than one outgoing link!", i+1);
      if (network->node[i].turnMovements->size == 0) fatalError("No turning movements listed for non-centroid node %d", i+1);
      break;
    case DIVERGE:
      if (network->node[i].reverseStar->size != 1) fatalError("Diverge node %d has more than one incoming link!", i+1);
      if (network->node[i].forwardStar->size <= 1) fatalError("Diverge node %d needs at least two outgoing links!", i+1);
      if (network->node[i].turnMovements->size == 0) fatalError("No turning movements listed for non-centroid node %d", i+1);
      break;
    case TWO_WAY_STOP:
    case FOUR_WAY_STOP:
    case BASIC_SIGNAL:
    case INTERCHANGE:
        if (network->node[i].reverseStar->size == 0) fatalError("No incoming links for non-centroid node %d\n", i+1);
      if (network->node[i].forwardStar->size == 0) fatalError("No outgoing links for non-centroid node %d\n", i+1);
         break;
    case FANCY_SIGNAL:
      warning(LOW_NOTIFICATIONS, "Not all node control types have error checking implemented, skipping node %d.\n", i+1);
         break;
    case UNKNOWN_CONTROL:
    default:
      fatalError("Unknown or missing node control type for node %d.", i+1);
    }
  }

  displayMessage(FULL_NOTIFICATIONS, "...done.\n");
}

/*
checkDuplicateTurnMovements -- Ensure that no turn movement is listed multiple times in a network.
Arguments:
```

```
  network -- pointer to a network_type for validation
*/
void checkDuplicateTurnMovements(network_type *network) {
  int i;
  long curHash, compareHash;
  turningLinkedListElt *curMovementElt, *compareMovementElt;
  turning_type *curMovement, *compareMovement;

  displayMessage(FULL_NOTIFICATIONS, "Checking for duplicate turning movements...\n");
  for (i = 0; i < network->numNodes; i++) {
    for (curMovementElt = network->node[i].turnMovements->head; curMovementElt != NULL; curMovementElt = curMovementElt->next) {
      curMovement = curMovementElt->movement;
      curHash = ptr2arc(network, curMovement->upstreamArc) * network->numArcs + ptr2arc(network, curMovement->downstreamArc);
      for (compareMovementElt = curMovementElt->next; compareMovementElt != NULL; compareMovementElt = compareMovementElt->next) {
        compareMovement = compareMovementElt->movement;
        compareHash = ptr2arc(network, compareMovement->upstreamArc) * network->numArcs + ptr2arc(network, compareMovement->downstreamArc);
        if (curHash == compareHash) fatalError("Duplicate turning movements for node %d", i+1);
      }
    }
  }

  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

/**** Network data structure creation ****/

/*
generateSchedule -- Dimension and allocate memory for all time-related objects
Arguments:
  network -- pointer to the relevant network_type
*/
void generateSchedule(network_type *network) {
  int i, ij;
  turningLinkedListElt *curMovement;

  /* Dimension turning movements */
  for (i = 0; i < network->numNodes; i++) {
    for (curMovement = network->node[i].turnMovements->head;
```

```
        curMovement != NULL; curMovement = curMovement->next) {
       curMovement->movement->travelTime = newVector(network->timeHorizon, int);
       curMovement->movement->upstreamCount = newVector(network->timeHorizon, int);
       curMovement->movement->downstreamCount = newVector(network->timeHorizon, int);
     }
   }

   /* Dimension all links */
   for (ij = 0; ij < network->numArcs; ij++) {
     network->arc[ij].travelTime = newVector(network->timeHorizon, int);
     network->arc[ij].upstreamCount = newVector(network->timeHorizon, int);
     network->arc[ij].downstreamCount = newVector(network->timeHorizon, int);
   }
    network->origin.travelTime = newVector(network->timeHorizon, int);
    network->origin.upstreamCount = newVector(network->timeHorizon, int);
    network->origin.downstreamCount = newVector(network->timeHorizon, int);
    network->destination.travelTime = newVector(network->timeHorizon, int);
    network->destination.upstreamCount = newVector(network->timeHorizon, int);
    network->destination.downstreamCount = newVector(network->timeHorizon, int);
}

/*
profileDemand -- Determine the number of vehicles departing in each time interval;
                 create the ODT array, which is sorted by time.  This makes it faster to
                 load vehicles during simulation.
Arguments:
   thisRun -- pointer to the parameters_type with all run parameters (including profile type
              and relevant parameters)
*/
void profileDemand(parameters_type *thisRun) {
  long r, s, t, odt = 0;
  long numODTs = 0, newVehicles, numVehicles = 0;
  network_type *network = thisRun->network;
  declareVector(float, proportion, network->timeHorizon);
```

```
declareVector(float, demand, network->timeHorizon);

/* Identify demand profile */
switch (thisRun->demandProfile) {
case UNIFORM_PROFILE:
  profileDemandUniform(proportion, network->lastVehicleOn, network->timeHorizon);
  break;
 case TRIANGLE:
   profileDemandTriangle(proportion, network, (triangleProfile_type *)(thisRun->profileParameters));
   break;
case PEAK:
case QUADRATIC:
  fatalError("Selected demand profile type not yet implemented.");
 case RAW:
   fatalError("Code should call readRawODTFile for RAW profiles, not profileDemand.");
default:
  fatalError("Unknown demand profile type %d.", thisRun->demandProfile);
}

/* Determine amount of demand in each time interval */
srand(thisRun->randomSeed);
for (t = 0; t < network->timeHorizon; t++) {
  displayMessage(DEBUG, "Demand proportion in time interval %d is %f\n", t, proportion[t]);
  for (r = 0; r < network->numZones; r++) {
  network->staticOD[r][r] = 0;
    for (s = 0; s < network->numZones; s++) {
      demand[t] = network->staticOD[r][s] * proportion[t];
      newVehicles = stochasticRound(demand[t]);
      if (newVehicles > 0) {
        numVehicles += newVehicles;
        numODTs++;
      }
    }
  }
}
network->numVehicles = numVehicles;
network->numODTs = numODTs;
network->ODT = newVector(numODTs, ODT_type);

/* Make second pass through OD table to generate vehicles as needed. (Doing passes separately to save memory, must re-seed RNG) */
srand(thisRun->randomSeed);
```

```
  for (t = 0; t < network->timeHorizon; t++) {
    for (r = 0; r < network->numZones; r++) {
      for (s = 0; s < network->numZones; s++) {
        demand[t] = network->staticOD[r][s] * proportion[t];
        newVehicles = stochasticRound(demand[t]);
        if (newVehicles > 0) {
          network->ODT[odt].origin = &(network->node[r]);
          network->ODT[odt].destination = &(network->node[s]);
          network->ODT[odt].departureTime = t;
          network->ODT[odt].demand = newVehicles;
          network->ODT[odt].vehicles = createVehicleDoublyLinkedList();
          network->ODT[odt].paths = createPathLinkedList();
          odt++;
        }
      }
    }
  }

  deleteVector(proportion);
  deleteVector(demand);
}

/*
profileDemandUniform -- generate a "uniform" proportions array, evenly distributed over the time horizon
Arguments:
   proportion -- array of floats which will be filled by this function
   lastVehicleOn -- last time interval where demand will be loaded
   timeHorizon -- upper bound for all time-dimensioned arrays
*/
void profileDemandUniform(float *proportion, int lastVehicleOn, int timeHorizon) {
  int t;
   fillLinearProfileDemand(proportion, 0, lastVehicleOn, 1, 1);
  for (t = lastVehicleOn; t < timeHorizon; t++) {
     proportion[t] = 0;
  }
}

/*
fillLinearProfileDemand -- fill a (subset of a) proportions array with proportions that change linearly with time
Arguments:
   proportion -- array of floats which will be filled by this function
   firstVehicleOn -- first time interval to fill in
   lastVehicleOn -- last time interval to fill in
   firstLastRatio -- ratio between demand at the last interval and
```

```
      first interval (>1 for increasing, =1 for constant, <1 for decreasing)
   totalProportion -- total proportion to assign between first and last intervals (1 for all demand, 1/2 for half of total demand, etc.)
*/
void fillLinearProfileDemand(float *proportion, int firstVehicleOn, int lastVehicleOn, float firstLastRatio, float totalProportion) {
   int t = firstVehicleOn;
   float base = 2 / ((lastVehicleOn - firstVehicleOn) * (1 + firstLastRatio)) * totalProportion;
   float slope = base * (firstLastRatio - 1) / (lastVehicleOn - firstVehicleOn - 1);

   if (firstLastRatio < 0) fatalError("fillLinearprofileDemand: firstLastRatio must be nonnegative!");
   if (firstVehicleOn > lastVehicleOn) fatalError("fillLinearProfileDemand: firstVehicleOn must be no greater than lastVehicleOn");

   displayMessage(FULL_DEBUG, "Drawing line with base %f and slope %f from %d to %d\n", base, slope, firstVehicleOn, lastVehicleOn);

   proportion[t++] = base;
   for (; t < lastVehicleOn; t++) {
      proportion[t] = proportion[t - 1] + slope;
   }
}

/*
profileDemandTriangle -- generate a "triangular" proportions array, as specified by the arguments
Arguments:
   proportion -- array of floats which will be filled by this function
   network -- pointer to a network_type containing tick length (needed to convert peak time, which is in external units, to simulation time intervals)
   parameters -- pointer to a triangleProfile_type containing relevant data for the triangle profile (ratios, peak time, etc.)
*/
void profileDemandTriangle(float *proportion, network_type *network, triangleProfile_type *parameters) {
  int t;
  float a, b, c; /* Demand levels at initial, peak, and final time intervals */
  int lastVehicleOn = network->lastVehicleOn;
  int timeHorizon = network->timeHorizon;
  int peakTime = ceil(parameters->peakTime / network->tickLength);
```

```
  float firstRatio = parameters->ratio1, lastRatio = parameters->ratio2;

  /* Validate data */
  if (peakTime > lastVehicleOn || peakTime < 0) fatalError("TRIANGLE profile: peak demand time interval %d is out of range (raw time %f)\n", peakTime, parameters->peakTime);
  if (firstRatio < 0 || lastRatio < 0) fatalError("TRIANGLE profile: ratios 1 and 2 must be nonnegative.\n");

  /* Convert into proper format */

  a = 2 / (peakTime * (1 + firstRatio) + (lastVehicleOn - peakTime + 1) * firstRatio * (1 + lastRatio) - 2 * firstRatio);
  b = a * firstRatio;
  c = b * lastRatio;
  displayMessage(FULL_DEBUG, "Triangle parameters a=%f b=%f c=%f\n", a, b, c);
  displayMessage(FULL_DEBUG, "Drawing lines with total proportion %f and %f\n", peakTime * (a+b)/2,  (lastVehicleOn - peakTime + 1) * (b + c) / 2);
  fillLinearProfileDemand(proportion, 0, peakTime, firstRatio, peakTime * (a + b) / 2);
  fillLinearProfileDemand(proportion, peakTime - 1, lastVehicleOn, lastRatio, (lastVehicleOn - peakTime + 1) * (b + c) / 2);
  for (t = lastVehicleOn; t < timeHorizon; t++) {
    proportion[t] = 0;
  }
}
```

D.1.4 dta.h

```
#ifndef _DTA_H_
#define _DTA_H_

#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "datastructures.h"
#include "fileio.h"
#include "network.h"
#include "node.h"
#include "sampling.h"
#include "utils.h"
#include "vehicle.h"

typedef enum {
```

```c
  UNKNOWN_PROFILE,
  UNIFORM_PROFILE, /* UNIFORM clashes with sampling.h's UNIFORM distribution */
  TRIANGLE,
  QUADRATIC,
  PEAK,
  RAW
} profile_enum;

typedef enum {
  MSA,
  LUCE
} algorithm_enum;

typedef struct parameters_type_s {
   char           networkFileName[STRING_SIZE];
   char           demandFileName[STRING_SIZE];
   char           coordinateFileName[STRING_SIZE];
   char           nodeControlFileName[STRING_SIZE];
   char           countsFileName[STRING_SIZE];
   char           linkSummaryName[STRING_SIZE];
   char           nodeSummaryName[STRING_SIZE];
   char           graphicsFileName[STRING_SIZE];
   profile_enum demandProfile;
   void           *profileParameters;  /* Additional optional parameters for different profile types */
   network_type   *network;
   algorithm_enum solutionAlgorithm;
   long           timeHorizon;         /* End of simulation, in *seconds*                   */
   long           warmUpLength;        /* Warm-up period, in *seconds* (for summaries)       */
   long           coolDownLength;      /* Cool-down period, in *seconds* (for summaries)     */
   long           lastVehicleOn;       /* Time last vehicle enters the network, in *seconds* */
   float          maxRunTime;          /* Seconds                                            */
   int        maxIterations;
   float          AECtarget;           /* Average excess cost termination level             */
   float        vehicleLength;      /* Feet                                              */
   float        demandMultiplier;
   float          tickLength;
   float        backwardWaveRatio;
  int        verbosity;
```

```
  int        randomSeed;
} parameters_type ;

typedef struct triangleProfile_type_s {
   long           peakTime;
   float          ratio1;
   float          ratio2;
} triangleProfile_type;

/**** Core DTA routines ****/

void DTA(parameters_type *run);
void simulateCTM(network_type *network);
void shiftMSA(network_type *network, float movingFraction);
void cleanUpDTARun(parameters_type *run);
void deleteSchedule(network_type *network);
void deleteNetwork(network_type *network);

void initializeDTARun(parameters_type *run,char *parametersFileName);
void initializeNetwork(parameters_type *run);
void generateCells(network_type *network);

/**** Network validation ****/

void validateNetwork(network_type *network);
void validateNodeControl(network_type *network);
void checkDuplicateTurnMovements(network_type *network);

/**** Network data structure creation ****/

void generateSchedule(network_type *network);

/**** Demand profiling ****/

void profileDemand(parameters_type *thisRun);
void profileDemandUniform(float *proportion, int lastVehicleOn, int
   timeHorizon);
void fillLinearProfileDemand(float *proportion, int firstVehicleOn, int
    lastVehicleOn, float firstLastRatio, float totalProportion);
void profileDemandTriangle(float *proportion, network_type *network,
   triangleProfile_type *parameters);

#endif
```

D.1.5 `fileio.c`

```
#include  "fileio.h"
```

```

/********************
 ** File processing **
 ********************/

/*
openFile -- wrapper for the library fopen function with error checking
Arguments:
   filename -- name of file to open
   access -- access mode for opening (same syntax as fopen)
*/
FILE *openFile(char *filename, char *access) {
  FILE *handle = fopen(filename, access);
  if (handle == NULL) fatalError("File %s not found", filename);
  return handle;
}

/**************************
 ** Reading network files **
 **************************/

/* Assumptions:
  All input data is in units of miles/hour; program converts to internal units
*/

/*
readNetworkFile -- reads a text network file for use in simulation
Arguments:
   network -- pointer to network_type to read in
   networkFileName -- name of network file
   backwardWaveRatio -- default delta value to use; typically set in parameters file
*/
void readNetworkFile(network_type *network, char *networkFileName, float backwardWaveRatio) {
  int i, j;
  int numParams, status;

  char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE];
  char metadataTag[STRING_SIZE], metadataValue[STRING_SIZE];

  FILE *linkFile = openFile(networkFileName, "r");

  bool endofMetadata = FALSE;
  float speedLimit;
  char checkChar;
```

```
int tail, head;

/* Read link file metadata */
do {
  if (fgets(fullLine, STRING_SIZE, linkFile) == NULL) fatalError("
     Network file %s ended before metadata complete.",
     networkFileName);
  status = parseMetadata(fullLine, metadataTag, metadataValue);
  if (status == BLANK_LINE || status == COMMENT) continue;
  if         (strcmp(metadataTag, "NUMBER OF ZONES") == 0) {
    network->numZones = atoi(metadataValue);
  } else if (strcmp(metadataTag, "NUMBER OF LINKS") == 0) {
    network->numArcs = atoi(metadataValue);
  } else if (strcmp(metadataTag, "NUMBER OF NODES") == 0) {
    network->numNodes = atoi(metadataValue);
  } else if (strcmp(metadataTag, "END OF METADATA") == 0) {
    endofMetadata = TRUE;
  } else {
    warning(MEDIUM_NOTIFICATIONS, "Ignoring unknown metadata tag %s
       in parameters file %s\n", metadataTag, networkFileName);
  }
} while (endofMetadata == FALSE);

/* Check input for completeness and correctness */
if (network->numZones == IS_MISSING) fatalError("Link file %s does
   not contain number of zones.", networkFileName);
if (network->numNodes == IS_MISSING) fatalError("Link file %s does
   not contain number of nodes.", networkFileName);
if (network->numArcs == IS_MISSING)  fatalError("Link file %s does
   not contain number of links.", networkFileName);
if (network->numZones < 1) fatalError("Link file %s does not contain
   a positive number of nodes.", networkFileName);
if (network->numArcs < 1) fatalError("Link file %s does not contain a
    positive number of links.",  networkFileName);
if (network->numNodes < 1) fatalError("Link file %s does not contain
   a positive number of nodes.", networkFileName);

network->node = newVector(network->numNodes, node_type);
network->arc = newVector(network->numArcs, arc_type);
network->staticOD = newMatrix(network->numZones, network->numZones,
   float);

for (i = 0; i < network->numNodes; i++) {
  network->node[i].ID = i + 1;
}

for (i = 0; i < network->numZones; i++) {
```

```
    for (j = 0; j < network->numZones; j++) {
      network->staticOD[i][j] = 0;
    }
  }

  /* Read link data */
  for (i = 0; i < network->numArcs; i++) {
    do {
      if (fgets(fullLine, STRING_SIZE, linkFile) == NULL) fatalError("
         Link file %s ended before link data complete.",
         networkFileName);
      status = parseLine(fullLine, trimmedLine);
    } while (status == BLANK_LINE || status == COMMENT);
    numParams = sscanf(trimmedLine,"%d %d %f %f %f %f %c",
      &tail,
      &head,
      &network->arc[i].capacity,
      &network->arc[i].length,
      &speedLimit,
      &network->arc[i].jamDensity,
      &checkChar);
    if (numParams != 7 || checkChar != ';') fatalError("Link file %s
       has an error in this line:\n %s", networkFileName, fullLine);
    if (tail < 1 || tail > network->numNodes) fatalError("Arc tail %d
       out of range in network file %s.", i, networkFileName);
    if (head < 1 || head > network->numNodes) fatalError("Arc head %d
       out of range in network file %s.", i, networkFileName);
    /* Create links to data structures */
    network->arc[i].ID = i + 1;
    network->arc[i].tail = &(network->node[tail-1]);
    network->arc[i].head = &(network->node[head-1]);
    /* Convert units to internal units.  Multiply by units in the
       numerator, divide by units in the denominator.  Unit definitions
        in dta.h */
    network->arc[i].length *= FEET;
    network->arc[i].capacity /= HOURS;
    speedLimit *= MILES / HOURS;
    network->arc[i].jamDensity /= MILES;
    /* Check for plausibility assuming trapezoidal fundamental diagram
       */
    if (network->arc[i].length < 0) fatalError("Arc length %d negative
       in network file %s.\n%s", i+1, networkFileName, fullLine);
    if (speedLimit <= 0) fatalError("Arc speed limit %d nonpositive in
       network file %s.\n%s", i+1, networkFileName, fullLine);
    if (network->arc[i].capacity <= 0) fatalError("Capacity %d
       nonpositive in network file %s.\n%s", i+1, networkFileName,
       fullLine);
```

```
    if (network->arc[i].jamDensity <= 0) fatalError("Jam density %d nonpositive in network file %s.\n%s", i+1, networkFileName, fullLine);
    /* if (network->arc[i].jamDensity <= network->arc[i].capacity / speedLimit) warning(FULL_NOTIFICATIONS, "Arc jam density %d too small to be consistent with specified capacity and speed in network file %s.\n%s\nFundamental diagram for this link will be triangular, not trapezoidal.\n", i+1, networkFileName, fullLine); */
    network->arc[i].freeFlowTime = network->arc[i].length / speedLimit;
    network->arc[i].waveRatio = backwardWaveRatio;
  }

  fclose(linkFile);
  displayMessage(MEDIUM_NOTIFICATIONS, "Network has %d nodes, %d arcs, and %d zones\n", network->numNodes, network->numArcs, network->numZones);
  displayMessage(LOW_NOTIFICATIONS, "Network file read and memory allocated.\n");
}

/*
readDemandFile -- detects whether demand type is RAW or based on a profile, and calls the appropriate function
Arguments:
   run -- pointer to parameters_type specifying demand type
*/
void readDemandFile(parameters_type *run) {
   if (run->demandProfile == RAW) {
      readRawODTFile(run->network, run->demandFileName, &(run->demandMultiplier));
   } else {
      readStaticODFile(run->network, run->demandFileName, &(run->demandMultiplier));
      profileDemand(run);
   }
}

/*
readRawODTFile -- reads a demand file in the RAW format.  As a general caution: these files only work with a particular time
                  discretization.  The time index is a time *interval*, not a "real" time value
Arguments:
   network -- pointer to network_type which will contain demand
   rawODTFileName -- name of RAW demand file
   demandMultiplier -- constant factor for scaling demand.  Can be
```

```
      provided in parameters file or in demand file
*/
void readRawODTFile(network_type *network, char *rawODTFileName, float *demandMultiplier) {
   int orig, dest, t, odt, numVehicles, numODTs = IS_MISSING;
   int numParams, status, check;
   char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE];
   char metadataTag[STRING_SIZE], metadataValue[STRING_SIZE];
   float rawDemand, totalDemandCheck = 0, totalODFlow = IS_MISSING;
   bool endofMetadata = FALSE;
   FILE *rawODTFile = openFile(rawODTFileName, "r");

  do {
    if (fgets(fullLine, STRING_SIZE, rawODTFile) == NULL) fatalError("Raw ODT file %s ended before metadata complete.", rawODTFileName);
    status = parseMetadata(fullLine, metadataTag, metadataValue);
    if (status == BLANK_LINE || status == COMMENT) continue;
    if        (strcmp(metadataTag, "NUMBER OF ZONES") == 0) {
      check = atoi(metadataValue);
      if (check != network->numZones) fatalError("Number of zones in network file and raw ODT file do not match.");
    } else if (strcmp(metadataTag, "TOTAL OD FLOW") == 0) {
      totalODFlow = atof(metadataValue);
    } else if (strcmp(metadataTag, "DEMAND MULTIPLIER") == 0) {
      *demandMultiplier = atof(metadataValue);
    } else if (strcmp(metadataTag, "NUMBER OF ODTS") == 0) {
        numODTs = atoi(metadataValue);
    } else if (strcmp(metadataTag, "END OF METADATA") == 0) {
      endofMetadata = TRUE;
    } else {
      warning(MEDIUM_NOTIFICATIONS, "Ignoring unknown metadata tag %s in trips file %s\n", metadataTag, rawODTFileName);
    }
  } while (endofMetadata == FALSE);
  network->numODTs = numODTs;
  network->ODT = newVector(numODTs, ODT_type);

  for (odt = 0; odt < numODTs; odt++) {
    do {
      if (fgets(fullLine, STRING_SIZE, rawODTFile) == NULL) fatalError("Raw ODT file %s ended before ODT data complete.", rawODTFileName);
      status = parseLine(fullLine, trimmedLine);
    } while (status == BLANK_LINE || status == COMMENT);
    numParams = sscanf(trimmedLine,"%d %d %d %f", &orig, &dest, &t, &rawDemand);
```

```
    if (numParams != 4) fatalError("Raw ODT file has an error in this line:\n%s", fullLine);
    if (orig < 1 || orig > network->numZones) fatalError("Origin zone %d out of range in raw ODT file.", orig);
    if (dest < 1 || dest > network->numZones) fatalError("Destination zone %d out of range in raw ODT file.", dest);
    if (t < 0 || t > network->lastVehicleOn) fatalError("Departure time %d out of range [0, %d] in raw ODT file.", t, network->lastVehicleOn);
      if (rawDemand < 0) fatalError("Number of vehicles for ODT %d -> %d @ %d must be nonnegative in raw ODT file.", orig, dest, t);
      network->ODT[odt].origin = &(network->node[orig-1]);
      network->ODT[odt].destination = &(network->node[dest-1]);
      network->ODT[odt].departureTime = t;
      numVehicles = stochasticRound(rawDemand * *demandMultiplier);
      network->ODT[odt].demand = numVehicles;
      network->ODT[odt].vehicles = createVehicleDoublyLinkedList();
      network->ODT[odt].paths = createPathLinkedList();
      totalDemandCheck += rawDemand;
  }

  if (totalODFlow != IS_MISSING) displayMessage(FULL_NOTIFICATIONS, "Total demand %f compared to metadata %f\n", totalDemandCheck, totalODFlow);
   network->totalODFlow = totalDemandCheck * *demandMultiplier;
   fclose(rawODTFile);
   displayMessage(LOW_NOTIFICATIONS, "Trip table read.\n");

}

/*
readStaticODFile -- reads a demand matrix file
Arguments:
   network -- pointer to network_type which will contain demand
   staticODFileName -- name of demand matrix file
   demandMultiplier -- constant factor for scaling demand.  Can be provided in parameters file or in demand file
*/
void readStaticODFile(network_type *network, char *staticODFileName, float *demandMultiplier) {
  int i, j;
  int numParams, status, check;
  double demand, totalDemandCheck = 0;

  char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE], *token;
  char metadataTag[STRING_SIZE], metadataValue[STRING_SIZE];

```

```c
bool endofMetadata = FALSE;
double totalODFlow = IS_MISSING;

FILE *tripFile = openFile(staticODFileName, "r");

/* Verify trip table metadata */
do {
  if (fgets(fullLine, STRING_SIZE, tripFile) == NULL) fatalError("Trips file %s ended before metadata complete.", staticODFileName);
  status = parseMetadata(fullLine, metadataTag, metadataValue);
  if (status == BLANK_LINE || status == COMMENT) continue;
  if        (strcmp(metadataTag, "NUMBER OF ZONES") == 0) {
    check = atoi(metadataValue);
    if (check != network->numZones) fatalError("Number of zones in trip and link files do not match.");
  } else if (strcmp(metadataTag, "TOTAL OD FLOW") == 0) {
    totalODFlow = atof(metadataValue);
  } else if (strcmp(metadataTag, "DEMAND MULTIPLIER") == 0) {
    *demandMultiplier = atof(metadataValue);
  } else if (strcmp(metadataTag, "END OF METADATA") == 0) {
    endofMetadata = TRUE;
  } else {
    warning(MEDIUM_NOTIFICATIONS, "Ignoring unknown metadata tag %s in trips file %s\n", metadataTag, staticODFileName);
  }
} while (endofMetadata == FALSE);

/* Now read trip table */
while (!feof(tripFile)) {
  if (fgets(fullLine, STRING_SIZE, tripFile) == NULL) break;
  status = parseLine(fullLine, trimmedLine);
  if (status == BLANK_LINE || status == COMMENT || feof(tripFile)) continue;
  if (strstr(trimmedLine, "Origin") != NULL) {
    sscanf(strstr(trimmedLine, "Origin")+6,"%d", &i);  /* i indexes current origin */
    if (i < 1 || i > network->numNodes) fatalError("Origin %d is out of range in trips file %s", j, staticODFileName);
    continue;
  }
  token = strtok(trimmedLine , ";");
  while (token != NULL && strlen(token) > 1) {
    numParams = sscanf(token, "%d : %lf", &j, &demand);
    if (j < 1 || j > network->numNodes) fatalError("Destination %d is out of range in trips file %s\n%s\n%s", j, staticODFileName, fullLine, token);
```

```
      if (numParams < 2) break;
      network->staticOD[i-1][j-1] = demand * *demandMultiplier;
      if (demand < 0) fatalError("Negative demand from origin %d to destination %d", i, j);
      totalDemandCheck += network->staticOD[i-1][j-1];
      token = strtok(NULL, ";");
    }
    blankInputString(trimmedLine, STRING_SIZE);
  }

  if (totalODFlow != IS_MISSING) displayMessage(FULL_NOTIFICATIONS, "Total demand %f compared to metadata %f\n", totalDemandCheck, totalODFlow);
  /* Regardless of the 'check' value, update the network totalODFlow to the true value */
  network->totalODFlow = totalDemandCheck * *demandMultiplier;
  fclose(tripFile);

   displayMessage(LOW_NOTIFICATIONS, "Trip table read.\n");

}

/*
readNodeCoordinateFile -- reads the node data file
Arguments:
   network -- pointer to network_type which contains nodes
   coordinateFileName -- name of node coordinates file
*/
void readNodeCoordinateFile(network_type *network, char *coordinateFileName) {
  int status;
  char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE];
  float tempX, tempY;

  FILE *coordinateFile = openFile(coordinateFileName, "r");

  int i;
  for (i = 0; i < network->numNodes; i++) {
    network->node[i].X = INFINITY;
    network->node[i].Y = INFINITY;
  }

  while (!feof(coordinateFile)) {
    if (fgets(fullLine, STRING_SIZE, coordinateFile) == NULL) break;
    status = parseLine(fullLine, trimmedLine);
    if (status == BLANK_LINE || status == COMMENT || feof(coordinateFile)) continue;
```

```
    sscanf(trimmedLine, "%d %f %f", &i, &tempX, &tempY);
    if (i < 1 || i > network->numNodes) {
      warning(MEDIUM_NOTIFICATIONS, "Node %d out of range in coordinates file, skipping.\n", i);
      continue;
    }
    network->node[i-1].X = tempX;
    network->node[i-1].Y = tempY;
  }

  for (i = 0; i < network->numNodes; i++) {
    if (network->node[i].X == INFINITY || network->node[i].Y == INFINITY) {
      fatalError("No coordinates found for node %d\n", i+1);
    }
  }
  fclose(coordinateFile);

  displayMessage(FULL_NOTIFICATIONS, "Finished reading node coordinate file.\n");
}

/*
readNodeControlFile -- reads an intersection control file
Arguments:
   network -- pointer to network_type which will contain node control
   nodeControlFileName -- name of intersection control file
*/
void readNodeControlFile(network_type *network, char *nodeControlFileName) {
  int curNode = IS_MISSING;
  int i, status;
  arcLinkedListElt *curArc;
  char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE], controlText[STRING_SIZE];
   twoWayStop_type *twoWayStopControl;
   turning_type *newTurningMovement;
   basicSignal_type *basicSignalControl;

  FILE *nodeControlFile = openFile(nodeControlFileName, "r");
  for (i = 0; i < network->numNodes; i++) {
    network->node[i].turnMovements = createTurningLinkedList();
    network->node[i].control = UNKNOWN_CONTROL;
  }
  for (i = 0; i < network->numArcs; i++) {
    network->arc[i].turnMovements = createTurningLinkedList();
    network->arc[i].upstreamMovements = createTurningLinkedList();
```

```
}

/* Create turn movements for each origin and destination */
network->origin.turnMovements = createTurningLinkedList();
network->destination.upstreamMovements = createTurningLinkedList();
for (i = 0; i < network->numZones; i++) {
  for (curArc = network->node[i].forwardStar->head; curArc != NULL;
     curArc = curArc->next) {
    newTurningMovement = newScalar(turning_type);
    newTurningMovement->upstreamArc = &(network->origin);
    newTurningMovement->downstreamArc = curArc->arc;
    newTurningMovement->vehicles = createVehicleDoublyLinkedList();
    insertTurningLinkedList(network->node[i].turnMovements,
       newTurningMovement, NULL);
    insertTurningLinkedList(network->origin.turnMovements,
       newTurningMovement, NULL);
    insertTurningLinkedList(curArc->arc->upstreamMovements,
       newTurningMovement, NULL);
  }
  for (curArc = network->node[i].reverseStar->head; curArc != NULL;
     curArc = curArc->next) {
    newTurningMovement = newScalar(turning_type);
    newTurningMovement->upstreamArc = curArc->arc;
    newTurningMovement->downstreamArc = &(network->destination);
    newTurningMovement->vehicles = createVehicleDoublyLinkedList();
    insertTurningLinkedList(network->node[i].turnMovements,
       newTurningMovement, NULL);
    insertTurningLinkedList(curArc->arc->turnMovements,
       newTurningMovement, NULL);
    insertTurningLinkedList(network->destination.upstreamMovements,
       newTurningMovement, NULL);
  }
}

while (!feof(nodeControlFile)) {
  if (fgets(fullLine, STRING_SIZE, nodeControlFile) == NULL) break;
  status = parseLine(fullLine, trimmedLine);
  if (status == BLANK_LINE || status == COMMENT) continue;
  if (strncmp(trimmedLine, "Node", 4) == 0) {
    sscanf(trimmedLine, "Node %d : %s", &curNode, controlText);
    if (curNode < 1 || curNode > network->numNodes) {
          warning(FULL_NOTIFICATIONS, "Node out of range in control
             file.  Ignoring input line:\n%s\n", fullLine);
          continue;
    }
    curNode--;
    if        (strcmp(controlText, "FOUR-WAY-STOP") == 0) {
```

```
        network->node[curNode].control = FOUR_WAY_STOP;
      } else if (strcmp(controlText, "INTERCHANGE") == 0) {
        network->node[curNode].control = INTERCHANGE;
      } else if (strcmp(controlText, "TWO-WAY-STOP") == 0) {
        network->node[curNode].control = TWO_WAY_STOP;
        network->node[curNode].controlData = newScalar(twoWayStop_type);
        twoWayStopControl = (twoWayStop_type *)(network->node[curNode].controlData);
        twoWayStopControl->minStopPriority = IS_MISSING;
        twoWayStopControl->priorityList = createPriorityLinkedList();
      } else if (strcmp(controlText, "BASIC-SIGNAL") == 0) {
        network->node[curNode].control = BASIC_SIGNAL;
        network->node[curNode].controlData = newScalar(basicSignal_type);
            basicSignalControl = (basicSignal_type *)(network->node[curNode].controlData);
            basicSignalControl->cycleLength = IS_MISSING;
            basicSignalControl->greenTime = createLinkedList();
      } else if (strcmp(controlText, "FANCY-SIGNAL") == 0) {
        network->node[curNode].control = FANCY_SIGNAL;
        warning(FULL_NOTIFICATIONS, "Fancy signal control not yet implemented!\n");
      } else if (strcmp(controlText, "CENTROID") == 0) {
        network->node[curNode].control = CENTROID;
      } else if (strcmp(controlText, "MERGE") == 0) {
        network->node[curNode].control = MERGE;
      } else if (strcmp(controlText, "DIVERGE") == 0) {
        network->node[curNode].control = DIVERGE;
      } else if (strcmp(controlText, "NONHOMOGENEOUS") == 0) {
        network->node[curNode].control = NONHOMOGENEOUS;
         } else if (strcmp(controlText, "UNKNOWN") == 0) {
            network->node[curNode].control = UNKNOWN_CONTROL;
      } else {
        fatalError("Unknown control type in control file.  Input line is:\n%s", fullLine);
      }
      if (curNode == IS_MISSING) {
        warning(LOW_NOTIFICATIONS, "Non-blank, non-comment line found before a node has been selected!  Input line is:\n%s\n", fullLine);
        continue;
      }
      continue;
    } else {
         readTurnMovement(trimmedLine, &(network->node[curNode]));
    }
```

```
  }
  fclose(nodeControlFile);

  displayMessage(FULL_NOTIFICATIONS, "Finished reading node control file.\n");
}

/*
readParametersFile -- reads the parameters file before starting a run
Arguments:
   thisRun -- pointer to a parameters_type which will have all run parameters
   parametersFileName -- name of node control file
*/
void readParametersFile(struct parameters_type_s *thisRun, char* parametersFileName) {
  int status;
  char fullLine[STRING_SIZE];
  char metadataTag[STRING_SIZE], metadataValue[STRING_SIZE];
  FILE *parametersFile = openFile(parametersFileName, "r");

  /* Initialize (set mandatory values to missing, mandatory strings to length zero, others to defaults) */
  thisRun->networkFileName[0] = '\0';
  thisRun->demandFileName[0] = '\0';
  thisRun->coordinateFileName[0] = '\0';
  thisRun->countsFileName[0] = '\0';
  thisRun->nodeControlFileName[0] = '\0';
   thisRun->graphicsFileName[0] = '\0';
   thisRun->linkSummaryName[0] = '\0';
   thisRun->nodeSummaryName[0] = '\0';
  thisRun->timeHorizon = IS_MISSING;
  thisRun->lastVehicleOn = IS_MISSING;
  thisRun->warmUpLength = IS_MISSING;
  thisRun->coolDownLength = IS_MISSING;
  thisRun->AECtarget = 0;
  thisRun->maxRunTime = INFINITY;
  thisRun->maxIterations = INT_MAX;
  thisRun->demandMultiplier = 1;
  thisRun->tickLength = IS_MISSING;
  thisRun->vehicleLength = IS_MISSING;
  thisRun->backwardWaveRatio = IS_MISSING;
  thisRun->verbosity = MEDIUM_NOTIFICATIONS;
  thisRun->demandProfile = UNKNOWN_PROFILE;
  thisRun->solutionAlgorithm = MSA;
  thisRun->randomSeed = time(NULL);

```

```c
/* Process parameter file */
while (!feof(parametersFile)) {
  do {
    if (fgets(fullLine, STRING_SIZE, parametersFile) == NULL) break;
    status = parseMetadata(fullLine, metadataTag, metadataValue);
  } while (status == BLANK_LINE || status == COMMENT);
  if        (strcmp(metadataTag, "NETWORK FILE") == 0) {
    strcpy(thisRun->networkFileName, metadataValue);
  } else if (strcmp(metadataTag, "DEMAND FILE") == 0) {
    strcpy(thisRun->demandFileName, metadataValue);
  } else if (strcmp(metadataTag, "NODE COORDINATE FILE") == 0) {
    strcpy(thisRun->coordinateFileName, metadataValue);
    /* Two values for compatibility with earlier versions */
  } else if (strcmp(metadataTag, "SUMMARY FILE") == 0 || strcmp(metadataTag, "COUNTS FILE") == 0) {
    strcpy(thisRun->countsFileName, metadataValue);
  } else if (strcmp(metadataTag, "LINK SUMMARY FILE") == 0) {
    strcpy(thisRun->linkSummaryName, metadataValue);
  } else if (strcmp(metadataTag, "NODE SUMMARY FILE") == 0) {
    strcpy(thisRun->nodeSummaryName, metadataValue);
  } else if (strcmp(metadataTag, "TIME HORIZON") == 0) {
    thisRun->timeHorizon = atol(metadataValue);
  } else if (strcmp(metadataTag, "TICK LENGTH") == 0) {
    thisRun->tickLength = atof(metadataValue);
  } else if (strcmp(metadataTag, "LAST VEHICLE ON") == 0) {
    thisRun->lastVehicleOn = atol(metadataValue);
  } else if (strcmp(metadataTag, "WARM UP PERIOD") == 0) {
    thisRun->warmUpLength = atol(metadataValue);
  } else if (strcmp(metadataTag, "COOL DOWN PERIOD") == 0) {
    thisRun->coolDownLength = atol(metadataValue);
  } else if (strcmp(metadataTag, "DELTA") == 0) {
    warning(LOW_NOTIFICATIONS, "Backward wave ratios now set on a link-by-link basis using jam density specified in network file.  Ignoring value in parameters file.\n");
  } else if (strcmp(metadataTag, "AEC TOLERANCE") == 0) {
    thisRun->AECtarget = atof(metadataValue);
  } else if (strcmp(metadataTag, "MAX RUN TIME") == 0) {
    thisRun->maxRunTime = (float) atof(metadataValue);
  } else if (strcmp(metadataTag, "MAX ITERATIONS") == 0) {
    thisRun->maxIterations = atoi(metadataValue);
  } else if (strcmp(metadataTag, "DEMAND MULTIPLIER") == 0) {
    thisRun->demandMultiplier = (float) atof(metadataValue);
  } else if (strcmp(metadataTag, "NODE CONTROL FILE") == 0) {
    strcpy(thisRun->nodeControlFileName, metadataValue);
  } else if (strcmp(metadataTag, "VERBOSITY LEVEL") == 0) {
    thisRun->verbosity = (short) atoi(metadataValue);
  } else if (strcmp(metadataTag, "VEHICLE LENGTH") == 0) {
```

```
      thisRun->vehicleLength = atof(metadataValue);
    } else if (strcmp(metadataTag, "BACKWARD WAVE RATIO") == 0) {
      thisRun->backwardWaveRatio = atof(metadataValue);
    } else if (strcmp(metadataTag, "GRAPHICS PARAMETERS FILE") == 0) {
       strcpy(thisRun->graphicsFileName, metadataValue);
    } else if (strcmp(metadataTag, "RANDOM SEED") == 0) {
      thisRun->randomSeed = atoi(metadataValue);
    } else if (strcmp(metadataTag, "DEMAND PROFILE") == 0) {
      if    (strcmp(metadataValue, "UNIFORM") == 0)   thisRun->demandProfile = UNIFORM;
      else if (strcmp(metadataValue, "PEAK") == 0)    thisRun->demandProfile = PEAK;
      else if (strcmp(metadataValue, "TRIANGLE") == 0) {
         thisRun->demandProfile = TRIANGLE;
         thisRun->profileParameters = newScalar(triangleProfile_type);
          ((triangleProfile_type *)(thisRun->profileParameters))->peakTime = IS_MISSING;
          ((triangleProfile_type *)(thisRun->profileParameters))->ratio1 = IS_MISSING;
          ((triangleProfile_type *)(thisRun->profileParameters))->ratio2 = IS_MISSING;
      }
      else if (strcmp(metadataValue, "QUADRATIC") == 0) thisRun->demandProfile = QUADRATIC;
      else if (strcmp(metadataValue, "RAW") == 0) thisRun->demandProfile = RAW;
      else    fatalError("Unknown profile type %s\n", metadataValue);
    } else if (strcmp(metadataTag, "SOLUTION ALGORITHM") == 0) {
      if    (strcmp(metadataValue, "MSA") == 0)       thisRun->solutionAlgorithm = MSA;
      else if (strcmp(metadataValue, "LUCE") == 0)    thisRun->solutionAlgorithm = LUCE;
      else    fatalError("Unknown algorithm type %s\n", metadataValue);
    } else if (strcmp(metadataTag, "PEAK DEMAND TIME") == 0) {
       if (thisRun->demandProfile == TRIANGLE) {
          ((triangleProfile_type *)(thisRun->profileParameters))->peakTime = atoi(metadataValue);
       } else {
          warning(LOW_NOTIFICATIONS, "Ignoring PEAK DEMAND TIME parameter (must follow definition of the demand profile type as TRIANGLE).\n");
       }
    } else if (strcmp(metadataTag, "RATIO 1") == 0) {
       if (thisRun->demandProfile == TRIANGLE) {
          ((triangleProfile_type *)(thisRun->profileParameters))->ratio1 = atof(metadataValue);
       } else {
```

```
        warning(LOW_NOTIFICATIONS, "Ignoring RATIO 1 parameter (must follow definition of the demand profile type as TRIANGLE).\n");
      }
    } else if (strcmp(metadataTag, "RATIO 2") == 0) {
      if (thisRun->demandProfile == TRIANGLE) {
        ((triangleProfile_type *)(thisRun->profileParameters))->ratio2 = atof(metadataValue);
      } else {
        warning(LOW_NOTIFICATIONS, "Ignoring RATIO 2 parameter (must follow definition of the demand profile type as TRIANGLE).\n");
      }
    } else {
      warning(MEDIUM_NOTIFICATIONS, "Ignoring unknown metadata tag in parameters file - %s\n", metadataTag);
    }
  }

  /* Check mandatory elements are present and validate input */
  if (strlen(thisRun->networkFileName) == 0) fatalError("Missing network file!");
  if (strlen(thisRun->demandFileName) == 0) fatalError("Missing demand file!");
  if (strlen(thisRun->coordinateFileName) == 0) fatalError("Missing node coordinate file!");
  if (thisRun->tickLength == IS_MISSING) { thisRun->tickLength = 6; warning(LOW_NOTIFICATIONS, "No tick length specified... using 6 seconds as default.\n"); }
  if (thisRun->vehicleLength == IS_MISSING) { thisRun->vehicleLength = 20; warning(LOW_NOTIFICATIONS, "No vehicle length specified... using 20 ft as default.\n"); }
  if (thisRun->backwardWaveRatio == IS_MISSING) { thisRun->backwardWaveRatio = 0.5; warning(LOW_NOTIFICATIONS, "No backward wave ratio specified... using 0.5 as default.\n"); }
  if (thisRun->timeHorizon == IS_MISSING) fatalError("Missing time horizon!");
  if (thisRun->lastVehicleOn == IS_MISSING) { thisRun->lastVehicleOn = thisRun->timeHorizon * 0.9; warning(FULL_NOTIFICATIONS, "No last vehicle on provided... setting to 90%% of time horizon.\n"); }
  if (thisRun->demandProfile == UNKNOWN_PROFILE) fatalError("Missing or unknown demand profile!");
  if (thisRun->demandMultiplier < 0) fatalError("Negative demand multiplier!");
  if (thisRun->demandMultiplier == 0) warning(LOW_NOTIFICATIONS, "Demand multiplier is zero -- no trips will be assigned!\n");
  if (thisRun->tickLength <= 0) fatalError("Tick length must be
```

```
positive!");
  if (thisRun->timeHorizon < thisRun->lastVehicleOn) fatalError("Last vehicle enters after time horizon!");
  if (thisRun->timeHorizon < thisRun->tickLength) fatalError("Tick length exceeds time horizon!");
  if (thisRun->maxIterations == INT_MAX && thisRun->maxRunTime == INFINITY && thisRun->AECtarget == 0) warning(LOW_NOTIFICATIONS, "No termination criteria specified... program will run until interrupted manually.\n");
  if (thisRun->demandProfile == TRIANGLE) {
      if (((triangleProfile_type *)(thisRun->profileParameters))->peakTime == IS_MISSING) fatalError("TRIANGLE profile requires specification of PEAK DEMAND TIME.");
      if (((triangleProfile_type *)(thisRun->profileParameters))->ratio1 == IS_MISSING) fatalError("TRIANGLE profile requires specification of RATIO 1.");
      if (((triangleProfile_type *)(thisRun->profileParameters))->ratio2 == IS_MISSING) fatalError("TRIANGLE profile requires specification of RATIO 2.");
  }
  if ((strlen(thisRun->linkSummaryName) > 0 || strlen(thisRun->nodeSummaryName) > 0)
        && (thisRun->warmUpLength == IS_MISSING || thisRun->coolDownLength == IS_MISSING)) {
      fatalError("Must provide warm-up and cool-down periods to generate summary files.\n");
   }

  fclose(parametersFile);
  displayMessage(FULL_NOTIFICATIONS, "Finished reading parameters file.\n");
}

void displayRunParameters(int minVerbosity, parameters_type *run) {
  displayMessage(minVerbosity, "Displaying run parameters:\n\n");
  displayMessage(minVerbosity, "Network file: %s\n", run->networkFileName);
  displayMessage(minVerbosity, "Demand file: %s\n", run->demandFileName);
  displayMessage(minVerbosity, "Node coordinate file: %s\n", run->coordinateFileName);
  displayMessage(minVerbosity, "Node control file: %s\n", run->nodeControlFileName);
  displayMessage(minVerbosity, "Counts file: %s\n", run->countsFileName);
  displayMessage(minVerbosity, "Node control file: %s\n", run->nodeControlFileName);
```

```
  displayMessage(minVerbosity, "Time horizon: %ld\n", run->timeHorizon);
  displayMessage(minVerbosity, "AEC tolerance: %f\n", run->AECtarget);
  displayMessage(minVerbosity, "Max running time: %f\n", run->maxRunTime);
  displayMessage(minVerbosity, "Vehicle length: %f\n", run->vehicleLength);
}

/*
writeLinkSummary -- create the link summary file after the DTA run is finished (or based on a counts file)
Arguments:
   run -- pointer to a parameters_type which has all run parameters
   linkSummaryName -- name of file to write link summary data to
*/
void writeLinkSummary(parameters_type *run, char *linkSummaryName) {
  int ij, t;
  int last15Volume, peak15Volume, last60Volume, peak60Volume;
  float time, delay, density, volume, PHF;
  int startTime = run->warmUpLength / run->tickLength, endTime = (run->timeHorizon - run->coolDownLength) / run->tickLength;
  int numPeriods = endTime - startTime;
  int length15 = 15 * MINUTES / run->tickLength;
  int length60 = 60 * MINUTES / run->tickLength;
  network_type *network = run->network;
  FILE *summaryFile = openFile(linkSummaryName, "w");

  displayMessage(FULL_NOTIFICATIONS, "Writing link summary file...");
  if (numPeriods < 1) {
     warning(LOW_NOTIFICATIONS, "Can't generate link summary file, entire run is warm-up or cool-down.\n");
     return;
  } else if (numPeriods < length60) {
      warning(LOW_NOTIFICATIONS, "Insufficient time horizon to calculate peak-hour factors.\n");
  }

  /* Output link cumulative counts */
  fprintf(summaryFile, "LINK SUMMARY (ALL VALUES TIME AVERAGES)\n");
  fprintf(summaryFile, "---------------------------------------\n");
  fprintf(summaryFile, "Link\tTravel time (s)\tDelay (s)\tDensity (veh/mi)\tVolume (veh/hr)\tPHF\n");
  for (ij = 0; ij < network->numArcs; ij++) {
    fprintf(summaryFile, "(%d,%d)\t", network->arc[ij].tail->ID, network->arc[ij].head->ID);
      /* Calculate link statistics */
```

```
      time = 0; delay = 0; density = 0; volume = 0; last15Volume = 0; last60Volume = 0; peak15Volume = 0; peak60Volume = 0;
      for (t = startTime; t < endTime; t++) {
         time += network->arc[ij].travelTime[t];
         delay += network->arc[ij].travelTime[t] - network->arc[ij].numCells;
         density += network->arc[ij].upstreamCount[t] - network->arc[ij].downstreamCount[t];
         if (t - length15 >= startTime) last15Volume = network->arc[ij].downstreamCount[t] - network->arc[ij].downstreamCount[t - length15];
         if (t - length60 >= startTime) last60Volume = network->arc[ij].downstreamCount[t] - network->arc[ij].downstreamCount[t - length60];
         peak15Volume = max(last15Volume, peak15Volume);
         peak60Volume = max(last60Volume, peak60Volume);
      }
      volume = network->arc[ij].downstreamCount[endTime] - network->arc[ij].downstreamCount[startTime];

      /* Normalize and convert units as necessary */
      time *= network->tickLength / numPeriods;
      delay *= network->tickLength / numPeriods;
      density *= MILES / (network->arc[ij].length * numPeriods);
      volume *= HOURS / (network->tickLength * numPeriods);

      fprintf(summaryFile, "%.0f\t%.0f\t%.0f\t%.0f\t", time, delay, density, volume);
      if (numPeriods >= length60 && peak15Volume > 0) {
         PHF = (float) peak60Volume / (4 * peak15Volume);
         fprintf(summaryFile, "%.2f\n", PHF);
      } else {
         fprintf(summaryFile, "---\n");
      }
   }

   fclose(summaryFile);
   displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

/*
writeNodeSummary -- create the node summary file after the DTA run is finished (or based on a counts file)
Arguments:
   run -- pointer to a parameters_type which has all run parameters
   nodeSummaryName -- name of file to write node summary data to
*/
```

```
void writeNodeSummary(parameters_type *run, char *nodeSummaryName) {
  int i, t;
  int last15Volume, peak15Volume, last60Volume, peak60Volume;
  float delay, volume, PHF;
  int startTime = run->warmUpLength / run->tickLength, endTime = (run->
     timeHorizon - run->coolDownLength) / run->tickLength;
  int numPeriods = endTime - startTime;
  int length15 = 15 * MINUTES / run->tickLength;
  int length60 = 60 * MINUTES / run->tickLength;
  network_type *network = run->network;
   turningLinkedListElt *curMovement;
  FILE *summaryFile = openFile(nodeSummaryName, "w");

  displayMessage(FULL_NOTIFICATIONS, "Writing node summary file...");
  if (numPeriods < 1) {
     warning(LOW_NOTIFICATIONS, "Can't generate node summary file,
        entire run is warm-up or cool-down.\n");
     return;
  } else if (numPeriods < length60) {
      warning(LOW_NOTIFICATIONS, "Insufficient time horizon to
         calculate peak-hour factors.\n");
  }

   /* Output movement data */
   fprintf(summaryFile, "NODE SUMMARY FILE\n");
   fprintf(summaryFile, "-----------------\n");
   fprintf(summaryFile, "\tMovement\tDelay (s)\tVolume (vph)\tPHF\n");
   for (i = 0; i < network->numNodes; i++) {
      fprintf(summaryFile, "Node %d summary\n", network->node[i].ID);
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
         fprintf(summaryFile, "\t%d -> %d -> %d\t", curMovement->
            movement->upstreamArc->tail->ID, network->node[i].ID,
            curMovement->movement->downstreamArc->head->ID);
         delay = 0; volume = 0; last15Volume = 0; last60Volume = 0;
            peak15Volume = 0; peak60Volume = 0;
         for (t = startTime; t < endTime; t++) {
            delay += curMovement->movement->travelTime[t];
            if (t - length15 >= startTime) last15Volume = curMovement->
               movement->downstreamCount[t] - curMovement->movement->
               downstreamCount[t - length15];
            if (t - length60 >= startTime) last60Volume = curMovement->
               movement->downstreamCount[t] - curMovement->movement->
               downstreamCount[t - length60];
            peak15Volume = max(last15Volume, peak15Volume);
            peak60Volume = max(last60Volume, peak60Volume);
         }
```

```
        /* Normalize and convert units as necessary */
        volume = curMovement->movement->downstreamCount[endTime] - curMovement->movement->downstreamCount[startTime];
        delay *= network->tickLength / numPeriods;
        volume *= HOURS / (network->tickLength * numPeriods);
        PHF = (float) peak60Volume / (4 * peak15Volume);
        fprintf(summaryFile, "%.0f\t%.0f\t", delay, volume);
        if (numPeriods >= length60 && peak15Volume > 0) {
           PHF = (float) peak60Volume / (4 * peak15Volume);
           fprintf(summaryFile, "%.2f\n", PHF);
        } else {
           fprintf(summaryFile, "---\n");
        }
     }
   }

  fclose(summaryFile);
  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

/*
writeCumulativeCounts -- create the comprehensive counts file after the DTA run is finished
Arguments:
   network -- pointer to a network_type which contains all link and turn  movement cumulative counts
   countsFileName -- name of file to write counts data to
*/
void writeCumulativeCounts(network_type *network, char *countsFileName) {
  int i, ij, t;
  turningLinkedListElt *curMovement;
  FILE *countsFile = openFile(countsFileName, "w");

  displayMessage(FULL_NOTIFICATIONS, "Writing counts file...");

  /* Output link cumulative counts */
  fprintf(countsFile, "LINK CUMULATIVE COUNTS\n");
  fprintf(countsFile, "----------------------\n");
  fprintf(countsFile, "t");
  for (ij = 0; ij < network->numArcs; ij++) {
    fprintf(countsFile, "\t(%d,%d)\tDownstream\tTime", network->arc[ij].tail->ID, network->arc[ij].head->ID);
  }
  fprintf(countsFile, "\n");
  for (t = 0; t < network->timeHorizon; t++) {
    fprintf(countsFile, "%d", (int) ((t+1) * network->tickLength));
```

```
    for (ij = 0; ij < network->numArcs; ij++) {
      fprintf(countsFile, "\t%d\t%d\t%d", network->arc[ij].upstreamCount[t], network->arc[ij].downstreamCount[t], (int) (network->arc[ij].travelTime[t] * network->tickLength)  );
    }
    fprintf(countsFile, "\n");
  }
  fprintf(countsFile, "\n");

  /* Turning movement cumulative counts */
  fprintf(countsFile, "TURN MOVEMENT CUMULATIVE COUNTS\n");
  fprintf(countsFile, "-------------------------------\n");
  fprintf(countsFile, "t");
  for (i = network->numZones; i < network->numNodes; i++) {
    for (curMovement = network->node[i].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
      fprintf(countsFile, "\t%d->%d->%d\tDownstream\tTime", curMovement->movement->upstreamArc->tail->ID, i+1, curMovement->movement->downstreamArc->head->ID);
    }
  }
  fprintf(countsFile, "\n");
  for (t = 0; t < network->timeHorizon; t++) {
    fprintf(countsFile, "%d", (int) ((t+1) * network->tickLength));
    for (i = network->numZones; i < network->numNodes; i++) {
      for (curMovement = network->node[i].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
        fprintf(countsFile, "\t%d\t%d\t%d", curMovement->movement->upstreamCount[t], curMovement->movement->downstreamCount[t], (int) (curMovement->movement->travelTime[t] * network->tickLength));
      }
    }
    fprintf(countsFile, "\n");
  }

  fclose(countsFile);
  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}


/*
readCumulativeCounts -- Reads a cumulative count file, but it must have been *written by writeCumulativeCounts*.
                        Unlike the other file I/O routines, this one is very picky about formatting.
Arguments:
```

```
   network -- pointer to a network_type into which counts will be read
   countsFileName -- name of file to read counts data from
*/
void readCumulativeCounts(network_type *network, char *countsFileName) {
  int i, ij, t, checkTime;
  turningLinkedListElt *curMovement;
  char fullLine[STRING_SIZE];
  FILE *countsFile = openFile(countsFileName, "r");

  displayMessage(FULL_NOTIFICATIONS, "Reading counts file...");

  /* Input link cumulative counts -- skip first three lines ("LINK CUMULATIVE COUNTS", dashes, and header) */
  fgets(fullLine, STRING_SIZE, countsFile);
  fgets(fullLine, STRING_SIZE, countsFile);
  do {
      fgets(fullLine, STRING_SIZE, countsFile);
  } while (strstr(fullLine, "\n") == NULL);
  for (t = 0; t < network->timeHorizon; t++) {
    fscanf(countsFile, "%d", &checkTime);
    if (checkTime != (int) ((t+1) * network->tickLength)) fatalError("Counts file doesn't match network.");
    for (ij = 0; ij < network->numArcs; ij++) {
      if (fscanf(countsFile, "\t%d\t%d\t%d",
              &(network->arc[ij].upstreamCount[t]),
              &(network->arc[ij].downstreamCount[t]),
              &checkTime) != 3) fatalError("Counts file doesn't match network.");
         network->arc[ij].travelTime[t] /= network->tickLength;
    }
    fscanf(countsFile, "\n");
  }
  fscanf(countsFile, "\n");

  /* Turning movement cumulative counts -- again skip first three lines */
  fgets(fullLine, STRING_SIZE, countsFile);
  fgets(fullLine, STRING_SIZE, countsFile);
  do {
      fgets(fullLine, STRING_SIZE, countsFile);
  } while (strstr(fullLine, "\n") == NULL);
  for (t = 0; t < network->timeHorizon; t++) {
    fscanf(countsFile, "%d", &checkTime);
    if (checkTime != (int) ((t+1) * network->tickLength)) fatalError("Counts file doesn't match network.");
    for (i = network->numZones; i < network->numNodes; i++) {
```

```
      for (curMovement = network->node[i].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
         if (fscanf(countsFile, "\t%d\t%d\t%d",
               &(curMovement->movement->upstreamCount[t]),
               &(curMovement->movement->downstreamCount[t]),
               &checkTime) != 3) fatalError("Counts file doesn't match network.");
         curMovement->movement->travelTime[t] /= network->tickLength;
      }
    }
    fscanf(countsFile, "\n");
  }
  fscanf(countsFile, "\n");

  fclose(countsFile);
  displayMessage(FULL_NOTIFICATIONS, "done.\n");

}

/*
writeNodeControlFile -- Outputs current node control information into an intersection control file; often used after performing warrant analysis.
Arguments:
   network -- pointer to a network_type from which intersection data will be read
   nodeControlFileName -- name of control file to write data to
*/
void writeNodeControlFile(network_type *network,  char *nodeControlFileName) {
   int i;

    FILE *nodeControlFile = openFile(nodeControlFileName, "w");

   for (i = 0; i < network->numNodes; i++) {
      writeNode(nodeControlFile, network, i); /* Split into separate function to allow single-node analysis */
   }

   fclose(nodeControlFile);
}

/*
writeNode -- Writes control data for a specific node, depending on its type.
Arguments:
```

```
   nodeControlFileName -- name of control file to write data to
   network -- pointer to a network_type from which intersection data
      will be read
   i -- node number to write
*/
void writeNode(FILE *nodeControlFile, network_type *network, int i) {
   int priority;
   turningLinkedListElt *curMovement, *priorityMovement;
   basicSignal_type *signalData;
   twoWayStop_type *stopData;
   priorityLinkedListElt *curPriority;
   linkedListElt *curGreen;

   fprintf(nodeControlFile, "Node %d : ", i+1);
   switch (network->node[i].control) {
   case CENTROID:
      fprintf(nodeControlFile, " CENTROID\n");
      break;
   case NONHOMOGENEOUS:
      fprintf(nodeControlFile, " NONHOMOGENEOUS\n");
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
         fprintf(nodeControlFile, "\t%d -> %d -> %d\t%f\n", curMovement
            ->movement->upstreamArc->tail->ID, i+1, curMovement->
            movement->downstreamArc->head->ID, curMovement->movement->
            saturationFlow * HOURS);
      }
      break;
   case MERGE:
      fprintf(nodeControlFile, " MERGE\n");
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
         fprintf(nodeControlFile, "\t%d -> %d -> %d\t%f\n", curMovement
            ->movement->upstreamArc->tail->ID, i+1, curMovement->
            movement->downstreamArc->head->ID, curMovement->movement->
            saturationFlow * HOURS);
      }
      break;
   case DIVERGE:
      fprintf(nodeControlFile, " DIVERGE\n");
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
         fprintf(nodeControlFile, "\t%d -> %d -> %d\t%f\n", curMovement
            ->movement->upstreamArc->tail->ID, i+1, curMovement->
            movement->downstreamArc->head->ID, curMovement->movement->
            saturationFlow * HOURS);
      }
```

```c
        break;
     case FOUR_WAY_STOP:
        fprintf(nodeControlFile, " FOUR-WAY-STOP\n");
        for (curMovement = network->node[i].turnMovements->head;
           curMovement != NULL; curMovement = curMovement->next) {
           fprintf(nodeControlFile, "\t%d -> %d -> %d\t%f\n", curMovement
              ->movement->upstreamArc->tail->ID, i+1, curMovement->
              movement->downstreamArc->head->ID, curMovement->movement->
              saturationFlow * HOURS);
        }
        break;
     case BASIC_SIGNAL:
        fprintf(nodeControlFile, " BASIC-SIGNAL\n");
        signalData = (basicSignal_type *)(network->node[i].controlData);
        fprintf(nodeControlFile, "\tCycle length %d\n", signalData->
           cycleLength);
        for (curMovement = network->node[i].turnMovements->head, curGreen
            = signalData->greenTime->head;
              curMovement != NULL; curMovement = curMovement->next,
                 curGreen = curGreen->next) {
           fprintf(nodeControlFile, "\t%d -> %d -> %d\t%d\t%f\n",
              curMovement->movement->upstreamArc->tail->ID, i+1,
              curMovement->movement->downstreamArc->head->ID, curGreen->
              value, curMovement->movement->saturationFlow * HOURS);
        }
        break;
     case TWO_WAY_STOP:
        fprintf(nodeControlFile, " TWO-WAY-STOP\n");
        stopData = (twoWayStop_type *)(network->node[i].controlData);
        fprintf(nodeControlFile, "\tIntersection saturation flow %f\n",
           stopData->saturationFlow * HOURS);
        fprintf(nodeControlFile, "\tMinimum stop priority %d\n", stopData
           ->minStopPriority);
        for (curMovement = network->node[i].turnMovements->head;
           curMovement != NULL; curMovement = curMovement->next) {
           /* Need to search priority lists to find the right one for
              this node */
           for (curPriority = stopData->priorityList->head; curPriority
              != NULL; curPriority = curPriority->next) {
              priority = curPriority->priorityLevel;
              for (priorityMovement = curPriority->movements->head;
                 priorityMovement != NULL; priorityMovement =
                 priorityMovement->next) {
                 if (priorityMovement->movement == curMovement->movement)
                     goto done; /* Found the right movement */
              }
           }
```

```
         /* This line is only reached if no match was found above.  */
         fatalError("writeNodeControlFile: Movement %d -> %d -> %d is
            not in any priority list!", curMovement->movement->
            upstreamArc->tail->ID, i+1, curMovement->movement->
            downstreamArc->head->ID);
         /* Otherwise, we jumped to here. */
         done:
         fprintf(nodeControlFile, "\t%d -> %d -> %d\t%d\t%f\n",
            curMovement->movement->upstreamArc->tail->ID, i+1,
            curMovement->movement->downstreamArc->head->ID, priority,
            curMovement->movement->saturationFlow * HOURS);
      }
      break;
   case FANCY_SIGNAL:
      fatalError("Fancy signals not yet implemented!");
      break;
   case INTERCHANGE:
      fprintf(nodeControlFile, " INTERCHANGE\n");
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
         fprintf(nodeControlFile, "\t%d -> %d -> %d\t%f\n", curMovement
            ->movement->upstreamArc->tail->ID, i+1, curMovement->
            movement->downstreamArc->head->ID, curMovement->movement->
            saturationFlow * HOURS);
      }
      break;
   case UNKNOWN_CONTROL:
      fprintf(nodeControlFile, " UNKNOWN\n");
   }

}

/*
writeParametersFile -- Writes a parameters file.  Typically used by
   warrant analysis when an "artificial" run must be conducted
Arguments:
   thisRun -- pointer to a parameters_type containing parameters to
      write
   parametersFileName -- name of the parameters file to create
*/
void writeParametersFile(struct parameters_type_s *thisRun, char*
   parametersFileName) {
  FILE *parametersFile = openFile(parametersFileName, "w");

   fprintf(parametersFile, "<NETWORK FILE> %s\n", thisRun->
      networkFileName);
   fprintf(parametersFile, "<DEMAND FILE> %s\n", thisRun->
```

```
      demandFileName);
   fprintf(parametersFile, "<NODE COORDINATE FILE> %s\n", thisRun->
      coordinateFileName);
   fprintf(parametersFile, "<COUNTS FILE> %s\n", thisRun->
      countsFileName);
   fprintf(parametersFile, "<TIME HORIZON> %ld\n", thisRun->timeHorizon
      );
   fprintf(parametersFile, "<TICK LENGTH> %f\n", thisRun->tickLength);
   fprintf(parametersFile, "<LAST VEHICLE ON> %ld\n", thisRun->
      lastVehicleOn);
   fprintf(parametersFile, "<AEC TOLERANCE> %f\n", thisRun->AECtarget);
   fprintf(parametersFile, "<MAX RUN TIME> %f\n", thisRun->maxRunTime);
   fprintf(parametersFile, "<MAX ITERATIONS> %d\n", thisRun->
      maxIterations);
   fprintf(parametersFile, "<DEMAND MULTIPLIER> %f\n", thisRun->
      demandMultiplier);
   fprintf(parametersFile, "<NODE CONTROL FILE> %s\n", thisRun->
      nodeControlFileName);
   fprintf(parametersFile, "<VERBOSITY LEVEL> %hd\n", thisRun->
      verbosity);
   fprintf(parametersFile, "<VEHICLE LENGTH> %f\n", thisRun->
      vehicleLength);
   fprintf(parametersFile, "<BACKWARD WAVE RATIO> %f\n", thisRun->
      backwardWaveRatio);
   fprintf(parametersFile, "<RANDOM SEED> %d\n", thisRun->randomSeed);
   fprintf(parametersFile, "<DEMAND PROFILE> ");
   switch (thisRun->demandProfile) {
   case UNIFORM:
      fprintf(parametersFile, "UNIFORM\n");
      break;
   case PEAK:
      fprintf(parametersFile, "PEAK\n");
      break;
   case TRIANGLE:
      fprintf(parametersFile, "TRIANGLE\n");
      fprintf(parametersFile, "<PEAK DEMAND TIME> %ld", ((
         triangleProfile_type *)(thisRun->profileParameters))->peakTime
         );
      fprintf(parametersFile, "<RATIO 1> %f", ((triangleProfile_type *)
         (thisRun->profileParameters))->ratio1);
      fprintf(parametersFile, "<RATIO 2> %f", ((triangleProfile_type *)
         (thisRun->profileParameters))->ratio2);
      break;
   case QUADRATIC:
      fprintf(parametersFile, "QUADRATIC\n");
      break;
   case RAW:
```

```
      fprintf(parametersFile, "RAW\n");
      break;
   default:
      fatalError("Unknown demand profile type %d when writing parameters file!\n", thisRun->demandProfile);
   }
   fprintf(parametersFile, "<SOLUTION ALGORITHM> ");
   switch (thisRun->solutionAlgorithm) {
   case MSA:
      fprintf(parametersFile, "MSA\n");
      break;
   case LUCE:
      fprintf(parametersFile, "LUCE\n");
      break;
   default:
      fatalError("Unknown solution algorithm %d when writing parameters file!\n", thisRun->solutionAlgorithm);
   }

  fclose(parametersFile);
  displayMessage(FULL_NOTIFICATIONS, "Finished writing parameters file.\n");
}

/**********************************************
 ** Reading control data for different nodes **
 **********************************************/

/*
readTurnMovement -- Reads a line from an intersection control file. Because different control types have different data,
                    this function just serves as a switch to the right parsing routine.
Arguments:
   inputLine -- entire line of the intersection control file to pass to appropriate routine
   node -- pointer to the node this movement belongs to
*/
void readTurnMovement(char *inputLine, node_type *node) {
  switch (node->control) {
  case CENTROID:
    warning(MEDIUM_NOTIFICATIONS, "Centroid %d should not have explicit turn movements listed.\n", node->ID);
    return;
  case NONHOMOGENEOUS:
  case DIVERGE:
  case MERGE:
```

```
  case FOUR_WAY_STOP:
  case INTERCHANGE:
    readBasicTurnMovement(inputLine, node);
    break;
  case TWO_WAY_STOP:
      readTwoWayStopMovement(inputLine, node);
      break;
  case BASIC_SIGNAL:
      readBasicSignalMovement(inputLine, node);
      break;
  case FANCY_SIGNAL:
    fatalError("Intersection type %d not yet implemented!", node->control);
  case UNKNOWN_CONTROL:
      break;
  default:
    fatalError("Unknown intersection type %d for node %d", node->control, node->ID);
  }

}

/*
readBasicTurnMovement -- Reads a turn movement from the "basic" intersection types that do not have fancy formatting
Arguments:
   inputLine -- entire line of the intersection control file to read
   node -- pointer to the node this movement belongs to
*/
void readBasicTurnMovement(char *inputLine, node_type *node) {
   turning_type *newTurningMovement = newScalar(turning_type);
   int numParams;
   int h, i, j;
   float s;

   numParams = sscanf(inputLine, "%d -> %d -> %d %f", &h, &i, &j, &s);
   if (numParams != 4) fatalError("Misformatted control sequence. Current line is:\n%s", inputLine);
   createMovement(newTurningMovement, h, i, j, node);

   if (s < 0) fatalError("Turn movement has negative saturation flow! Current line is:\n%s", inputLine);
   if (s == 0) warning(LOW_NOTIFICATIONS, "Turn movement %d -> %d -> %d has zero saturation flow!", h, i, j);
   newTurningMovement->saturationFlow = s / HOURS;
}
```

```
/*
readBasicSignalMovement -- Reads a turn movement for a signal (looking for cycle length, etc.)
Arguments:
   inputLine -- entire line of the intersection control file to read
   node -- pointer to the node this movement belongs to
*/
void readBasicSignalMovement(char *inputLine, node_type *node) {
   turning_type *newTurningMovement = newScalar(turning_type);
   basicSignal_type *signalData = (basicSignal_type *)(node->controlData);
   int numParams;
   int h, i, j;
   int c, g;
   float s;

   /* Does this line contain the cycle length? */
   if (strstr(inputLine, "Cycle length") != NULL) {
      numParams = sscanf(inputLine, "Cycle length %d", &c);
      if (numParams != 1) fatalError("Misformatted cycle length control sequence in line containing:\n%s", inputLine);
      if (c <= 0) fatalError("Basic signal has non-positive cycle length!  Current line contains:\n%s", inputLine);
      signalData->cycleLength = c;
      return;
   }

   /* If not, it contains a turn movement with green time and saturation flow*/
   numParams = sscanf(inputLine, "%d -> %d -> %d %d %f", &h, &i, &j, &g, &s);
   if (numParams != 5) fatalError("Misformatted control sequence in line containing:\n%s", inputLine);
   createMovement(newTurningMovement, h, i, j, node);

   if (s < 0) fatalError("Turn movement has negative saturation flow! Current line contains:\n%s", inputLine);
   if (s == 0) warning(LOW_NOTIFICATIONS, "Turn movement %d -> %d -> %d has zero saturation flow!", h, i, j);
   newTurningMovement->saturationFlow = s / HOURS;

   if (g <= 0) fatalError("Basic signal turn movement has non-positive green time!  Current line contains:\n%s", inputLine);
   insertLinkedList(signalData->greenTime, g, NULL); /* This insertion has to mirror the insertion in createMovement */
}

```

```c
/*
readTwoWayStopMovement -- Reads a turn movement from the two-way stop intersection type (looking out for special parameters)
Arguments:
   inputLine -- entire line of the intersection control file to read
   node -- pointer to the node this movement belongs to
*/
void readTwoWayStopMovement(char *inputLine, node_type *node) {
   turning_type *newTurningMovement;
   priorityLinkedListElt *curPriority, *prevPriority;
   twoWayStop_type *stopData = (twoWayStop_type *)(node->controlData);
   int numParams;
   int h, i, j, p;
   float s;

   /* Does this line contain an intersection parameter? */
   if        (strstr(inputLine, "Minimum stop priority") != NULL) {
      numParams = sscanf(inputLine, "Minimum stop priority %d", &stopData->minStopPriority);
      if (numParams != 1) fatalError("Misformatted minimum stop priority control sequence in line containing:\n%s", inputLine);
      return;
   } else if (strstr(inputLine, "Intersection saturation flow") != NULL) {
      numParams = sscanf(inputLine, "Intersection saturation flow %f", &s);
      if (numParams != 1) fatalError("Misformatted intersection saturation flow control sequence in line containing:\n%s", inputLine);
      if (s < 0) fatalError("Intersection has negative saturation flow! Current line contains:\n%s", inputLine);
      if (s == 0) warning(LOW_NOTIFICATIONS, "Intersection %d has zero saturation flow!", node->ID);
      stopData->saturationFlow = s / HOURS;
      return;
   }

   /* If not, it contains a turn movement with priority level */
   newTurningMovement = newScalar(turning_type);
   numParams = sscanf(inputLine, "%d -> %d -> %d %d %f", &h, &i, &j, &p, &s);
   if (numParams != 5) fatalError("Misformatted control sequence in line containing:\n%s", inputLine);
   createMovement(newTurningMovement, h, i, j, node);
   if (s < 0) fatalError("Turn movement has negative saturation flow! Current line contains:\n%s", inputLine);
```

```
   if (s == 0) warning(LOW_NOTIFICATIONS, "Turn movement %d -> %d -> %d has zero saturation flow!", h, i, j);
   newTurningMovement->saturationFlow = s / HOURS;

   /* Find appropriate place in priority list */
   prevPriority = NULL;
   for (curPriority = stopData->priorityList->head; curPriority != NULL; curPriority = curPriority->next) {
      if (curPriority->priorityLevel >= p) break;
      prevPriority = curPriority;
   }
   if (curPriority == NULL || curPriority->priorityLevel != p) {
      curPriority = insertPriorityLinkedList(stopData->priorityList, p, prevPriority);
   }
   insertTurningLinkedList(curPriority->movements, newTurningMovement, NULL);
}

/*
createMovement -- Initializes data structures related to a turning_type, with some error checking
Arguments:
   movement -- pointer to the movement being initialized
   upstreamNode -- ID of the upstream intersection for this movement
   curNode -- ID of the intersection containing this movement
   downstreamNode -- ID of the downstream intersection for this movement
   node -- pointer to the node this movement belongs to
*/
void createMovement(turning_type *movement, int upstreamNode, int curNode, int downstreamNode, node_type *node) {
   arcLinkedListElt *curArc;

   if (curNode != node->ID) fatalError("Turn movement %d -> %d -> %d does not match node %d!\n", curNode, node->ID);

   /* Find relevant upstream and downstream arcs, insert turning movement into relevant data structures */
   movement->upstreamArc = NULL;
   movement->vehicles = createVehicleDoublyLinkedList();
   for (curArc = node->reverseStar->head; curArc != NULL; curArc = curArc->next) {
      if (curArc->arc->tail->ID == upstreamNode) {
         movement->upstreamArc = curArc->arc;
         break;
      }
```

```c
   }
   if (movement->upstreamArc == NULL) fatalError("Upstream arc not found for movement %d -> %d -> %d", upstreamNode, curNode, downstreamNode);
   movement->downstreamArc = NULL;
   for (curArc = node->forwardStar->head; curArc != NULL; curArc = curArc->next) {
      if (curArc->arc->head->ID == downstreamNode) {
         movement->downstreamArc = curArc->arc;
         break;
      }
   }
   if (movement->downstreamArc == NULL) fatalError("Downstream arc not found for movement %d -> %d -> %d", upstreamNode, curNode, downstreamNode);

   insertTurningLinkedList(node->turnMovements, movement, NULL);
   insertTurningLinkedList(movement->upstreamArc->turnMovements, movement, NULL);
   insertTurningLinkedList(movement->downstreamArc->upstreamMovements, movement, NULL);
}

/***********************
 ** String processing **
 ***********************/

/*
blankInputString -- Replaces all characters in a string with NULLs
Arguments:
   string -- string to blank
   length -- string length
*/
void blankInputString(char *string, int length) {
  int i;
  for (i = 0; i < length; i++) string[i] = '\0';
}

/*
parseMetadata -- Splits a metadata line into its metadata tag and value. Metadat tags are marked with <> signs; whitespace between tag and value is ignored.
Arguments:
   inputLine -- full line from the input file
   metadataTag -- string to store the metadata tag
   metadataValue -- string to store the metadata value
*/
```

```
int parseMetadata(char* inputLine, char* metadataTag, char* metadataValue) {
  int i = 0, j = 0;
  while (inputLine[i] != '<') {
     if (inputLine[i] == '\0' || inputLine[i] == '\n' || inputLine[i] == '\r') return BLANK_LINE;
     if (inputLine[i] == '~') return COMMENT;
     i++;
  }
  i++;
  while (inputLine[i] != '\0' && inputLine[i] != '>') {
    metadataTag[j++] = toupper(inputLine[i++]);
  }
  metadataTag[j] = '\0';
  if (inputLine[i] == '\0') fatalError("Metadata tag not closed in parameters file - ", metadataTag);
  i++;
  while (inputLine[i] != '\0' && (inputLine[i] == ' ' || inputLine[i] == '\t')) i++;
  j = 0;
  while (inputLine[i] != '\0' && inputLine[i] != '\n' && inputLine[i] != '\r' && inputLine[i] != '~') {
    metadataValue[j++] = inputLine[i++];
  }
  metadataValue[j] = '\0';
  return SUCCESS;
}

/*
parseLine -- Checks for comments and blank lines, and removes leading spaces.  Returns either COMMENT, BLANK_LINE, or SUCCESS based on the line
Arguments:
   inputLine -- full line from the input file
   metadataTag -- string to store the metadata tag
   metadataValue -- string to store the metadata value
*/
int parseLine(char* inputLine, char* outputLine) {
  int i = 0, j = 0;
  while (inputLine[i] != '\0' && (inputLine[i] == ' ' || inputLine[i] == '\t')) i++;
  if (inputLine[i] == '~') return COMMENT;
  if (inputLine[i] == '\0' || inputLine[i] == '\n' || inputLine[i] == '\r') return BLANK_LINE;
  while (inputLine[i] != '\0') {
    outputLine[j++] = inputLine[i++];
  }
```

```
  outputLine[j] = '\0';
  return SUCCESS;
}
```

D.1.6 fileio.h

```
#ifndef _FILEIO_H_
#define _FILEIO_H_

#include <ctype.h>
#include <limits.h>
#include <math.h>
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "dta.h"
#include "network.h"
#include "sampling.h"
#include "utils.h"
#include "vehicle.h"

struct parameters_type_s;

/* Return codes for metadata parsing */
enum {
  SUCCESS,
  BLANK_LINE,
  COMMENT
};

/*********************
 ** File processing **
 *********************/

FILE *openFile(char *filename, char *access);

void readDemandFile(struct parameters_type_s *run);
void readNetworkFile(network_type *network, char *networkFileName,
   float backwardWaveRatio);
void readStaticODFile(network_type *network, char *staticODFileName,
   float *demandMultiplier);
void readNodeCoordinateFile(network_type *network, char *
   coordinateFileName);
void readNodeControlFile(network_type *network, char *
   nodeControlFileName);
void readParametersFile(struct parameters_type_s *thisRun, char*
```

```
parametersFileName);
void readRawODTFile(network_type *network, char *rawODTFileName, float *demandMultiplier);

void displayRunParameters(int minVerbosity, struct parameters_type_s *run);

void writeCumulativeCounts(network_type *network, char *countsFileName);
void readCumulativeCounts(network_type *network, char *countsFileName);
void writeNodeControlFile(network_type *network,  char *nodeControlFileName);
void writeNode(FILE *nodeControlFile, network_type *network, int i);
void writeParametersFile(struct parameters_type_s *thisRun, char* parametersFileName);
void writeLinkSummary(struct parameters_type_s *run, char *nodeSummaryName);
void writeNodeSummary(struct parameters_type_s *run, char *nodeSummaryName);

/*********************************************
 ** Reading control data for different nodes **
 *********************************************/

void readTurnMovement(char *inputLine, node_type *node);
void readBasicTurnMovement(char *inputLine, node_type *node);
void readBasicSignalMovement(char *inputLine, node_type *node);
void readTwoWayStopMovement(char *inputLine, node_type *node);
void createMovement(turning_type *movement, int upstreamNode, int curNode, int downstreamNode, node_type *node);

/**********************
 ** String processing **
 **********************/

void blankInputString(char *string, int length);
int parseMetadata(char* inputLine, char* metadataTag, char* metadataValue);
int parseLine(char* inputLine, char* outputLine);


#endif
```

D.1.7 `node.c`

```
#include "node.h"

```

```
/**** Calculate link parameters ****/

void calculateReceivingFlows(arc_type *arc) {
  cellDoublyLinkedListElt *curCell;

  for (curCell = arc->cells->tail; curCell != NULL; curCell = curCell->next) {
    curCell->cell->receivingFlow = min(arc->waveRatio * (arc->cellMaxVehicles - curCell->cell->vehicles->size), arc->cellCapacity);
  }
  arc->receivingFlow = arc->cells->tail->cell->receivingFlow;
}

void calculateSendingFlows(arc_type *arc) {
  cellDoublyLinkedListElt *curCell;

  for (curCell = arc->cells->tail; curCell != NULL; curCell = curCell->next) {
    curCell->cell->sendingFlow = min(curCell->cell->vehicles->size, arc->cellCapacity);
  }
  arc->sendingFlow = arc->cells->head->cell->sendingFlow;
}

/*********************************
 * PROCESSING SPECIFIC NODE TYPES *
 *********************************/

/*
 These algorithms are used to update the values of capacity and targetDelay before calling the generic intersection processor.
 These node processing algorithms *destroy* the values of sending and receiving flows.  They should not be trusted afterwards.
  */

void processNode(network_type *network, node_type *node, int t) {
  switch (node->control) {
  case CENTROID:
    processCentroidNode(network, node, t);
    return; /* Centroid nodes are treated differently than all other intersection types */
  case NONHOMOGENEOUS: /* Capacity and delay values for these intersection types are constant */
  case DIVERGE:        /* Therefore, there is no need to update their values each iteration   */
  case MERGE:        /* (Initialization routines give the correct
```

```
     values for t = 0          */
  case FOUR_WAY_STOP:
  case INTERCHANGE:
    break;
  case TWO_WAY_STOP:
      processTwoWayStopNode(node);
      break;
  case BASIC_SIGNAL:
      processBasicSignalNode(network, node);
      break;
  case FANCY_SIGNAL:
    fatalError("Intersection type %d not yet implemented!", node->control);
  case UNKNOWN_CONTROL:
  default:
    fatalError("Unknown intersection type %d for node %d", node->control, node->ID);
  }
  processGeneralIntersection(node, t);
}

/*
Processing centroids.
Vehicles have already been loaded to origin movements in the main simulateCTM loop
What remains:
  Move as many vehicles in origin turn movements to downstream link as possible
  Shift all incoming [sending] flows to the destination turn movement
  Move all vehicles in destination turn movements to the destination node.
*/
void processCentroidNode(network_type *network, node_type *node, int t) {
  turningLinkedListElt *curMovement;
  vehicle_type *vehicle;
  turning_type *movement;
  int movingFlow;

  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    movement = curMovement->movement;
    if (movement->upstreamArc == &(network->origin)) {
      /* Move vehicles from origin movement onto downstream links */
      movingFlow = min(movement->vehicles->size, movement->downstreamArc->receivingFlow);
      while (movingFlow-- > 0) {
```

```
        vehicle = movement->vehicles->tail->vehicle;
        transferVehicleToLink(vehicle, movement, movement->downstreamArc, t);
      }
    } else if (movement->downstreamArc == &(network->destination)) {
      /* Move vehicles from upstream link onto movement, and then to destination */
      movingFlow = movement->upstreamArc->sendingFlow;
      while (movingFlow-- > 0) {
        vehicle = movement->upstreamArc->cells->head->cell->vehicles->tail->vehicle;
        transferVehicleToMovement(vehicle, movement->upstreamArc, movement, t);
        transferVehicleToLink(vehicle, movement, movement->downstreamArc, t);
      }
    } else { /* Currently, we don't allow centroids to be "through nodes" */
      fatalError("Node %d is a centroid, but turning movement %d -> %d -> %d neither loads or removes a vehicle.", node->ID, movement->upstreamArc->tail->ID, node->ID, movement->downstreamArc->head->ID);
    }
  }
}

/* Get capacities by doing a forward simulation...
   Set all movement capacities to 0
   Set all movement sending flows to 0
   For each upstream link
      Search through the vehicle DLLE until you have scanned the entire sending flow
      Increase movement sending flow by 1 for each relevant vehicle
   Go in order of priority list
      Iterate over turn movements (when through, go back to start)
         If intersection capacity remains AND movement sending flow positive, deduct movement sending flow, increment movement capacity
Do not change target delay. */
void processTwoWayStopNode(node_type *node) {
   arcLinkedListElt *curArc;
   turning_type *movement;
   turningLinkedListElt *curMovement;
   vehicleDoublyLinkedListElt *curVehicle;
   priorityLinkedListElt *curPriority;
   twoWayStop_type *stopData = (twoWayStop_type *) node->controlData;
   int veh, remainingCapacity;
```

```
    bool finishedPriority;

    /* Step 1. Initialize */
    for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
       curMovement->movement->capacity = 0;
       curMovement->movement->sendingFlow = curMovement->movement->vehicles->size;
    }

    /* Step 2. Classify sending flows by movement */
    for (curArc = node->reverseStar->head; curArc != NULL; curArc = curArc->next) {
       for (curVehicle = curArc->arc->cells->head->cell->vehicles->tail, veh = 0;
            curVehicle != NULL && veh < curArc->arc->sendingFlow;
            curVehicle = curVehicle->prev, veh++) {
          movement = curVehicle->vehicle->curPathPosition->movement;
          movement->sendingFlow++;
          displayMessage(FULL_DEBUG, "Found vehicle %d of %d for %d -> %d -> %d\n", veh+1, curArc->arc->sendingFlow, movement->upstreamArc->tail->ID, movement->upstreamArc->head->ID, movement->downstreamArc->head->ID);
       }
    }

    /* Step 3. Go in order of priority list, allocating capacity as possible */
    remainingCapacity = stopData->capacity;
    for (curPriority = stopData->priorityList->head; curPriority != NULL; curPriority = curPriority->next) {
       finishedPriority = FALSE;
       while (finishedPriority == FALSE) {
         displayMessage(FULL_DEBUG, "Processing priority level %d; remaining capacity is %d\n", curPriority->priorityLevel, remainingCapacity);
          finishedPriority = TRUE;
          for (curMovement = curPriority->movements->head; curMovement != NULL; curMovement = curMovement->next) {
           displayMessage(FULL_DEBUG, "Considering movement %d -> %d -> %d with sending flow %d\n", curMovement->movement->upstreamArc->tail->ID, curMovement->movement->upstreamArc->head->ID, curMovement->movement->downstreamArc->head->ID, curMovement->movement->sendingFlow);
             if (curMovement->movement->sendingFlow == 0) continue;
             if (remainingCapacity == 0) return; /* No more capacity left, so we are done */
```

```
            curMovement->movement->sendingFlow--;
            curMovement->movement->capacity++;
            remainingCapacity--;
            displayMessage(FULL_DEBUG, "Increment capacity for %d -> %d -> %d\n", curMovement->movement->upstreamArc->tail->ID, curMovement->movement->upstreamArc->head->ID, curMovement->movement->downstreamArc->head->ID);
            finishedPriority = FALSE;
         }
      }
   }
}

/* Get v/c by comparing vehicles in the turning movement to capacity [will lag by one, but shouldn't matter too much] and delay.  Do not change capacities */
void processBasicSignalNode(network_type *network, node_type *node) {
   turningLinkedListElt *curMovement;
   linkedListElt *curGreenTime;
   basicSignal_type *signalData = (basicSignal_type *) node->controlData;
   float degreeOfSaturation;
   float greenFraction;

   for (curMovement = node->turnMovements->head, curGreenTime = signalData->greenTime->head;
        curMovement != NULL && curGreenTime != NULL;
        curMovement = curMovement->next, curGreenTime = curGreenTime->next) {
      degreeOfSaturation = ((float) curMovement->movement->vehicles->size) / curMovement->movement->capacity;
      greenFraction = ((float) curGreenTime->value) / signalData->cycleLength;
      curMovement->movement->targetDelay = 0.5 * signalData->cycleLength * (1 - greenFraction) / (1 - min(1, degreeOfSaturation) * greenFraction);
      curMovement->movement->targetDelay /= network->tickLength;
    displayMessage(FULL_DEBUG, "Set movement %d -> %d -> %d target to delay %d based on degree of saturation %f and green fraction %f\n", curMovement->movement->upstreamArc->tail->ID, node->ID, curMovement->movement->downstreamArc->head->ID, curMovement->movement->targetDelay, degreeOfSaturation, greenFraction);
   }
}

/********************************
 * CORE INTERSECTION PROCESSING *
```

```
********************************/

void processGeneralIntersection(node_type *node, int t) {
  turningLinkedListElt *curMovement;
  arcLinkedListElt *downstreamArc;
  int pastTime;

  if (verbosity >= FULL_DEBUG) {
    displayMessage(FULL_DEBUG, "Processing node %d at time %d\n", node
        ->ID, t);
    displayMessage(FULL_DEBUG, "Sending flows:\n", node->ID, t);
    for (downstreamArc = node->reverseStar->head; downstreamArc != NULL
        ; downstreamArc = downstreamArc->next) {
      displayMessage(FULL_DEBUG, "(%d,%d)\t%d\n", downstreamArc->arc->
          tail->ID, node->ID, downstreamArc->arc->sendingFlow);
    }
    displayMessage(FULL_DEBUG, "Receiving flows:\n", node->ID, t);
    for (downstreamArc = node->forwardStar->head; downstreamArc != NULL
        ; downstreamArc = downstreamArc->next) {
      displayMessage(FULL_DEBUG, "(%d,%d)\t%d\n", node->ID,
          downstreamArc->arc->head->ID, downstreamArc->arc->
          receivingFlow);
    }

  }

  /* Phase I. Move vehicles from upstream links into turn movements */
  moveVehiclesIntoIntersectionMovements(node, t);

  /* Phase II. Recalculate necessary values and turn movement sending
     flows */
  for (downstreamArc = node->forwardStar->head; downstreamArc != NULL;
     downstreamArc = downstreamArc->next) {
    downstreamArc->arc->receivingFlow = downstreamArc->arc->cells->tail
        ->cell->receivingFlow;
  }
  for (curMovement = node->turnMovements->head; curMovement != NULL;
     curMovement = curMovement->next) {
    pastTime = t - curMovement->movement->targetDelay;
    pastTime = max(pastTime, 0);
    curMovement->movement->sendingFlow = curMovement->movement->
        upstreamCount[pastTime] - curMovement->movement->downstreamCount
        [t];
  }

  /* Phase III. Move vehicles from turn movements onto downstream links
      */
```

```
  for (downstreamArc = node->forwardStar->head; downstreamArc != NULL; downstreamArc = downstreamArc->next) {
    processMergeFlows(downstreamArc->arc->upstreamMovements, t);
  }
}


void moveVehiclesIntoIntersectionMovements(node_type *node, int t) {
  turningLinkedListElt *curMovement;
  vehicleDoublyLinkedListElt *curVehicle;
  arcLinkedListElt *upstreamArc, *downstreamArc;
  cell_type *upstreamCell;
  turning_type *movement;
  int veh;

  /* 1. Initialize and calculate movement sending flows from arc sending flows */
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->flow = 0;
    curMovement->movement->sendingFlow = curMovement->movement->vehicles->size;
  }
  for (upstreamArc = node->reverseStar->head; upstreamArc != NULL; upstreamArc = upstreamArc->next) {
    upstreamCell = upstreamArc->arc->cells->head->cell;
    for (veh = 0, curVehicle = upstreamCell->vehicles->tail; curVehicle != NULL && veh < upstreamArc->arc->sendingFlow; veh++, curVehicle = curVehicle->prev) {
      movement = curVehicle->vehicle->curPathPosition->movement;
      movement->sendingFlow++;
    }
  }
  displayMessage(FULL_DEBUG, "Movement sending flows and capacities [size]:\n");
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->sendingFlow = min(curMovement->movement->sendingFlow, curMovement->movement->capacity);
    displayMessage(FULL_DEBUG, "%d -> %d -> %d\t%d\t%d\t[%d]\n", curMovement->movement->upstreamArc->tail->ID, node->ID, curMovement->movement->downstreamArc->head->ID, curMovement->movement->sendingFlow, curMovement->movement->capacity, curMovement->movement->vehicles->size);
  }

  /* 2. Calculate each turning movement's receiving flow */
```

```
  for (downstreamArc = node->forwardStar->head; downstreamArc != NULL; downstreamArc = downstreamArc->next) {
    calculateMergeFlows(downstreamArc->arc->upstreamMovements);
  }
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->receivingFlow = curMovement->movement->flow;
  }

  if (verbosity >= FULL_DEBUG) {
    displayMessage(FULL_DEBUG, "Calculated movement receiving flows:\n");
    for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
      displayMessage(FULL_DEBUG, "%d -> %d -> %d\t%d\n", curMovement->movement->upstreamArc->tail->ID, node->ID, curMovement->movement->downstreamArc->head->ID, curMovement->movement->receivingFlow);
    }
  }



  /* 3. Load vehicles onto movements */
  for (upstreamArc = node->reverseStar->head; upstreamArc != NULL; upstreamArc = upstreamArc->next) {
    processDivergeFlows(upstreamArc->arc->turnMovements, t);
  }
}

/* Moves vehicles from an upstream link onto turning movements in a diverge-like fashion */
int processDivergeFlows(turningLinkedList *divergeMovements, int t) {
   arc_type *upstreamArc = divergeMovements->head->movement->upstreamArc;
   cell_type *upstreamCell = upstreamArc->cells->head->cell;
   vehicle_type *vehicle;
   turning_type *movement;
   int movingFlow = upstreamArc->sendingFlow;
   int vehiclesMoved = 0;

   while (movingFlow-- > 0) {
      vehicle = upstreamCell->vehicles->tail->vehicle;
      movement = vehicle->curPathPosition->movement;
      if (movement->receivingFlow > 0) { /* Load vehicle onto movement */
         transferVehicleToMovement(vehicle, movement->upstreamArc,
```

```
          movement, t);
        vehiclesMoved++;
        upstreamArc->sendingFlow--;
        movement->receivingFlow--;
        displayMessage(FULL_DEBUG, "Moved vehicle onto movement %d -> %d -> %d\n", movement->upstreamArc->tail->ID, movement->upstreamArc->head->ID, movement->downstreamArc->head->ID);
      } else { /* Downstream receiving flow limits vehicle transfer; cut off all other flow*/
        break;
      }
    }

    return vehiclesMoved;
}

/* Moves vehicles from turning movements onto a downstream link in a merge-like fashion */
int processMergeFlows(turningLinkedList *mergeMovements, int t) {
  int ij, vehiclesMoved = 0;
  arc_type *downstreamArc = mergeMovements->head->movement->downstreamArc;
  turningLinkedListElt *curMovement;
  vehicle_type *vehicle;

  calculateMergeFlows(mergeMovements);
  for (ij = 0, curMovement = mergeMovements->head; curMovement != NULL; ij++, curMovement = curMovement->next) {
    while (curMovement->movement->flow-- > 0) {
      vehicle = curMovement->movement->vehicles->tail->vehicle;
        transferVehicleToLink(vehicle, curMovement->movement, downstreamArc, t);
        vehiclesMoved++;
        displayMessage(FULL_DEBUG, "Moved vehicle from movement %d -> %d -> %d\n", curMovement->movement->upstreamArc->tail->ID, curMovement->movement->upstreamArc->head->ID, curMovement->movement->downstreamArc->head->ID);
    }
  }
  return vehiclesMoved;
}

void calculateMergeFlows(turningLinkedList *mergeMovements) {
  int ij, totalCapacity;
  float remainingFlow;
  arc_type *downstreamArc = mergeMovements->head->movement->downstreamArc;
```

```
turningLinkedListElt *curMovement;
declareMatrix(float, mergeFlows, mergeMovements->size, 1);

for (ij = 0, curMovement = mergeMovements->head; curMovement != NULL;
    ij++, curMovement = curMovement->next) {
  mergeFlows[ij][0] = 0;
}

do {
  totalCapacity = 0;
  remainingFlow = downstreamArc->receivingFlow;
  for (ij = 0, curMovement = mergeMovements->head; curMovement !=
     NULL; ij++, curMovement = curMovement->next) {
    if (curMovement->movement->sendingFlow > mergeFlows[ij][0])
       totalCapacity += curMovement->movement->capacity;
    remainingFlow -= mergeFlows[ij][0];
  }
  if (totalCapacity == 0 || remainingFlow < MOVING_FLOW_EPSILON)
     break; /* No additional vehicles can move */
  for (ij = 0, curMovement = mergeMovements->head; curMovement !=
     NULL; ij++, curMovement = curMovement->next) {
    mergeFlows[ij][0] = min(curMovement->movement->sendingFlow,
       mergeFlows[ij][0] + remainingFlow * curMovement->movement->
       capacity / totalCapacity);
  }
} while (remainingFlow > 0); /* Loop while receiving flow remains */

if (verbosity >= FULL_DEBUG) {
  displayMessage(FULL_DEBUG, "Unrounded merge flows:\n");
  for (ij = 0, curMovement = mergeMovements->head; curMovement !=
     NULL; ij++, curMovement = curMovement->next) {
    displayMessage(FULL_DEBUG, "%d -> %d -> %d\t%f\n", curMovement->
       movement->upstreamArc->tail->ID, curMovement->movement->
       upstreamArc->head->ID, curMovement->movement->downstreamArc->
       head->ID, mergeFlows[ij][0]);
  }
}

roundStochasticMatrix(mergeFlows, mergeMovements->size, 1,
   MERGE_PRECISION);

displayMessage(FULL_DEBUG, "Rounded merge flows:\n");
for (ij = 0, curMovement = mergeMovements->head; curMovement != NULL;
    ij++, curMovement = curMovement->next) {
  curMovement->movement->flow = mergeFlows[ij][0];
  displayMessage(FULL_DEBUG, "%d -> %d -> %d\t%d\n", curMovement->
     movement->upstreamArc->tail->ID, curMovement->movement->
```

```
        upstreamArc->head->ID, curMovement->movement->downstreamArc->
        head->ID, curMovement->movement->flow);
  }

  deleteMatrix(mergeFlows, mergeMovements->size);
}

/**********************
 * NODE INITIALIZATION *
 **********************/

void initializeNodes(network_type *network) {
  int i;
  for (i = 0; i < network->numNodes; i++) {
    switch (network->node[i].control) {
    case CENTROID:
      continue; /* No need to initialize centroids */
    case NONHOMOGENEOUS:
      initializeNonhomogeneousNode(network, &network->node[i]);
      break;
    case DIVERGE:
      initializeDivergeNode(network, &network->node[i]);
      break;
    case MERGE:
      initializeMergeNode(network, &network->node[i]);
      break;
    case FOUR_WAY_STOP:
      initializeFourWayStopNode(network, &network->node[i]);
      break;
    case TWO_WAY_STOP:
        initializeTwoWayStopNode(network, &network->node[i]);
        break;
    case BASIC_SIGNAL:
        initializeBasicSignalNode(network, &network->node[i]);
        break;
    case INTERCHANGE:
        initializeInterchangeNode(network, &network->node[i]);
        break;
    case FANCY_SIGNAL:
      fatalError("Intersection type %d not yet implemented!", network->
        node[i].control);
    case UNKNOWN_CONTROL:
    default:
      fatalError("Unknown intersection type %d for node %d", network->
        node[i].control, network->node[i].ID);
    }
  }
```

```
}

void initializeNonhomogeneousNode(network_type *network, node_type *
    node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL;
      curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->
        saturationFlow * network->tickLength);
    curMovement->movement->targetDelay = 0;
  }
}

void initializeDivergeNode(network_type *network, node_type *node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL;
      curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->
        saturationFlow * network->tickLength);
    curMovement->movement->targetDelay = 0;
  }
}

void initializeMergeNode(network_type *network, node_type *node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL;
      curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->
        saturationFlow * network->tickLength);
    curMovement->movement->targetDelay = 0;
  }
}

void initializeTwoWayStopNode(network_type *network, node_type *node) {
   turningLinkedListElt *curMovement;
   priorityLinkedListElt *curPriority;
   twoWayStop_type *stopData = (twoWayStop_type *)(node->controlData);
   stopData->capacity = ceil(stopData->saturationFlow * network->
      tickLength);
   for (curPriority = stopData->priorityList->head; curPriority != NULL
      ; curPriority = curPriority->next) {
      for (curMovement = curPriority->movements->head; curMovement !=
         NULL; curMovement = curMovement->next) {
         curMovement->movement->targetDelay = (curPriority->
            priorityLevel < stopData->minStopPriority) ? 0 : ceil(
            STOP_DELAY / network->tickLength);
         curMovement->movement->capacity = ceil(curMovement->movement->
```

```
            saturationFlow * network->tickLength);
      }
    }
}

void initializeFourWayStopNode(network_type *network, node_type *node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->saturationFlow * network->tickLength);
    curMovement->movement->targetDelay = ceil(STOP_DELAY / network->tickLength);
  }
}

void initializeBasicSignalNode(network_type *network, node_type *node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->saturationFlow * network->tickLength);
  }
}

void initializeInterchangeNode(network_type *network, node_type *node) {
  turningLinkedListElt *curMovement;
  for (curMovement = node->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    curMovement->movement->capacity = ceil(curMovement->movement->saturationFlow * network->tickLength);
    curMovement->movement->targetDelay = 0;
  }
}

/**** Intersection data structures ****/

priorityLinkedList *createPriorityLinkedList() {
  declareScalar(priorityLinkedList, newll);
  newll->head = NULL;
  newll->tail = NULL;
  newll->size = 0;
  return newll;
}
```

```c

priorityLinkedListElt *insertPriorityLinkedList(priorityLinkedList *list, int priorityLevel, priorityLinkedListElt *after) {
  declareScalar(priorityLinkedListElt, newNode);
  newNode->priorityLevel = priorityLevel;
  newNode->movements = createTurningLinkedList();
  if (after != NULL) { /* Not inserting at head */
    newNode->next = after->next;
    if (list->tail == after) list->tail = newNode;
    after->next = newNode;
  } else { /* Inserting at head */
    newNode->next = list->head;
    if (list->tail == after) list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}

void deletePriorityLinkedList(priorityLinkedList *list) {
  priorityLinkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
    savenode = curnode->next;
    deleteTurningLinkedList(curnode->movements);
    killScalar(curnode);
    curnode = savenode;
  }
  killScalar(list);
}
```

D.1.8 node.h

```c
#ifndef _NODE_H_
#define _NODE_H_

#include <limits.h>
#include "network.h"
#include "sampling.h"
#include "vehicle.h"

/* Consider a "merge" calculation converged if 99% of flow has been assigned (should be enough given rounding) */
#define MOVING_FLOW_EPSILON 0.01
/* Bits of precision to use for stochastic rounding.  Experiment to find good default value */
#define MERGE_PRECISION 3
/* Minimum floating-point receiving flow to allow a vehicle to enter a
```

```c
    link */
#define MIN_INTERSECTION_RECEIVING_FLOW 0.99

/* Seconds of delay at a stop sign (in the absence of conflicting flows ) */
#define STOP_DELAY 4

/**** Intersection-specific structs ****/

typedef struct basicSignal_type_s {
    int cycleLength;
    linkedList *greenTime; /* Contains each phase's green time in the same order as the node's turningLinkedList */
} basicSignal_type;

typedef struct twoWayStop_type_s {
    int minStopPriority; /* Lowest priority set of movements which must stop at the sign */
    float saturationFlow; /* Maximum saturation flow *for the entire intersection* in natural units */
    int capacity; /* Maximum vehicles that can flow through intersection in a simulation tick */
    struct priorityLinkedList_s *priorityList; /* Linked list for the number of priority levels, itself linking to a list of movements for that priority level */
} twoWayStop_type;



/**** Calculate link parameters ****/

void calculateReceivingFlows(arc_type *arc);
void calculateSendingFlows(arc_type *arc);

/**** Processing specific node types ****/

void processNode(network_type *network, node_type *node, int t);
void processCentroidNode(network_type *network, node_type *node, int t);
void processTwoWayStopNode(node_type *node);
void processBasicSignalNode(network_type *network, node_type *node);

/**** Core intersection processing ****/

void processGeneralIntersection(node_type *node, int t);
void moveVehiclesIntoIntersectionMovements(node_type *node, int t);
int processDivergeFlows(turningLinkedList *divergeMovements, int t);
```

```c
int processMergeFlows(turningLinkedList *mergeMovements, int t);
void calculateMergeFlows(turningLinkedList *mergeMovements);

/**** Node initialization ****/

void initializeNodes(network_type *network);
void initializeNonhomogeneousNode(network_type *network, node_type *node);
void initializeDivergeNode(network_type *network, node_type *node);
void initializeMergeNode(network_type *network, node_type *node);
void initializeTwoWayStopNode(network_type *network, node_type *node);
void initializeFourWayStopNode(network_type *network, node_type *node);
void initializeBasicSignalNode(network_type *network, node_type *node);
void initializeInterchangeNode(network_type *network, node_type *node);

/**** Intersection data structures ****/

typedef struct priorityLinkedListElt_s {
  int priorityLevel;
  struct turningLinkedList_s *movements;
  struct priorityLinkedListElt_s *next;
} priorityLinkedListElt;

typedef struct priorityLinkedList_s {
  priorityLinkedListElt *head;
  priorityLinkedListElt *tail;
  long size;
} priorityLinkedList;

priorityLinkedList *createPriorityLinkedList();
priorityLinkedListElt *insertPriorityLinkedList(priorityLinkedList *list, int priorityLevel, priorityLinkedListElt *after);
void deletePriorityLinkedList(priorityLinkedList *list);

#endif
```

D.1.9 `vehicle.c`

```c
#include "vehicle.h"

/**** Link motion ****/

void moveIntralinkVehicles(arc_type *arc) {
  int movingFlow;
  cellDoublyLinkedListElt *curCell;

  /* If there are [1, 2, ..., n] cells, shift 1->2, 2->3, ..., (n-1)->n
```

```
    */
  for (curCell = arc->cells->tail; curCell != NULL; curCell = curCell->next) {
    if (curCell->next == NULL) break; /* Avoid OB1 error */
    movingFlow = min(curCell->cell->sendingFlow, curCell->next->cell->receivingFlow);
    while (movingFlow-- > 0) advanceVehicle(curCell);
  }
}


/* Advances a vehicle to adjacent cell *within a link*. */
void advanceVehicle(cellDoublyLinkedListElt *curCell) {
  vehicle_type *vehicle = curCell->cell->vehicles->tail->vehicle;
  deleteVehicleDoublyLinkedListElt(vehicle->list, vehicle->listElt);
  vehicle->list = curCell->next->cell->vehicles;
  vehicle->listElt = insertVehicleDoublyLinkedList(vehicle->list, vehicle, NULL);
}

/*
Note that these transfer codes require that a vehicle's movement be LINK - MOVEMENT - LINK - MOVEMENT - LINK, etc.
Artificial origin and destination links help with this
Shifts vehicle from its current turn movement to the *upstream* end of a link
*/
void transferVehicleToLink(vehicle_type *vehicle, turning_type *fromMovement, arc_type *toArc, int t) {
  fromMovement->downstreamCount[t]++;
  deleteVehicleDoublyLinkedListElt(vehicle->list, vehicle->listElt);
  vehicle->list = toArc->cells->tail->cell->vehicles;
  vehicle->listElt = insertVehicleDoublyLinkedList(vehicle->list, vehicle, NULL);
  toArc->upstreamCount[t]++;
  vehicle->curPathPosition = vehicle->curPathPosition->next;
}

/* Shifts vehicle from its current link to the *upstream* end of a turning movement */
void transferVehicleToMovement(vehicle_type *vehicle, arc_type *fromArc, turning_type *toMovement, int t) {
  fromArc->downstreamCount[t]++;
  deleteVehicleDoublyLinkedListElt(vehicle->list, vehicle->listElt);
  vehicle->list = toMovement->vehicles;
  vehicle->listElt = insertVehicleDoublyLinkedList(vehicle->list, vehicle, NULL);
```

```
  toMovement->upstreamCount[t]++;
}


float averageExcessCost(network_type *network) {
  displayMessage(DEBUG, "Calculating AEC based on TSTT %ld and SPTT %ld (# vehicles %ld)\n", totalSystemTravelTime(network), shortestPathTravelTime(network), network->numVehicles);
  return (totalSystemTravelTime(network) - shortestPathTravelTime(network)) / (float) network->numVehicles;
}

/*
Calculates total travel time if everyone were to be loaded on shortest paths
*Assumes that travel times have not changed since last call to TDSP, i.e. that the shortest path is included in the network pathset
*/
long shortestPathTravelTime(network_type *network) {
  long SPTT = 0;
  pathLinkedListElt *curPath;
  int ODTminCost;
  long odt;

  for (odt = 0; odt < network->numODTs; odt++) {
    ODTminCost = network->timeHorizon + 1;
    for (curPath = network->ODT[odt].paths->head; curPath != NULL; curPath = curPath->next) {
      ODTminCost = min(ODTminCost, curPath->path->travelTime);
    }
    SPTT += ODTminCost * network->ODT[odt].demand;
  }

  return SPTT * network->tickLength;
}

/* Calculates total travel time using actual paths. */
long totalSystemTravelTime(network_type *network) {
  long odt, TSTT = 0;
  vehicleDoublyLinkedListElt *curVehicle;

  for (odt = 0; odt < network->numODTs; odt++) {
    for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
      TSTT += curVehicle->vehicle->path->travelTime;
    }
  }
```

```c

  return TSTT * network->tickLength;
}

int latestArrivalTime(network_type *network) {
  long t, odt, latestArrival = 0;
  vehicleDoublyLinkedListElt *curVehicle;

  for (odt = 0; odt < network->numODTs; odt++) {
    t = network->ODT[odt].departureTime;
    for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
      latestArrival = max(latestArrival, t + curVehicle->vehicle->path->travelTime);
    }
  }
  return latestArrival * network->tickLength;
}

/* Generate vehicles and data structures after ODTs are complete */
void generateVehicles(network_type *network) {
  int odt, veh;
  vehicle_type *newVehicle;

  displayMessage(FULL_NOTIFICATIONS, "Generating vehicles...");

  for (odt = 0; odt < network->numODTs; odt++) {
    for (veh = 0; veh < network->ODT[odt].demand; veh++) {
      newVehicle = newScalar(vehicle_type);
      newVehicle->path = NULL;
      newVehicle->ODT = &(network->ODT[odt]);
      newVehicle->list = network->origin.cells->head->cell->vehicles;
      newVehicle->listElt = insertVehicleDoublyLinkedList(network->origin.cells->head->cell->vehicles, newVehicle, network->origin.cells->head->cell->vehicles->tail);
      insertVehicleDoublyLinkedList(network->ODT[odt].vehicles, newVehicle, network->ODT[odt].vehicles->tail);
    }
  }

  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

void initializeVehicles(network_type *network) {
   int odt;
   vehicleDoublyLinkedListElt *curVehicle;
   path_type *initialPath;
```

```c

    for (odt = 0; odt < network->numODTs; odt++) {
     initialPath = network->ODT[odt].paths->head->path;
     if (initialPath == NULL) fatalError("No paths available for ODT %d -> %d @ %d", network->ODT[odt].origin->ID, network->ODT[odt].destination->ID, network->ODT[odt].departureTime);
       for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
          curVehicle->vehicle->path = initialPath;
       curVehicle->vehicle->curPathPosition = initialPath->turnMovements->head;
       }
    }
}

void prepareAllTrips(network_type *network) {
    int odt;
    vehicleDoublyLinkedListElt *curVehicle;

    for (odt = 0; odt < network->numODTs; odt++) {
       for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
          curVehicle->vehicle->listElt = insertVehicleDoublyLinkedList(network->origin.cells->head->cell->vehicles, curVehicle->vehicle, network->origin.cells->head->cell->vehicles->tail);
          curVehicle->vehicle->list = network->origin.cells->head->cell->vehicles;
          curVehicle->vehicle->curPathPosition = curVehicle->vehicle->path->turnMovements->head;
       }
    }

}

void terminateAllTrips(network_type *network) {
    int odt;
    int remainingVehicles = 0;
    vehicleDoublyLinkedListElt *curVehicle;

    for (odt = 0; odt < network->numODTs; odt++) {
       for (curVehicle = network->ODT[odt].vehicles->head; curVehicle != NULL; curVehicle = curVehicle->next) {
         if (curVehicle->vehicle->list != network->destination.cells->head->cell->vehicles) remainingVehicles++;
          deleteVehicleDoublyLinkedListElt(curVehicle->vehicle->list, curVehicle->vehicle->listElt);
```

```
        curVehicle->vehicle->listElt = insertVehicleDoublyLinkedList(network->destination.cells->head->cell->vehicles, curVehicle->vehicle, network->destination.cells->head->cell->vehicles->tail);
        curVehicle->vehicle->list = network->destination.cells->head->cell->vehicles;
      }
    }

  if (remainingVehicles > 0) warning(MEDIUM_NOTIFICATIONS, "Simulation ended before %d vehicles left network; consider increasing time horizon.\n", remainingVehicles);
}

/**** Vehicle doubly linked lists ****/

vehicleDoublyLinkedList *createVehicleDoublyLinkedList() {
  declareScalar(vehicleDoublyLinkedList, newdll);
  newdll->head = NULL;
  newdll->tail = NULL;
  newdll->size = 0;
  return newdll;
}

vehicleDoublyLinkedListElt *insertVehicleDoublyLinkedList(vehicleDoublyLinkedList *list, vehicle_type *value, vehicleDoublyLinkedListElt *after) {
  declareScalar(vehicleDoublyLinkedListElt, newNode);
  newNode->vehicle = value;
  if (after != NULL) {
    newNode->prev = after;
    newNode->next = after->next;
    if (list->tail != after) newNode->next->prev = newNode; else list->tail = newNode;
    after->next = newNode;
  } else {
    newNode->prev = NULL;
    newNode->next = list->head;
    if (list->tail != after) newNode->next->prev = newNode; else list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}

void deleteVehicleDoublyLinkedList(vehicleDoublyLinkedList *list) {
```

```c
  while (list->head != NULL)
    deleteVehicleDoublyLinkedListElt(list, list->tail);
  killScalar(list);
}

void deleteVehicleDoublyLinkedListElt(vehicleDoublyLinkedList *list, vehicleDoublyLinkedListElt *elt) {
  if (list->tail != elt) {
    if (list->head != elt) elt->prev->next = elt->next; else list->head = elt->next;
    elt->next->prev = elt->prev;
  } else {
    list->tail = elt->prev;
    if (list->head != elt) elt->prev->next = elt->next; else list->head = elt->next;
  }
  list->size--;
  killScalar(elt);
}

void displayVehicleDoublyLinkedList(int minVerbosity, vehicleDoublyLinkedList *list) {
  vehicleDoublyLinkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p %p %p %p\n", (void *)curnode, curnode->vehicle, (void *)curnode->prev, (void *)curnode->next);
    curnode = (*curnode).next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->tail);
}
```

D.1.10 `vehicle.h`

```c
#ifndef _VEHICLE_H_
#define _VEHICLE_H_

#include "cell.h"
#include "datastructures.h"
#include "network.h"
#include "utils.h"
#include "vehicle.h"

typedef struct vehicle_type_s {
  path_type *path;
```

```
  turningLinkedListElt *curPathPosition; /* A pointer to the path's turning movement list for next location */
  ODT_type  *ODT;
  /* Pointers to list vehicle is stored in (makes for fast moving)
     Vehicles are *always* a member of a vehicleDoublyLinkedList, either waitingVehicles, arrivedVehicles,
     or a list associated with a link or turning movement */
  struct vehicleDoublyLinkedList_s    *list;
  struct vehicleDoublyLinkedListElt_s *listElt;
} vehicle_type;


void moveIntralinkVehicles(arc_type *arc);
void advanceVehicle(cellDoublyLinkedListElt *curCell);
void transferVehicleToLink(vehicle_type *vehicle, turning_type *fromMovement, arc_type *toArc, int t); /* Shifts vehicle from current location to an arc */
void transferVehicleToMovement(vehicle_type *vehicle, arc_type *fromArc, turning_type *toMovement, int t); /* Shifts vehicle from current location to a movement */

/**** Vehicle gap functions ****/

float averageExcessCost(network_type *network);
long shortestPathTravelTime(network_type *network);
long totalSystemTravelTime(network_type *network);
int latestArrivalTime(network_type *network);

/**** Establish vehicle data structures ****/

void generateVehicles(network_type *network);
void initializeVehicles(network_type *network);
void prepareAllTrips(network_type *network);
void terminateAllTrips(network_type *network);

/**** Vehicle doubly linked lists ****/

typedef struct vehicleDoublyLinkedListElt_s {
  vehicle_type *vehicle;
  struct vehicleDoublyLinkedListElt_s *next;
  struct vehicleDoublyLinkedListElt_s *prev;
} vehicleDoublyLinkedListElt;

typedef struct vehicleDoublyLinkedList_s {
  vehicleDoublyLinkedListElt *head;
  vehicleDoublyLinkedListElt *tail;
  long size;
```

```c
} vehicleDoublyLinkedList;

vehicleDoublyLinkedList *createVehicleDoublyLinkedList();
vehicleDoublyLinkedListElt *insertVehicleDoublyLinkedList(vehicleDoublyLinkedList *list, vehicle_type *value, vehicleDoublyLinkedListElt *after);
void deleteVehicleDoublyLinkedList(vehicleDoublyLinkedList *list);
void deleteVehicleDoublyLinkedListElt(vehicleDoublyLinkedList *list, vehicleDoublyLinkedListElt *elt);
void displayVehicleDoublyLinkedList(int minVerbosity, vehicleDoublyLinkedList *list);


#endif
```

D.1.11 `cell.c`

```c
#include "cell.h"

/**** Cell doubly linked lists ****/


/*
These functions mirror those in datastructures.h, but for cell doubly linked lists.
*/

cellDoublyLinkedList *createCellDoublyLinkedList() {
  declareScalar(cellDoublyLinkedList, newdll);
  newdll->head = NULL;
  newdll->tail = NULL;
  newdll->size = 0;
  return newdll;
}

cellDoublyLinkedListElt *insertCellDoublyLinkedList(cellDoublyLinkedList *list, cell_type *value, cellDoublyLinkedListElt *after) {
  declareScalar(cellDoublyLinkedListElt, newNode);
  newNode->cell = value;
  if (after != NULL) {
    newNode->prev = after;
    newNode->next = after->next;
    if (list->head != after) newNode->next->prev = newNode; else list->head = newNode;
    after->next = newNode;
  } else {
```

```c
    newNode->prev = NULL;
    newNode->next = list->head;
    if (list->head != after) newNode->next->prev = newNode; else list->head= newNode;
    list->tail = newNode;
  }
  list->size++;
  return newNode;
}

void deleteCellDoublyLinkedList(cellDoublyLinkedList *list) {
  while (list->tail != NULL)
    deleteCellDoublyLinkedListElt(list, list->head);
  killScalar(list);
}

void deleteCellDoublyLinkedListElt(cellDoublyLinkedList *list, cellDoublyLinkedListElt *elt) {
  if (list->head != elt) {
    if (list->tail != elt) elt->prev->next = elt->next; else list->tail = elt->next;
    elt->next->prev = elt->prev;
  } else {
    list->head = elt->prev;
    if (list->tail != elt) elt->prev->next = elt->next; else list->tail = elt->next;
  }
  list->size--;
  killScalar(elt);
}

void displayCellDoublyLinkedList(int minVerbosity, cellDoublyLinkedList *list) {
  cellDoublyLinkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p %p %p %p\n", (void *)curnode, curnode->cell, (void *)curnode->prev, (void *)curnode->next);
    curnode = (*curnode).next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->tail);
}
```

D.1.12 `cell.h`

```c
#ifndef _CELL_H_
#define _CELL_H_

#include <math.h>
#include "datastructures.h"
#include "limits.h"
#include "network.h"
#include "utils.h"

struct vehicleDoublyLinkedList_s;

typedef struct cell_type_s {
  struct vehicleDoublyLinkedList_s *vehicles;
  arc_type *parentLink;
  int sendingFlow;
  int receivingFlow;
} cell_type;


/**** Cell doubly linked lists ****/

typedef struct cellDoublyLinkedListElt_s {
  cell_type *cell;
  struct cellDoublyLinkedListElt_s *next;
  struct cellDoublyLinkedListElt_s *prev;
} cellDoublyLinkedListElt;

typedef struct cellDoublyLinkedList_s {
  cellDoublyLinkedListElt *head;
  cellDoublyLinkedListElt *tail;
  long size;
} cellDoublyLinkedList;

cellDoublyLinkedList *createCellDoublyLinkedList();
cellDoublyLinkedListElt *insertCellDoublyLinkedList(cellDoublyLinkedList *list, cell_type *cell, cellDoublyLinkedListElt *after);
void deleteCellDoublyLinkedList(cellDoublyLinkedList *list);
void deleteCellDoublyLinkedListElt(cellDoublyLinkedList *list, cellDoublyLinkedListElt *elt);
void displayCellDoublyLinkedList(int minVerbosity, cellDoublyLinkedList *list);

#endif
```

D.1.13 network.c

```c
#include "network.h"

/**** Network algorithms ****/

#define ODT_REPORTING_INTERVAL 1000
void addShortestPaths(network_type *network) {
  int odt;
  pathLinkedListElt *oldPath;

  for (odt = 0; odt < network->numODTs; odt++) {
    path_type *newPath = createNewPath(network);
    TDAStar(network, network->ODT[odt].origin, network->ODT[odt].destination, network->ODT[odt].departureTime, newPath);
    /* Does this path already exist? */
    for (oldPath = network->ODT[odt].paths->head; oldPath != NULL; oldPath = oldPath->next) {
      if (comparePaths(oldPath->path, newPath) == TRUE) break;
    }
    if (oldPath == NULL) { /* Path is new, add to relevant lists */
      insertPathLinkedList(network->paths, newPath, network->paths->tail);
      insertPathLinkedList(network->ODT[odt].paths, newPath, network->ODT[odt].paths->tail);
    } else { /* Path is a duplicate, delete */
      deletePath(newPath);
    }
    if (odt % ODT_REPORTING_INTERVAL == 0) displayMessage(FULL_NOTIFICATIONS, "Found new paths for %d of %d ODTs (%d%%)\r", odt, network->numODTs, 100 * odt / network->numODTs);
  }
  displayMessage(FULL_NOTIFICATIONS, "Found new paths for %d of %d ODTs (%d%%)\n", network->numODTs, network->numODTs, 100);

}

/* Time-dependent A* */
void TDAStar(network_type *network, node_type *origin, node_type *destination, int departureTime, path_type *path) {
  int ij, s = ptr2node(network, destination);
  turningLinkedListElt *curMovement;
  int curArc, tempLabel;
  declareVector(turning_type *, backMovement, network->numArcs);
  declareVector(int, label, network->numArcs);

  heap_type *dijkstraHeap = createHeap(network->numArcs, network->numArcs);

```

```
 if (origin->ID == 158 && destination->ID == 19 && departureTime == 0) verbosity = DEBUG;

/* Initialization */
for (ij = 0; ij < network->numArcs; ij++) {
  if (network->arc[ij].tail == origin) {
    dijkstraHeap->valueFn[ij] = min(departureTime + network->arc[ij].freeFlowToDest[s], network->timeHorizon);
    label[ij] = departureTime;
    for (curMovement = network->arc[ij].upstreamMovements->head; curMovement != NULL; curMovement = curMovement->next) {
      if (curMovement->movement->upstreamArc == &(network->origin)) break;
    }
    if (curMovement != NULL) { /* curMovement would be NULL if there are no emanating arcs from origin.  checkNetworkConnectivity assures no problems with this. */
        backMovement[ij] = curMovement->movement;
        insertHeap(dijkstraHeap, ij, dijkstraHeap->valueFn[ij]);
      }
  } else {
    dijkstraHeap->valueFn[ij] = network->timeHorizon - 1;
    label[ij] = network->timeHorizon - 1;
    backMovement[ij] = NULL;
  }
}

/* Iteration */
while (dijkstraHeap->last != NOT_IN_HEAP) {
  curArc = findMinHeap(dijkstraHeap);
    displayMessage(DEBUG, "Scanning (%d,%d)\n", network->arc[curArc].tail->ID, network->arc[curArc].head->ID);
  if (network->arc[curArc].head == destination) break;
  if (label[curArc] + network->arc[curArc].freeFlowToDest[s] >= network->timeHorizon) break;
  deleteMinHeap(dijkstraHeap);
  for (curMovement = network->arc[curArc].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    if (curMovement->movement->downstreamArc == &(network->destination)) continue;
    displayMessage(DEBUG, "Considering movement %d -> %d -> %d,", curMovement->movement->upstreamArc->tail->ID, curMovement->movement->downstreamArc->tail->ID, curMovement->movement->downstreamArc->head->ID);
    ij = ptr2arc(network, curMovement->movement->downstreamArc);
    tempLabel = label[curArc];
    tempLabel += network->arc[curArc].travelTime[tempLabel];
```

```
      tempLabel = min(tempLabel, network->timeHorizon - 1);
      tempLabel += curMovement->movement->travelTime[tempLabel];
      tempLabel = min(tempLabel, network->timeHorizon - 1);
      displayMessage(DEBUG, " exact label is %d vs current label %d\n", tempLabel, label[ij]);
      if (tempLabel < label[ij]) {
        displayMessage(DEBUG, "Noting improvement.\n");
        label[ij] = tempLabel;
        if (dijkstraHeap->nodeNDX[ij] == NOT_IN_HEAP) {
          insertHeap(dijkstraHeap, ij, label[ij] + network->arc[ij].freeFlowToDest[s]);
          displayMessage(DEBUG, "Adding (%d,%d)\n", network->arc[ij].tail->ID, network->arc[ij].head->ID);
        } else {
          decreaseKey(dijkstraHeap, ij, label[ij] + network->arc[ij].freeFlowToDest[s]);
        }
        backMovement[ij] = curMovement->movement;
      }
    }
  }
  if (dijkstraHeap->last == NOT_IN_HEAP) {
      /* warning(MEDIUM_NOTIFICATIONS, "Unusual termination for A* from %d to %d (empty heap but not at destination).  Using free-flow path.\n", origin->ID, destination->ID); */
      curArc = ptr2arc(network, origin->forwardStar->head->arc);
   }

  /* Now recover path */
  path->travelTime = min(label[curArc] + network->arc[curArc].travelTime[label[curArc]], network->timeHorizon - 1);
  path->demand = 0;
  clearTurningLinkedList(path->turnMovements);
  /* 1. Segment from current arc back to origin */
  while (network->arc[curArc].tail != origin) {
    insertTurningLinkedList(path->turnMovements, backMovement[curArc], NULL);
    curArc = ptr2arc(network, backMovement[curArc]->upstreamArc);
  }
   insertTurningLinkedList(path->turnMovements, backMovement[curArc], NULL);
  /* 2. Segment from current arc on to destination.
     This uses the movements based on free-flow times.  This is OK, because the 2 regular termination criteria for the A* iteration are
        -> Destination is reached.  In this case, we need to fill in the terminating turn movement, which is same as free-flow
```

```
            movement
         -> Time horizon is exceeded.  In this case, the path consists of the free-flow SP from here on out.
      For unusual termination (Dijkstra heap being exhausted early), in an attempt to recover the program will use the free-flow SP and warn the user.
  */
   curArc = ptr2arc(network, path->turnMovements->tail->movement->downstreamArc);
   while (path->turnMovements->tail->movement->downstreamArc != &(network->destination)) {
      if (network->arc[curArc].freeFlowMovement[s] == NULL) fatalError("A* is unable to find a path from %d to %d at time %d\n", origin->ID, destination->ID, departureTime);
      insertTurningLinkedList(path->turnMovements, network->arc[curArc].freeFlowMovement[s], path->turnMovements->tail);
      curArc = ptr2arc(network, network->arc[curArc].freeFlowMovement[s]->downstreamArc);
   }
  if (path->turnMovements->tail->movement->downstreamArc != &(network->destination)) fatalError("Path from %d to %d at time %d can't reach destination", origin->ID, destination->ID, departureTime);

  deleteHeap(dijkstraHeap);
  deleteVector(backMovement);
  deleteVector(label);

   displayMessage(DEBUG, "***** Finished iteration *****\n");
   verbosity = FULL_NOTIFICATIONS;
}


/*
   Calculate min-cost labels using free-flow times to generate lower bounds for A*
   It's essentially all-to-one label correcting in the dual graph (a node for each arc, and an arc for each turning movement)
   Assume movements have zero travel time for faster LB calculation
*/
void calculateFreeFlowSPLabels(network_type *network, int destination) {
   arcLinkedListElt *curArc;
   arc_type *upstreamArc;
   turningLinkedListElt *curMovement;
   int ij, hi;
   int tempLabel;

  heap_type *dijkstraHeap = createHeap(network->numArcs, network->
```

```
    numArcs);

  for (ij = 0; ij < network->numArcs; ij++) {
    dijkstraHeap->valueFn[ij] = INT_MAX; /* value function are the node
        labels.  Need to be much higher than time horizon in case paths
        exceed horizon. */
    network->arc[ij].freeFlowMovement[destination] = NULL;
  }

   /* Initialize heap with links terminating at destination */
   for (curArc = network->node[destination].reverseStar->head; curArc
      != NULL; curArc = curArc->next) {
      ij = ptr2arc(network, curArc->arc);
      insertHeap(dijkstraHeap, ij, curArc->arc->numCells);
      network->arc[ij].freeFlowMovement[destination] = network->arc[ij
         ].turnMovements->head->movement;
   }
  while (dijkstraHeap->last != NOT_IN_HEAP) {
    ij = findMinHeap(dijkstraHeap);
    deleteMinHeap(dijkstraHeap);
    for (curMovement = network->arc[ij].upstreamMovements->head;
       curMovement != NULL; curMovement = curMovement->next) {
         upstreamArc = curMovement->movement->upstreamArc;
         if (upstreamArc == &(network->origin)) continue;
         hi = ptr2arc(network, upstreamArc);
         tempLabel = dijkstraHeap->valueFn[ij] + upstreamArc->numCells;
         if (tempLabel < dijkstraHeap->valueFn[hi]) {
            network->arc[hi].freeFlowMovement[destination] =
               curMovement->movement;
            if (dijkstraHeap->nodeNDX[hi] == NOT_IN_HEAP) {
               insertHeap(dijkstraHeap, hi, tempLabel);
            } else {
               decreaseKey(dijkstraHeap, hi, tempLabel);
            }
         }
    }
  }

  for (ij = 0; ij < network->numArcs; ij++) {
      network->arc[ij].freeFlowToDest[destination] = dijkstraHeap->
         valueFn[ij];
  }

  deleteHeap(dijkstraHeap);

}
```

```c

/*
Check zone-to-zone connectivity for OD pairs with positive demand
Due to turning movement structure, connectivity check is run in the *dual* graph (a node for each arc, and an arc for each turning movement))
Node reachability checked after arc reachability is determined
*/
void checkNetworkConnectivity(network_type *network) {
  int ij, origin, destination;
  declareVector(bool, isNodeReachable, network->numNodes);
  declareVector(bool, isArcReachable, network->numArcs);
  queue_type scanList = createQueue(network->numNodes, network->numArcs);
  turningLinkedListElt *curMovement;

  displayMessage(FULL_NOTIFICATIONS, "Checking network connectivity...");

  for (origin = 0; origin < network->numZones; origin++) {
    /* Initialize */
    for (destination = 0; destination < network->numNodes; destination++) {
      isNodeReachable[destination] = FALSE;
    }
    for (ij = 0; ij < network->numArcs; ij++) {
      isArcReachable[ij] = FALSE;
    }
    for (curMovement = network->node[origin].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
      if (curMovement->movement->upstreamArc == &(network->origin)) {
        isArcReachable[ptr2arc(network, curMovement->movement->downstreamArc)] = TRUE;
        enQueue(&scanList, ptr2arc(network, curMovement->movement->downstreamArc));
      }
    }

    /* Identify reachable arcs */
    while (scanList.curelts > 0) {
      ij = deQueue(&scanList);
      for (curMovement = network->arc[ij].turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
        if (curMovement->movement->downstreamArc == &(network->destination)) continue;
        if (isArcReachable[ptr2arc(network, curMovement->movement->downstreamArc)] == FALSE) {
```

```
          isArcReachable[ptr2arc(network, curMovement->movement->downstreamArc)] = TRUE;
          enQueue(&scanList, ptr2arc(network, curMovement->movement->downstreamArc));
        }
      }
    }

    /* Identify reachable nodes */
    for (ij = 0; ij < network->numArcs; ij++) {
      if (isArcReachable[ij] == TRUE) isNodeReachable[ptr2node(network, network->arc[ij].head)] = TRUE;
    }


    /* Warn if positive demand and disconnected graph */
    for (destination = 0; destination < network->numZones; destination++) {
      if (isNodeReachable[destination] == FALSE) {
        if (network->staticOD[origin][destination] > 0) fatalError("Origin %d and destination %d are unconnected but have positive demand.", origin+1, destination+1);
      }
    }
  }
  deleteVector(isNodeReachable);
  deleteVector(isArcReachable);
  deleteQueue(&scanList);

  displayMessage(FULL_NOTIFICATIONS, "done.\n");
}

bool comparePaths(path_type *path1, path_type *path2) {
   turningLinkedListElt *curMovement1, *curMovement2;

   /* Easy case */
   if (pathHash(path1) != pathHash(path2)) return FALSE;

   curMovement1 = path1->turnMovements->head;
   curMovement2 = path2->turnMovements->head;

  /* Now have to compare movement by movement; they must coincide at every movement and end simultaneously */
   do {
    if (curMovement1 == NULL && curMovement2 == NULL) return TRUE; /* Both end simultaneously */
    if (curMovement1 == NULL || curMovement2 == NULL) return FALSE; /*
```

```c
        One ends before the other */
    if (curMovement1->movement != curMovement2->movement) return FALSE;
         /* Different movements */
    curMovement1 = curMovement1->next;
    curMovement2 = curMovement2->next;
  } while (TRUE);

  /* Not all movements in common.  Shouldn't reach this point, but here
      to prevent compiler warnings */
  return FALSE;
}

/* Calculates hashes... involves signed overflow which is undefined
   behavior, but is hopefully treated deterministically */
long pathHash(path_type *path) {
   long hash = 0;
   turningLinkedListElt *curMovement;

   for (curMovement = path->turnMovements->head; curMovement != NULL;
      curMovement = curMovement->next) {
    hash += (curMovement->movement->upstreamArc->head->ID * curMovement
       ->movement->downstreamArc->head->ID);
   }

   return hash;
}


/**** Basic network calculations and functions ****/


/* Returning a value greater than the time horizon indicates out-of-
   range (vehicle entering at currentTime can't leave within time
   horizon) */
int calculateLinkTravelTime(arc_type *link, int currentTime, int
   timeHorizon) {
  int t;
  if (currentTime < 0) return link->numCells;
  for (t = currentTime + link->numCells; t < timeHorizon; t++) {
    if (link->downstreamCount[t] >= link->upstreamCount[currentTime])
       return t - currentTime;
  }
  return timeHorizon + 1;
}

int calculateMovementTravelTime(turning_type *movement, int currentTime
   , int timeHorizon) {
```

```
  int t;
  if (currentTime < 0) return 0;
  for (t = currentTime; t < timeHorizon; t++) {
    if (movement->downstreamCount[t] >= movement->upstreamCount[currentTime]) return t - currentTime;
  }
  return timeHorizon + 1;
}

void calculatePathTravelTime(network_type *network, path_type *path, int departureTime) {
  int t;
  turningLinkedListElt *curMovement;

  t = departureTime;
  for (curMovement = path->turnMovements->head; curMovement != NULL; curMovement = curMovement->next) {
    /* Add delay from the movement */
    t += curMovement->movement->travelTime[t];
    t = min(t, network->timeHorizon - 1);

    /* Add delay from the next arc */
    t += curMovement->movement->downstreamArc->travelTime[t];
    t = min(t, network->timeHorizon - 1);
  }

  path->travelTime = t - departureTime;
}

/*
Copies cumulative counts from one time interval to another.
Most commonly used to copy current values when starting a new simulation tick
...but could be used more generally.
*/
void copyCounts(network_type *network, int old_t, int new_t) {
  int i, ij;
  turningLinkedListElt *curMovement;

  for (ij = 0; ij < network->numArcs; ij++) {
    network->arc[ij].upstreamCount[new_t] = network->arc[ij].upstreamCount[old_t];
    network->arc[ij].downstreamCount[new_t] = network->arc[ij].downstreamCount[old_t];
  }
  for (i = 0; i < network->numNodes; i++) {
    for (curMovement = network->node[i].turnMovements->head;
```

```
       curMovement != NULL; curMovement = curMovement->next) {
      curMovement->movement->upstreamCount[new_t] = curMovement->
         movement->upstreamCount[old_t];
      curMovement->movement->downstreamCount[new_t] = curMovement->
         movement->downstreamCount[old_t];
    }
  }
}

void displayPath(int minVerbosity, path_type *path) {
  turningLinkedListElt *curMovement;

  for (curMovement = path->turnMovements->head; curMovement != NULL;
     curMovement = curMovement->next) {
    displayMessage(minVerbosity, "%d -> %d -> %d", curMovement->
       movement->upstreamArc->tail->ID, curMovement->movement->
       upstreamArc->head->ID, curMovement->movement->downstreamArc->
       head->ID);
    if (curMovement != NULL) displayMessage(minVerbosity, ", ");
  }
}

void initializeCounts(network_type *network) {
  int i, ij, t;
  turningLinkedListElt *curMovement;

  /* Initialize all counts to zero */
  for (t = 0; t < network->timeHorizon; t++) {
    for (ij = 0; ij < network->numArcs; ij++) {
      network->arc[ij].upstreamCount[t] = 0;
      network->arc[ij].downstreamCount[t] = 0;
    }
    for (i = 0; i < network->numNodes; i++) {
      for (curMovement = network->node[i].turnMovements->head;
         curMovement != NULL; curMovement = curMovement->next) {
        curMovement->movement->upstreamCount[t] = 0;
        curMovement->movement->downstreamCount[t] = 0;
      }
    }
  }
}

void initializeTravelTimes(network_type *network) {
  int i, ij, t;
  turningLinkedListElt *curMovement;

  /* Initialize all counts to zero */
```

```
  for (t = 0; t < network->timeHorizon; t++) {
    for (ij = 0; ij < network->numArcs; ij++) {
      network->arc[ij].travelTime[t] = network->arc[ij].numCells;
    }
    for (i = 0; i < network->numNodes; i++) {
      for (curMovement = network->node[i].turnMovements->head;
           curMovement != NULL; curMovement = curMovement->next) {
        curMovement->movement->travelTime[t] = 0;
      }
    }
  }

  for (t = 0; t < network->timeHorizon; t++) {
    network->origin.travelTime[t] = 0;
    network->destination.travelTime[t] = 0;
  }
}

/* Returns arc array index from pointer to an arc */
int ptr2arc(network_type *network, arc_type *arcptr) {
  return (int) (arcptr - network->arc);
}

/* Returns node array index from pointer to a node */
int ptr2node(network_type *network, node_type *nodeptr) {
  return (int) (nodeptr - network->node);
}

/* Updates all link, movement, and path travel times */
#define TRAVEL_TIME_REPORTING_INTERVAL 100
#define ODT_REPORTING_INTERVAL 1000
void updateAllTravelTimes(network_type *network) {
  int i, ij, odt, t;
  turningLinkedListElt *curMovement;
  pathLinkedListElt *curPath;

  /* Update link and movement times */
  for (t = 0; t < network->timeHorizon; t++) {
    for (ij = 0; ij < network->numArcs; ij++) {
      network->arc[ij].travelTime[t] = calculateLinkTravelTime(&(
           network->arc[ij]), t, network->timeHorizon);
    }
    for (i = 0; i < network->numNodes; i++) {
      for (curMovement = network->node[i].turnMovements->head;
           curMovement != NULL; curMovement = curMovement->next) {
        curMovement->movement->travelTime[t] =
             calculateMovementTravelTime(curMovement->movement, t,
```

```
          network->timeHorizon);
      }
    }
    if (t % TRAVEL_TIME_REPORTING_INTERVAL == 0) displayMessage(FULL_NOTIFICATIONS, "Updated link times for %d of %d ticks (%d%%)\r", t, network->timeHorizon, 100 * t / network->timeHorizon);
  }
  displayMessage(FULL_NOTIFICATIONS, "Updated link times for %d of %d ticks (%d%%)\n", network->timeHorizon, network->timeHorizon, 100);

  /* Update path labels */
  for (odt = 0; odt < network->numODTs; odt++) {
    for (curPath = network->ODT[odt].paths->head; curPath != NULL; curPath = curPath->next) {
      calculatePathTravelTime(network, curPath->path, network->ODT[odt].departureTime);
    }
    if (odt % ODT_REPORTING_INTERVAL == 0) displayMessage(FULL_NOTIFICATIONS, "Updated path times for %d of %d ODTs (%d%%)\r", odt, network->numODTs, 100 * odt / network->numODTs);
  }
  displayMessage(FULL_NOTIFICATIONS, "Updated path times for %d of %d ODTs (%d%%)\n", network->numODTs, network->numODTs, 100);


}

/**** Generate data structures ****/


/* Create forward and reverse star lists */
void createStarLists(network_type *network) {
  int i, ij;
  for (i = 0; i < network->numNodes; i++) {
    network->node[i].forwardStar = createArcLinkedList();
    network->node[i].reverseStar = createArcLinkedList();
  }
  for (ij = 0; ij < network->numArcs; ij++) {
    insertArcLinkedList(network->arc[ij].tail->forwardStar, &(network->arc[ij]), network->arc[ij].tail->forwardStar->tail);
    insertArcLinkedList(network->arc[ij].head->reverseStar, &(network->arc[ij]), network->arc[ij].head->reverseStar->tail);
  }
  displayMessage(FULL_NOTIFICATIONS, "Created forward and reverse star lists.\n");
}
```

```
path_type *createNewPath(network_type *network) {
   path_type *newPath = newScalar(path_type);

   newPath->turnMovements = createTurningLinkedList();
   newPath->travelTime = network->timeHorizon + 1;
   newPath->demand = 0;

   return newPath;
}

void deletePath(path_type *path) {
   deleteTurningLinkedList(path->turnMovements);
   deleteScalar(path);
}


/**** Arc linked lists ****/

arcLinkedList *createArcLinkedList() {
  declareScalar(arcLinkedList, newll);
  newll->head = NULL;
  newll->tail = NULL;
  newll->size = 0;
  return newll;
}

arcLinkedListElt *insertArcLinkedList(arcLinkedList *list, arc_type *
   value, arcLinkedListElt *after) {
  declareScalar(arcLinkedListElt, newNode);
  newNode->arc = value;
  if (after != NULL) { /* Not inserting at head */
    newNode->next = after->next;
    if (list->tail == after) list->tail = newNode;
    after->next = newNode;
  } else { /* Inserting at head */
    newNode->next = list->head;
    if (list->tail == after) list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}

void deleteArcLinkedList(arcLinkedList *list) {
  arcLinkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
```

```
    savenode = curnode->next;
    killScalar(curnode);
    curnode = savenode;
  }
  killScalar(list);
}

void displayArcLinkedList(int minVerbosity, arcLinkedList *list) {
  arcLinkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p: (%d,%d) -> %p\n", (void *)curnode, curnode->arc->tail->ID, curnode->arc->head->ID, (void *)curnode->next);
    curnode = curnode->next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->tail);
}

/**** Turning movement linked lists ****/

turningLinkedList *createTurningLinkedList() {
  declareScalar(turningLinkedList, newll);
  newll->head = NULL;
  newll->tail = NULL;
  newll->size = 0;
  return newll;
}

turningLinkedListElt *insertTurningLinkedList(turningLinkedList *list, struct turning_type_s *value, turningLinkedListElt *after) {
  declareScalar(turningLinkedListElt, newNode);
  newNode->movement = value;
  if (after != NULL) { /* Not inserting at head */
    newNode->next = after->next;
    if (list->tail == after) list->tail = newNode;
    after->next = newNode;
  } else { /* Inserting at head */
    newNode->next = list->head;
    if (list->tail == after) list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}
```

```
void clearTurningLinkedList(turningLinkedList *list) {
  turningLinkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
    savenode = curnode->next;
    killScalar(curnode);
    curnode = savenode;
  }
}

void deleteTurningLinkedList(turningLinkedList *list) {
  turningLinkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
    savenode = curnode->next;
    killScalar(curnode);
    curnode = savenode;
  }
  killScalar(list);
}

void displayTurningLinkedList(int minVerbosity, struct
    turningLinkedList_s *list) {
  turningLinkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list
      ->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p: (%d,%d,%d) -> %p\n", (void *)
        curnode, curnode->movement->upstreamArc->tail->ID, curnode->
        movement->upstreamArc->head->ID, curnode->movement->
        downstreamArc->ID, (void *)curnode->next);
    curnode = curnode->next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->
      tail);
}


/**** Path linked lists ****/

pathLinkedList *createPathLinkedList() {
  declareScalar(pathLinkedList, newll);
  newll->head = NULL;
  newll->tail = NULL;
  newll->size = 0;
  return newll;
}

```

```
pathLinkedListElt *insertPathLinkedList(pathLinkedList *list, struct path_type_s *value, pathLinkedListElt *after) {
  declareScalar(pathLinkedListElt, newNode);
  newNode->path = value;
  if (after != NULL) { /* Not inserting at head */
    newNode->next = after->next;
    if (list->tail == after) list->tail = newNode;
    after->next = newNode;
  } else { /* Inserting at head */
    newNode->next = list->head;
    if (list->tail == after) list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}

void deletePathLinkedList(pathLinkedList *list) {
  pathLinkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
    savenode = curnode->next;
    killScalar(curnode);
    curnode = savenode;
  }
  killScalar(list);
}
```

D.1.14 network.h

```
#ifndef _NETWORK_H_
#define _NETWORK_H_

#include <limits.h>
#include <math.h>
#include "datastructures.h"
#include "utils.h"

typedef enum {
  UNKNOWN_CONTROL,
  CENTROID,
  NONHOMOGENEOUS,
  DIVERGE,
  MERGE,
  FOUR_WAY_STOP,
  TWO_WAY_STOP,
  BASIC_SIGNAL,
  FANCY_SIGNAL,
```

```c
  INTERCHANGE
} intersection_type;

typedef struct node_type_s {
  struct arcLinkedList_s     *forwardStar;
  struct arcLinkedList_s     *reverseStar;
  struct turningLinkedList_s *turnMovements;
  intersection_type control;
  int ID;
  float X;
  float Y;
   void *controlData; /* Pointer to additional information depending on intersection type (signal, stop, etc.) */
} node_type;

/*
'float's refer to PHYSICAL link characteristics (total link length, free flow time) in standard units
'int's  refer to CELL characteristics in cell units (timestep, cell length)
*/
typedef struct arc_type_s {
  struct cellDoublyLinkedList_s *cells;
  struct turningLinkedList_s    *turnMovements;
  struct turningLinkedList_s    *upstreamMovements;
  node_type *tail;
  node_type *head;
  int *travelTime;      /* [time] */
  int *upstreamCount;   /* [time] */
  int *downstreamCount; /* [time] */
  int *freeFlowToDest;   /* [dest] -- labels for free-flow time to destination, to be used in A* */
  struct turning_type_s **freeFlowMovement; /* [dest] -- movement for free-flow time to destination, in case A* exceeds time horizon */
  float length;
  float capacity;
  float jamDensity;
  float waveRatio;
  float freeFlowTime;
  int numCells;
  int cellCapacity;
  int cellMaxVehicles;
  int sendingFlow;
  int receivingFlow;
  int ID;
} arc_type;

```

```
typedef struct turning_type_s {
  struct vehicleDoublyLinkedList_s *vehicles;
  arc_type *upstreamArc;
  arc_type *downstreamArc;
  int *travelTime;      /* [time] */
  int *upstreamCount;   /* [time] */
  int *downstreamCount; /* [time] */
  int targetDelay;
  int capacity;         /* simulation ticks */
  float saturationFlow; /* real units */
  int flow;
  int sendingFlow;
  int receivingFlow;
} turning_type;

typedef struct path_type_s {
  struct turningLinkedList_s *turnMovements;
  int travelTime;
  int demand;
} path_type;

typedef struct ODT_type_s {
  node_type *origin;
  node_type *destination;
  int departureTime;
  int demand;
  struct vehicleDoublyLinkedList_s *vehicles;
  struct pathLinkedList_s *paths;
} ODT_type;

typedef struct network_type_s {
  arc_type  *arc;
  node_type *node;
  ODT_type  *ODT;
  struct pathLinkedList_s *paths;
  arc_type  origin; /* Artificial origin arc used to store vehicles and as upstreamArc for originating turning moveemnts */
  arc_type  destination; /* Artificial destination arc used to store vehicles and as downstreamArc for arrival turning movements */
  node_type sink; /* Artificial node for origin and destination arc*/
  float **staticOD; /* [origin][destination] */
  float totalODFlow;
  float tickLength;
  int numArcs;
  int numNodes;
  int numZones;
  long numVehicles;
```

```
  long numODTs; /* Needs to be of size long = int*int */
  int timeHorizon; /* In *clock ticks* (compare with parameters_type which has time horizon in seconds) */
  int lastVehicleOn; /* In *clock ticks* (compare with parameters_type which has last vehicle on in seconds) */
} network_type;


/**** Network algorithms ****/

void allDestinationsTDSP(network_type *network, path_type *path);
void TDAStar(network_type *network, node_type *origin, node_type *destination, int departureTime, path_type *path);
void calculateFreeFlowSPLabels(network_type *network, int destination);
void checkNetworkConnectivity(network_type *network);
bool comparePaths(path_type *path1, path_type *path2);
long pathHash(path_type *path);

/**** Basic network calculations and functions ****/

void addShortestPaths(network_type *network);
int calculateLinkTravelTime (arc_type *link, int currentTime, int timeHorizon);
int calculateMovementTravelTime (turning_type *movement, int currentTime, int timeHorizon);
void calculatePathTravelTime(network_type *network, path_type *path, int departureTime);
void copyCounts(network_type *network, int old_t, int new_t);
void displayPath(int minVerbosity, path_type *path);
void initializeCounts(network_type *network);
void initializeTravelTimes(network_type *network);
int ptr2arc(network_type *network, arc_type *arcptr);
int ptr2node(network_type *network, node_type *nodeptr);
void updateAllTravelTimes(network_type *network);

/**** Network data structures ****/

path_type *createNewPath(network_type *network);
void createStarLists(network_type *network);
void deletePath(path_type *path);

/**** Arc linked lists ****/

typedef struct arcLinkedListElt_s {
  arc_type *arc;
  struct arcLinkedListElt_s *next;
} arcLinkedListElt;
```

```
typedef struct arcLinkedList_s {
  arcLinkedListElt *head;
  arcLinkedListElt *tail;
  int size;
} arcLinkedList;

arcLinkedList *createArcLinkedList();
arcLinkedListElt *insertArcLinkedList(arcLinkedList *list, arc_type *value, arcLinkedListElt *after);
void deleteArcLinkedList(arcLinkedList *list);
void displayArcLinkedList(int minVerbosity, arcLinkedList *list);

/**** Turning movement linked lists ****/

typedef struct turningLinkedListElt_s {
  turning_type *movement;
  struct turningLinkedListElt_s *next;
} turningLinkedListElt;

typedef struct turningLinkedList_s {
  turningLinkedListElt *head;
  turningLinkedListElt *tail;
  int size;
} turningLinkedList;

turningLinkedList *createTurningLinkedList();
turningLinkedListElt *insertTurningLinkedList(turningLinkedList *list, turning_type *value, turningLinkedListElt *after);
void clearTurningLinkedList(turningLinkedList *list);
void deleteTurningLinkedList(turningLinkedList *list);
void displayTurningLinkedList(int minVerbosity, turningLinkedList *list);

/**** Path linked lists ****/

typedef struct pathLinkedListElt_s {
  path_type *path;
  struct pathLinkedListElt_s *next;
} pathLinkedListElt;

typedef struct pathLinkedList_s {
  pathLinkedListElt *head;
  pathLinkedListElt *tail;
  int size;
} pathLinkedList;

```

```
pathLinkedList *createPathLinkedList();
pathLinkedListElt *insertPathLinkedList(pathLinkedList *list, struct
    path_type_s *value, pathLinkedListElt *after);
void deletePathLinkedList(pathLinkedList *list);

#endif
```

D.1.15 sampling.c

```
#include "sampling.h"

/* Factorial table to save computation */
long factorial[10] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};

/* Constants used in economized polynomial approximation */
#define A_PRECISION 7
double A[A_PRECISION + 1] = {-0.49999999, 0.33333328, -0.25000678,
    0.20001178, -0.16612694, 0.14218783, -0.13847944, 0.12500596};

double randomSample(double mean, double stdev, distribution_type
    distribution) {
  switch (distribution) {
  case DETERMINISTIC:
    return mean;
  case UNIFORM:
    return randUniform(mean - stdev * 1.73205080756887729353, mean +
        stdev * 1.73205080756887729353);
  case POISSON:
    return randPoisson(mean);
  case LOGNORMAL:
    return randLognormalMeanStdev(mean, stdev);
  case NORMAL:
    return randNormal(mean, stdev);
  case EXPONENTIAL:
    return randExponential(1 / mean);
  default:
    fatalError("Unknown distribution type %d\n", distribution);
  }
  return 0; /* Should never be reached; included to avoid compiler
      warnings */
}

long randInt(long min, long max) {
   return min + floor(randUniform(0, max-min));
}

long stochasticRound(double x) {
```

```
  return floor(x) + (randUniform(0, 1) < x - floor(x) ? 1 : 0);
}


/* Poisson sampling procedure from Ahrens and Dieter;
   variable names and labels taken directly from their paper */
#define TOO_HIGH 50
long randPoisson(double mu) {
  double s, d, J, L, T, G, U = 0, E;
  double omega, b1, b2, c3, c2, c1, c0, c;
  double px, py, fx, fy;
  double delta;
  double V, X;
  double M, p, q, p0;
  double P[36];
  int i;
  long K = 0;
  char comingFrom;
  if (mu >= 10) goto CASE_A;
  else goto CASE_B;

CASE_A:
  s = sqrt(mu);
  d = 6 * mu * mu;
  L = floor(mu - 1.1484);
/* N: Label indicated in algorithm but not used in code; commented out
   to avoid compiler warnings */
  T = randNormal(0, 1);
  G = mu + s * T;
  if (G >= 0) K = floor(G); else goto P;
/* I: Label indicated in algorithm but not used in code; commented out
   to avoid compiler warnings */
  if (K >= L) return K;
/* S: Label indicated in algorithm but not used in code; commented out
   to avoid compiler warnings */
  U = randUniform(0, 1);
  if (d * U >= (mu - K) * (mu - K) * (mu - K)) return K;
P:
  omega = 0.39894228 / s;
  b1 = 0.0416666667 / mu;
  b2 = 0.3 * b1 * b1;
  c3 = 0.142857143 * b1 * b2;
  c2 = b2 - 15 * c3;
  c1 = b1 - 6 * b2 + 45 * c3;
  c0 = 1 - b1 + 3 * b2 - 15 * c3;
  c = 0.1069 / mu;
  if (G >= 0) { comingFrom = 'P'; goto F; } else goto E;
```

```
Q:
  if (fy * (1 - U) <= py * exp(px - fx)) return K;
E:
  E = randExponential(1);
  U = randUniform(0, 1);
  U = U + U - 1;
  T = 1.8 + E * copysign(1.0, U);
  if (T <= -0.6744) goto E;
  K = floor(mu + s * T);
  comingFrom = 'E';
  goto F;
H:
  if (c * fabs(U) > py * exp(px + E) - fy * exp(fx + E)) goto E;
  return K;
F:
  if (K < 10) {
    px = -mu;
    py = pow(mu, K) / factorial[K];
  } else {
    delta = 0.0833333333 / K;
    delta = delta - 4.8 * pow(delta, 3);
    V = (mu - K) / (double) K;
    if (fabs(V) > 0.25) {
      px = K * log(1 + V) - (mu - K) - delta;
    } else {
      px = A[A_PRECISION];
      for (i = A_PRECISION - 1; i >= 0; i--) {
        px = A[i] + px * V;
      }
      px *= K * V * V;
      px -= delta;
    }
    py = 0.39894228 / sqrt(K);
  }
  X = (K - mu + 0.5) / s;
  fx = -0.5 * X * X;
  fy = omega * (((c3 * X * X + c2) * X * X + c1) * X * X + c0);
  if (comingFrom == 'P') goto Q;
  if (comingFrom == 'E') goto H;
  fatalError("randomPoisson: invalid comingFrom value");

CASE_B:
  if (mu > 1) M = mu; else M = 1;
  L = 0;
  p = exp(-mu);
  q = p;
  p0 = p;
```

```
  U = randUniform(0, 1);
U:
  K = 0;
  if (U <= p0) return K;
/* T:  Label indicated in algorithm but not used in code; commented out to avoid compiler warnings */
  if (L == 0) goto C;
  if (U > 0.458) {
    if (L < M) J = L; else J = M;
  } else {
    J = 1;
  }
  for (K = J; K <= L; K++) {
    if (U <= P[K]) return K;
  }
  goto U;
C:
  for (K = L + 1; K <= 35; K++) {
    p = p * mu / (double) K;
    q = q + p;
    P[K] = q;
    if (U <= q) {
      L = K;
      return K;
    }
  }
  L = 35;
  goto U;
}

double randExponential(double lambda) {
  double d = 1;
  while (d >= 1) d = randUniform(0, 1);
  return -log(1 - d) / lambda;
}

double randUniform(double a, double b) {
  return a + (((double) rand() - 1) / RAND_MAX) * (b - a);
}

double randNormal(double mean, double stdev) {
  static long numSamples = 0;
  static double Z2;
  if ((numSamples++ & 1) == 0) {
    double Z1, U1, U2;
    do { U1 = randUniform(0, 1); } while (U1 <= 0 || U1 >= 1);
    do { U2 = randUniform(0, 1); } while (U1 <= 0 || U1 >= 1);
```

```
    Z1 = sqrt(-2 * log(U1)) * cos(6.28318531 * U2);
    Z2 = sqrt(-2 * log(U1)) * sin(6.28318531 * U2);
    return mean + stdev * Z1;
  } else {
    return mean + stdev * Z2;
  }
}

double randLognormal(double mu, double sigma) {
  return exp(randNormal(mu, sigma));
}

double randLognormalMeanStdev(double mean, double stdev) {
  return randLognormal( log(mean) - 0.5 * log(1 + (stdev * stdev) / (mean * mean)) , log(1 + (stdev * stdev) / (mean * mean)));
}

/* Create an extra row/col with the "extra bits" left over... calculate row/col sums then % precision */
void roundStochasticMatrix(float **matrix, int numRows, int numCols, int precision) {
   int bit, row, col;
   long scaleFactor = 1 << precision, sum;
   declareMatrix(long, scaledMatrix, numRows + 1, numCols + 1);

   /* Rounding code needs integer values; multiply by appropriate power of 2 and cast to int */
   for (row = 0; row < numRows; row++) {
      for (col = 0; col < numCols; col++) {
         matrix[row][col] *= scaleFactor;
         scaledMatrix[row][col] = round2long(matrix[row][col]);
      }
   }

   /* Fill in "remainder" rows and colums to have integer row/column sums */
   for (row = 0; row < numRows; row++) {
      sum = 0;
      for (col = 0; col < numCols; col++) {
         sum += scaledMatrix[row][col];
      }
      scaledMatrix[row][numCols] = scaleFactor - sum % scaleFactor;
   }
   for (col = 0; col < numCols + 1; col++) {
      sum = 0;
      for (row = 0; row < numRows; row++) {
         sum += scaledMatrix[row][col];
```

```
      }
      scaledMatrix[numRows][col] = scaleFactor - sum % scaleFactor;
   }


   /* Now do stochastic rounding, bit by bit */
   for (bit = 0; bit < precision; bit++) {
      roundIntegerBit(scaledMatrix, numRows + 1, numCols + 1, bit);

   }

   /* Now undo scaling */
   for (row = 0; row < numRows; row++) {
      for (col = 0; col < numCols; col++) {
         matrix[row][col] = scaledMatrix[row][col] >> precision;
      }
   }

   deleteMatrix(scaledMatrix, numRows + 1);
}

/* What happens if there are an odd number of 'bits' ?  algo suggested an extra row/col ???
   Perhaps this is not an issue unless precision is super high */
void roundIntegerBit(long **matrix, int numRows, int numCols, int bit) {
   int row, col, switchRow, switchCol, parity;
   long mask = 1 << bit;

   /* Generate network structure */
   declareMatrix(int, rowMatch, numRows, numCols);
   declareMatrix(int, colMatch, numRows, numCols);
   generateParityNetwork(matrix, rowMatch, colMatch, numRows, numCols, mask);

   for (row = 0; row < numRows; row++) {
      for (col = 0; col < numCols; col++) {
         if (matrix[row][col] & mask) {
            switchRow = row;
            switchCol = col;
        parity = rand() & 1;
            do {
               switch (parity) {
               case 0: /* Set bit to zero and move row-wise*/
                  matrix[switchRow][switchCol] -= mask;
                  switchRow = rowMatch[switchRow][switchCol];
                  break;
```

```
               case 1: /* Set bit to one and move column-wise */
                  matrix[switchRow][switchCol] += mask;
                  switchCol = colMatch[switchRow][switchCol];
                  break;
               }
               parity = 1 - parity;
            } while (matrix[switchRow][switchCol] & mask);
         }
      }
   }

   deleteMatrix(rowMatch, numRows);
   deleteMatrix(colMatch, numRows);
}

void generateParityNetwork(long **matrix, int **rowMatch, int **
   colMatch, int numRows, int numCols, long mask) {
   int row, col, companionRow = IS_MISSING;
   declareVector(int, companionCol, numRows);

   for (row = 0; row < numRows; row++) {
      companionCol[row] = IS_MISSING;
      for (col = 0; col < numCols; col++) {
         rowMatch[row][col] = IS_MISSING;
         colMatch[row][col] = IS_MISSING;
      }
   }

   for (col = 0; col < numCols; col++) {
      for (row = 0; row < numRows; row++) {
         if (matrix[row][col] & mask) {
            if (companionRow == IS_MISSING) {
               companionRow = row;
            } else {
               rowMatch[row][col] = companionRow;
               rowMatch[companionRow][col] = row;
               companionRow = IS_MISSING;
            }
            if (companionCol[row] == IS_MISSING) {
               companionCol[row] = col;
            } else {
               colMatch[row][col] = companionCol[row];
               colMatch[row][companionCol[row]] = col;
               companionCol[row] = IS_MISSING;
            }
         }
      }
```

```
    }

    deleteVector(companionCol);
}
```

D.1.16 `sampling.h`

```
#ifndef _SAMPLING_H_
#define _SAMPLING_H_

#include <math.h>
#include <stdlib.h>
#include "datastructures.h"
#include "utils.h"

typedef enum {
  DETERMINISTIC,
  UNIFORM,
  POISSON,
  NORMAL,
  EXPONENTIAL,
  LOGNORMAL,
  UNKNOWN_DISTRIBUTION
} distribution_type;


/* Random number generation */

double randomSample(double mean, double stdev, distribution_type distribution);

long randInt(long min, long max);
long randPoisson(double mu);
double randExponential(double lambda);
double randUniform(double a, double b);
double randNormal(double mean, double stdev);
double randLognormal(double mu, double sigma);
double randLognormalMeanStdev(double mean, double stdev);

long stochasticRound(double x);

/* Stochastic rounding */

long roundStochastic(double x);
void roundStochasticMatrix(float **matrix, int numRows, int numCols, int precision); /* Preserves row and column sums with stochastic rounding */
```

```c
void roundIntegerBit(long **matrix, int numRows, int numCols, int bit);
void generateParityNetwork(long **matrix, int **rowMatch, int **
    colMatch, int numRows, int numCols, long mask);

#endif
```

D.1.17 `datastructures.c`

```c
#include "datastructures.h"

/*
This file contains implementation for commonly-used data structures,
    including
singly and doubly linked lists, binary heaps, queues, as well as memory
     allocation
and deallocation.
*/

/******************
 ** Linked lists **
 ******************/

/**** Singly linked lists ****/

linkedList *createLinkedList() {
  declareScalar(linkedList, newll);
  newll->head = NULL;
  newll->tail = NULL;
  newll->size = 0;
  return newll;
}

linkedListElt *insertLinkedList(linkedList *list, int value,
    linkedListElt *after) {
  declareScalar(linkedListElt, newNode);
  newNode->value = value;
  if (after != NULL) { /* Not inserting at head */
    newNode->next = after->next;
    if (list->tail == after) list->tail = newNode;
    after->next = newNode;
  } else { /* Inserting at head */
    newNode->next = list->head;
    if (list->tail == after) list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
```

```c
}

void deleteLinkedList(linkedList *list) {
  linkedListElt *savenode, *curnode = list->head;
  while (curnode != NULL) {
    savenode = curnode->next;
    killScalar(curnode);
    curnode = savenode;
  }
  killScalar(list);
}

void displayLinkedList(int minVerbosity, linkedList *list) {
  linkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p: %d -> %p\n", (void *)curnode, curnode->value, (void *)curnode->next);
    curnode = curnode->next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->tail);
}

/**** Doubly linked lists ****/

doublyLinkedList *createDoublyLinkedList() {
  declareScalar(doublyLinkedList, newdll);
  newdll->head = NULL;
  newdll->tail = NULL;
  newdll->size = 0;
  return newdll;
}

doublyLinkedListElt *insertDoublyLinkedList(doublyLinkedList *list, double value, doublyLinkedListElt *after) {
  declareScalar(doublyLinkedListElt, newNode);
  newNode->value = value;
  if (after != NULL) {
    newNode->prev = after;
    newNode->next = after->next;
    if (list->tail != after) newNode->next->prev = newNode; else list->tail = newNode;
    after->next = newNode;
  } else {
    newNode->prev = NULL;
```

```
    newNode->next = list->head;
    if (list->tail != after) newNode->next->prev = newNode; else list->tail = newNode;
    list->head = newNode;
  }
  list->size++;
  return newNode;
}

void deleteDoublyLinkedList(doublyLinkedList *list) {
  while (list->head != NULL)
    deleteDoublyLinkedListElt(list, list->tail);
  killScalar(list);
}

void deleteDoublyLinkedListElt(doublyLinkedList *list, doublyLinkedListElt *elt) {
  if (list->tail != elt) {
    if (list->head != elt) elt->prev->next = elt->next; else list->head = elt->next;
    elt->next->prev = elt->prev;
  } else {
    list->tail = elt->prev;
    if (list->head != elt) elt->prev->next = elt->next; else list->head = elt->next;
  }
  list->size--;
  killScalar(elt);
}

void displayDoublyLinkedList(int minVerbosity, doublyLinkedList *list) {
  doublyLinkedListElt *curnode = list->head;
  displayMessage(minVerbosity, "Start of the list: %p\n", (void *)list->head);
  while (curnode != NULL) {
    displayMessage(minVerbosity, "%p %f %p %p\n", (void *)curnode, curnode->value, (void *)curnode->prev, (void *)curnode->next);
    curnode = (*curnode).next;
  }
  displayMessage(minVerbosity, "End of the list: %p\n", (void *)list->tail);
}

/************
 ** Queues **
 ************/
```

```
/**** Standard queue with memory ****/

queue_type createQueue(long size, long eltsize) {
  long i;

  queue_type queue;
  queue.node = newVector(size, long);
  queue.history = newVector(eltsize, char);
  queue.readptr = 0;
  queue.writeptr = 0;
  queue.size = size;
  queue.curelts = 0;

  for (i = 0; i < eltsize; i++) queue.history[i] = NEVER_IN_QUEUE;
  for (i = 0; i < size; i++) queue.node[i] = 0;
  return queue;
}

void deleteQueue(queue_type *queue) {
  deleteVector(queue->node);
  deleteVector(queue->history);
}

void enQueue(queue_type *queue, long elt) {
  if (queue->history[elt] == IN_QUEUE) return;
  if (queue->curelts == queue->size) fatalError("Queue not large enough!");
  queue->curelts++;
  queue->node[queue->writeptr] = elt;
  queue->writeptr++;
  if (queue->writeptr == queue->size) queue->writeptr = 0;
  queue->history[elt] = IN_QUEUE;
}

void frontQueue(queue_type *queue, long elt) {
  if (queue->history[elt] == IN_QUEUE) return;
  if (queue->readptr == 0) queue->readptr = queue->size; else queue->readptr--;
  if (queue->curelts == queue->size) fatalError("Queue not large enough!");
  queue->curelts++;
  queue->node[queue->readptr] = elt;
  queue->history[elt] = IN_QUEUE;
}

long deQueue(queue_type *queue) {
```

```
  long val = queue->node[queue->readptr];
  queue->history[queue->node[queue->readptr]] = WAS_IN_QUEUE;
  queue->readptr++;
  queue->curelts--;
  if (queue->readptr >= queue->size) queue->readptr = 0;
  return val;
}

void displayQueue(int minVerbosity, queue_type *queue) {
  long i;
  for (i = 0; i < queue->size; i++) {
    displayMessage(minVerbosity, "%ld ", queue->node[i]);
    if (i == queue->readptr) displayMessage(minVerbosity, "R"); else
       displayMessage(minVerbosity, " ");
    if (i == queue->writeptr) displayMessage(minVerbosity, "W"); else
       displayMessage(minVerbosity, " ");
    displayMessage(minVerbosity, "   %ld %d", i, queue->history[i]);
    displayMessage(minVerbosity, "\n");
  }
}


/*****************
 ** Binary heaps **
 *****************/

heap_type *createHeap(int heapsize, int eltsize) {

  int i;
  declareScalar(heap_type, newHeap);
  newHeap->node = newVector(heapsize, int);
  newHeap->nodeNDX = newVector(eltsize, int);
  newHeap->last = NOT_IN_HEAP;
  newHeap->valueFn = newVector(eltsize, int);
  newHeap->maxsize = heapsize;
  newHeap->maxelts = eltsize;

  for(i = 0; i < eltsize; i++) newHeap->nodeNDX[i] = NOT_IN_HEAP;
  return newHeap;
}

void insertHeap(heap_type *heap, int key, int value) {
  int elt = ++(heap->last);
  if (heap->last >= heap->maxsize) fatalError("Heap not big enough.");
  heap->node[heap->last] = key;
  heap->nodeNDX[key] = heap->last;
  heap->valueFn[key] = value;
```

```
  siftUp(heap, elt);
}

int findMinHeap(heap_type *heap) {
  return heap->node[0];
}

void deleteMinHeap(heap_type *heap) {
  if (heap->last < 0) fatalError("Negative heap size!");
  heap->nodeNDX[heap->node[heap->last]] = 0;
  heap->nodeNDX[heap->node[0]] = NOT_IN_HEAP;
  if (heap->last > 0) swap(heap->node[0], heap->node[heap->last]);
  heap->last--;
  if (heap->last >= 0) siftDown(heap, 1);
}

void deleteHeap(heap_type *heap) {
  deleteVector(heap->node);
  deleteVector(heap->nodeNDX);
  deleteVector(heap->valueFn);
  deleteScalar(heap);
}

void decreaseKey(heap_type *heap, int elt, int value) {
  heap->valueFn[heap->node[heap->nodeNDX[elt]]] = value;
  siftUp(heap, heap->nodeNDX[elt]);
}

void increaseKey(heap_type *heap, int elt, int value) {
  heap->valueFn[heap->node[heap->nodeNDX[elt]]] = value;
  siftDown(heap, heap->nodeNDX[elt]);
}

void siftUp(heap_type *heap, int elt) {
  while (elt > 0 && heap->valueFn[heap->node[elt]] < heap->valueFn[heap->node[heapPred(elt)]]) {
    swap(heap->nodeNDX[heap->node[elt]], heap->nodeNDX[heap->node[heapPred(elt)]]);
    swap(heap->node[elt], heap->node[heapPred(elt)]);
    elt = heapPred(elt);
  }
}

void siftDown(heap_type *heap, int elt) {
  int tmp;
  while (heapSucc(elt) <= heap->last && heap->valueFn[heap->node[elt]] > heap->valueFn[heap->node[minChild(heap, elt)]]) {
```

```
    tmp = minChild(heap, elt);
    swap(heap->nodeNDX[heap->node[elt]], heap->nodeNDX[heap->node[tmp
       ]]);
    swap(heap->node[elt], heap->node[tmp]);
    elt = tmp;
  }
}

int heapPred(int elt) {
  return (elt - 1) / 2;
}

int heapSucc(int elt) {
  return (elt * 2) + 1;
}

int minChild(heap_type *heap, int elt) {
  if (heapSucc(elt) == heap->last)
    return heap->last;
  if (heap->valueFn[heap->node[heapSucc(elt)]] <= heap->valueFn[heap->
     node[heapSucc(elt) + 1]])
    return heapSucc(elt);
  return heapSucc(elt) + 1;
}

void heapify(heap_type *heap) {
  long i;
  for (i = heapPred(heap->last); i >= 0; i--)
    siftDown(heap, i);
}

void displayHeap(int minVerbosity, heap_type *heap) {
  int i;
  displayMessage(minVerbosity, "HEAP current size, capacity, number of
     elements: %d %d %d\n", heap->last, heap->maxsize, heap->maxelts);
  displayMessage(minVerbosity, "HEAP STATUS");
  for (i = 0; i < heap->maxsize; i++) {
    displayMessage(minVerbosity, "\n%d %d", i, heap->node[i]);
    if (i == heap->last) displayMessage(minVerbosity, " LAST");
  }
  displayMessage(minVerbosity, "\nNODE NDX");
  for (i = 1; i <= heap->maxelts; i++)
    displayMessage(minVerbosity, "\n%d %d", i, heap->nodeNDX[i]);
  displayMessage(minVerbosity, "\nVALUE FUNCTION");
  for (i = 1; i <= heap->maxelts; i++)
    displayMessage(minVerbosity, "\n%d %f", i, heap->valueFn[i]);
}
```

```
/***************************
 ** Memory (de)allocation **
 ***************************/


void *allocateScalar(size_t size) {
  void *scalar = malloc(size);
  if (scalar == NULL) fatalError("Unable to allocate memory for a
     scalar.");
  #ifdef MEMCHECK
  memcheck_numScalars++;
  if (memcheck_numScalars > MEMCHECK_THRESHOLD) { displayMessage(DEBUG,
      "Now have %ld scalars.\n", memcheck_numScalars); }
  #endif
  return scalar;
}

void *allocateVector(long u, size_t size) {
  void *vector = malloc(u * size);
  if (vector == NULL) fatalError("Unable to allocate memory for vector
     of size %ld.", u);
  #ifdef MEMCHECK
  memcheck_numVectors++;
  if (memcheck_numVectors > MEMCHECK_THRESHOLD) { displayMessage(DEBUG,
      "Now have %ld vectors.\n", memcheck_numVectors); }
  #endif
  return vector;
}

void **allocateMatrix(long u1, long u2, size_t size) {
  long i;
  void **matrix = malloc(u1 * sizeof(void *));
  if (matrix == NULL) fatalError("Unable to allocate memory for matrix
     of size %ld x %ld.", u1, u2);
  for (i = 0; i < u1; i++) {
    matrix[i] = malloc(u2 * size);
    if (matrix[i] == NULL) fatalError("Unable to allocate memory for
       matrix of size %ld x %ld.", u1, u2);
  }
  #ifdef MEMCHECK
  memcheck_numMatrices++;
  if (memcheck_numMatrices > MEMCHECK_THRESHOLD) { displayMessage(DEBUG
     , "Now have %ld matrices.\n", memcheck_numMatrices); }
  #endif
  return matrix;
}
```

```c

void ***allocate3DArray(long u1, long u2, long u3, size_t size) {
  long i, j;
  void ***matrix = malloc(u1 * sizeof(void **));
  if (matrix == NULL) fatalError("Unable to allocate 3D array of size %ld x %ld x %ld.", u1, u2, u3);
  for (i = 0; i < u1; i++) {
    matrix[i] = malloc(u2 * sizeof(void *));
    if (matrix[i] == NULL) fatalError("Unable to allocate 3D array of size %ld x %ld x %ld.", u1, u2, u3);
    for (j = 0; j < u2; j++) {
      matrix[i][j] = malloc(u3 * size);
      if (matrix[i][j] == NULL) fatalError("Unable to allocate 3D array of size %ld x %ld x %ld.", u1, u2, u3);
    }
  }
  #ifdef MEMCHECK
  memcheck_num3DArrays++;
  if (memcheck_num3DArrays > MEMCHECK_THRESHOLD) { displayMessage(DEBUG, "Now have %ld 3D arrays.\n", memcheck_num3DArrays); }
  #endif
  return matrix;
}

void killScalar(void *scalar) {
  free(scalar);
  #ifdef MEMCHECK
  memcheck_numScalars--;
  if (memcheck_numScalars > MEMCHECK_THRESHOLD) { displayMessage(DEBUG, "Now have %ld scalars.\n", memcheck_numScalars);  }
  #endif
}

void killVector(void *vector) {
  free(vector);
  #ifdef MEMCHECK
  memcheck_numVectors--;
  if (memcheck_numVectors > MEMCHECK_THRESHOLD) { displayMessage(DEBUG, "Now have %ld vectors.\n", memcheck_numVectors); }
  #endif
}

void killMatrix(void **matrix, long u1) {
  long i;
  for (i = 0; i < u1; i++) free(matrix[i]);
  free(matrix);
  #ifdef MEMCHECK
```

```
  memcheck_numMatrices--;
  if (memcheck_numMatrices > MEMCHECK_THRESHOLD) { displayMessage(DEBUG, "Now have %ld matrices.\n", memcheck_numMatrices); }
  #endif
}

void kill3DArray(void ***array, long u1, long u2) {
  long i, j;
  for (i = 0; i < u1; i++) {
    for (j = 0; j < u2; j++) {
      free(array[i][j]);
    }
    free(array[i]);
  }
  free(array);
  #ifdef MEMCHECK
  memcheck_num3DArrays--;
  if (memcheck_num3DArrays > MEMCHECK_THRESHOLD) { displayMessage(DEBUG, "Now have %ld 3D arrays.\n", memcheck_num3DArrays); }
  #endif
}
```

D.1.18 `datastructures.h`

```
#ifndef _DATASTRUCTURES_H_
#define _DATASTRUCTURES_H_

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"



/******************
 ** Linked lists **
 ******************/

/***** Singly linked lists *****/

typedef struct linkedListElt_s {
  int value;
  struct linkedListElt_s *next;
} linkedListElt;

typedef struct {
  linkedListElt *head;
  linkedListElt *tail;
```

```
  long size;
} linkedList;

linkedList *createLinkedList();
linkedListElt *insertLinkedList(linkedList *list, int value,
    linkedListElt *after);
void deleteLinkedList(linkedList *list);
void displayLinkedList(int minVerbosity, linkedList *list);

/**** Doubly linked lists ****/

typedef struct doublyLinkedListElt_s {
  double value;
  struct doublyLinkedListElt_s *next;
  struct doublyLinkedListElt_s *prev;
} doublyLinkedListElt;

typedef struct {
  doublyLinkedListElt *head;
  doublyLinkedListElt *tail;
  long size;
} doublyLinkedList;

doublyLinkedList *createDoublyLinkedList();
doublyLinkedListElt *insertDoublyLinkedList(doublyLinkedList *list,
    double value, doublyLinkedListElt *after);
void deleteDoublyLinkedList(doublyLinkedList *list);
void deleteDoublyLinkedListElt(doublyLinkedList *list,
    doublyLinkedListElt *elt);
void displayDoublyLinkedList(int minVerbosity, doublyLinkedList *list);


/************
 ** Queues **
 ************/

/**** Standard queue with memory ****/

enum {
  IN_QUEUE,
  WAS_IN_QUEUE,
  NEVER_IN_QUEUE
};

typedef enum {
  DEQUE,
  FIFO,
```

```
  LIFO
} queueDiscipline;

typedef struct {
  long* node;
  char* history;
  long readptr;
  long writeptr;
  long size;
  long curelts;
} queue_type;

queue_type createQueue(long size, long eltsize);
void deleteQueue(queue_type *queue);
void enQueue(queue_type *queue, long elt);
void frontQueue(queue_type *queue, long elt);
long deQueue(queue_type *queue);
void displayQueue(int minVerbosity, queue_type *queue);

/******************
 ** Binary heaps **
 ******************/

#define NOT_IN_HEAP -1

typedef struct {
  int* node;
  int last;
  int* valueFn;
  int* nodeNDX;
  int maxsize;
  int maxelts;
} heap_type;

heap_type *createHeap(int heapsize, int eltsize);
void insertHeap(heap_type *heap, int key, int value) ;
int  findMinHeap(heap_type *heap);
void deleteMinHeap(heap_type *heap);
void deleteHeap(heap_type *heap);
void decreaseKey(heap_type *heap, int elt, int value);
void increaseKey(heap_type *heap, int elt, int value);
void siftUp(heap_type *heap, int elt);
void siftDown(heap_type *heap, int elt);
int  heapPred(int elt);
int  heapSucc(int elt);
int  minChild(heap_type *heap, int elt);
void heapify(heap_type *heap);
```

```c
void displayHeap(int minVerbosity, heap_type *heap);

/**************************
 ** Memory (de)allocation **
 **************************/


/* Comment out this line to disable memory leak checking */
/* #define MEMCHECK */
#define MEMCHECK_THRESHOLD 1000 /* Threshold before reporting data structure counts for memory leak checking */

void *allocateScalar(size_t size);
void *allocateVector(long u, size_t size);
void **allocateMatrix(long u1, long u2, size_t size);
void ***allocate3DArray(long u1, long u2, long u3, size_t size);
void killScalar(void *scalar);
void killVector(void *vector);
void killMatrix(void **matrix, long u1);
void kill3DArray(void ***array, long u1, long u2);

#define newScalar(y)                (y *)allocateScalar(sizeof(y))
#define newVector(u,y)              (y *)allocateVector(u,sizeof(y))
#define newMatrix(u1,u2,y)          (y **)allocateMatrix(u1,u2,sizeof(y))
#define new3DArray(u1,u2,u3,y)      (y ***)allocate3DArray(u1,u2,u3,sizeof(y))

#define declareScalar(y,S)          y *S = newScalar(y)
#define declareVector(y,V,u)        y *V = newVector(u,y)
#define declareMatrix(y,M,u1,u2)    y **M = newMatrix(u1,u2,y)
#define declare3DArray(y,A,u1,u2,u3) y ***A = new3DArray(u1,u2,u3,y)

#define deleteScalar(y)             killScalar(y)
#define deleteVector(y)             killVector(y)
#define deleteMatrix(y,u1)          killMatrix((void **)y,u1)
#define delete3DArray(y,u1,u2)      kill3DArray((void ***)y,u1,u2)

#ifdef MEMCHECK
long memcheck_numScalars, memcheck_numVectors, memcheck_numMatrices, memcheck_num3DArrays;
#endif

#endif
```

D.1.19 utils.c

```c
#include "utils.h"

void waitForKey() {
  getchar();
}

void SWAP(void* a, void* b, int size) {
  void* c = malloc(size);
  memcpy(c, a, size);
  memcpy(a, b, size);
  memcpy(b, c, size);
  free(c);
}

double updateElapsedTime(clock_t startTime, double *elapsedTime) {
  *elapsedTime += ((double) (clock() - startTime)) / CLOCKS_PER_SEC;
  return *elapsedTime;
}

/********************
 ** Status messages **
 ********************/

void displayMessage(int minVerbosity, char *format, ...) {
  va_list message;
  if (verbosity < minVerbosity) return;
  if (minVerbosity < DEBUG) {
    va_start(message, format);
    vprintf(format, message);
    va_end(message);
    fflush(stdout);
  }
  #ifdef DEBUG_MODE
    va_start(message, format);
    vfprintf(debugFile, format, message);
    va_end(message);
    fflush(debugFile);
  #endif
}

void fatalError(char *format, ...) {
  va_list message;
  va_start(message, format);
  printf("Fatal error: ");
  vprintf(format, message);
  va_end(message);
  fflush(stdout);
```

```
  #ifdef DEBUG_MODE
    va_start(message, format);
    fprintf(debugFile, "Fatal error: ");
    vfprintf(debugFile, format, message);
    va_end(message);
    fflush(debugFile);
  #endif
  if (PAUSE_ON_ERROR == TRUE) waitForKey();
  #ifdef DEBUG_MODE
    fclose(debugFile);
  #endif
  exit(EXIT_FAILURE);
}

void warning(int minVerbosity, char *format, ...) {
  va_list message;
  if (verbosity < minVerbosity) return;
  va_start(message, format);
  printf("Warning: ");
  vprintf(format, message);
  va_end(message);
  fflush(stdout);
  #ifdef DEBUG_MODE
    va_start(message, format);
    fprintf(debugFile, "Warning: ");
    vfprintf(debugFile, format, message);
    va_end(message);
    fflush(debugFile);
  #endif
  if (PAUSE_ON_WARNING == TRUE) waitForKey();
}
```

D.1.20 utils.h

```
#ifndef _UTILS_H_
#define _UTILS_H_

#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define DEBUG_MODE /* Uncomment this line to echo output to log file. */

#define IS_MISSING -1
```

```
#define STRING_SIZE 9999

#define PAUSE_ON_ERROR FALSE
#define PAUSE_ON_WARNING FALSE


/*
Standard units: feet, seconds
Multiplying a quantity by these values will convert it to standard
    units
Dividing a quantity by these values will convert it from standard units
*/
#define HOURS      3600.0
#define MINUTES    60.0
#define SECONDS    1.0
#define MILES      5280.0
#define KILOMETERS 3280.839895
#define METERS     3.280839895
#define FEET       1.0
#define INCHES     0.083333333

#define min(x,y)   ( ((x)<(y)) ? (x) : (y) )
#define max(x,y)   ( ((x)>(y)) ? (x) : (y) )
#define swap(a,b) SWAP(&a, &b, sizeof(a))
#define round2int(x)  (int)((x) < 0 ? ((x) - 0.5) : ((x) + 0.5))
#define round2long(x)  (long)((x) < 0 ? ((x) - 0.5) : ((x) + 0.5))

#ifndef __cplusplus
typedef enum {
  FALSE,
  TRUE
} bool;
#endif

#ifdef DEBUG_MODE
  char debugFileName[STRING_SIZE];
  FILE *debugFile;
#endif


enum { /* Verbosity levels for status messages */
  NOTHING,
  LOW_NOTIFICATIONS,
  MEDIUM_NOTIFICATIONS,
  FULL_NOTIFICATIONS,
  DEBUG,
  FULL_DEBUG
```

```c
};


int verbosity;


#ifdef MEMCHECK
extern long memcheck_numScalars, memcheck_numVectors,
    memcheck_numMatrices;
#endif

void waitForKey();
void SWAP(void *a, void *b, int size);
double updateElapsedTime(clock_t startTime, double *elapsedTime);

/********************
 ** Status messages **
 ********************/

void displayMessage(int minVerbosity, char *format, ...);
void fatalError(char *format, ...);
void warning(int minVerbosity, char *format, ...);

#endif
```

D.2 Warrants module

D.2.1 main_warrant.c

```c
#include "main_warrant.h"

int main(int numArgs, char *args[]) {
    #ifdef DEBUG_MODE /* Debug mode enables extra logging.  Define this
        macro in utils.h */
        debugFile = openFile("logfile.txt", "w");
        verbosity = DEBUG;
        displayMessage(DEBUG, "Starting new run.\n");
    #endif

    switch (numArgs) {
    case 5: /* Comprehensive warrant analysis */
        generateWarrantNodeControls(args[1], args[2], args[3], args[4]);
        break;
    case 4: /* Analysis of selected nodes only */
        {
            parameters_type run;
```

```
         FILE *controlFile = openFile(args[3], "w");

         initializeDTARun(&run, args[1]);
         readCumulativeCounts(run.network, run.countsFileName);
         analyzeNode(&run, atof(args[2]) - 1);
         writeNode(controlFile, run.network, atof(args[2]) - 1);
         cleanUpDTARun(&run);
         fclose(controlFile);
         break;
      }
   default:
      displayUsage();
   }

   #ifdef DEBUG_MODE
      fclose(debugFile);
   #endif

   return EXIT_SUCCESS;
}

void displayUsage() {
   printf("Error in arguments -- two possible usages:\n");
   printf("\n");
   printf("Usage 1: To perform a complete run when there is no summary file already available:\n");
   printf("   warrant parametersFile networkFile initialICF finalICF\n");
   printf("      parametersFile - standard parameters file for executing calibrating run\n");
   printf("      networkFile - network file for executing calibration run\n");
   printf("      initialICF - file containing initial configuration; UNKNOWN marks intersections for analysis\n");
   printf("      finalICF - file to output final intersection configuration\n");
   printf("\n");
   printf("Usage 2: To perform a run of a single node when a summary file is already available:\n");
   printf("   warrant parametersFile nodeNumber outputFile\n");
   printf("      parametersFile - standard parameters file, with summary file indicated\n");
   printf("      nodeNumber - ID for the node to re-analyze");
   printf("      outputFile - name for file to write the new control data");
   exit(EXIT_FAILURE);
}
```

D.3 `main_warrant.h`

```c
/*
 Hierarchy of header files (bottom-up):

 utils.h
 datastructures.h
 sampling.h
 network.h
 cell.h
 vehicle.h
 node.h
 fileio.h
 dta.h
 warrant.h
 main.h

 Declarations referring to lower-level headers can use typedefs; declarations referring to higher-level headers must use structs
*/


#include <stdlib.h>
#include "cell.h"
#include "dta.h"
#include "fileio.h"
#include "utils.h"
#include "warrant.h"

void displayUsage();
```

D.3.1 `warrant.c`

```c
#include "warrant.h"

void generateBasicNodeControls(char *networkFileName, char *inputNodeControlFileName, char *outputNodeControlFileName) {
   network_type *network = newScalar(network_type);
   readNetworkFile(network, networkFileName, 1); /* Using dummy backward wave ratio; no simulation for the basic analysis.  (More sophisticated warrant analysis will change this.) */
   createStarLists(network);
  network->paths = createPathLinkedList();
  readNodeControlFile(network, inputNodeControlFileName);
   setAllNodesTo4WayStop(network);
   writeNodeControlFile(network, outputNodeControlFileName);
```

```c

    /* Memory cleanup (only what's allocated in the functions called above */
    deleteNetwork(network);
}

void generateWarrantNodeControls(char *parametersFileName, char *networkFileName, char *inputNodeControlFileName, char *outputNodeControlFileName) {
  parameters_type trialRun;

    /* Step 1: Generate temporary control file using basic controls */
    generateBasicNodeControls(networkFileName, inputNodeControlFileName, TEMP_INTERSECTION_FILENAME);

    /* Step 2: Create temporary parameters file: same as original parameters file but with temporary control file */
    generateTemporaryParametersFile(parametersFileName, TEMP_INTERSECTION_FILENAME, TEMP_PARAMETERS_FILENAME);

    /* Step 3: Perform a single run with temporary parameters file */
    initializeDTARun(&trialRun, TEMP_PARAMETERS_FILENAME);
    DTA(&trialRun);

    /* Step 4: Do warrant analysis */
    performWarrantAnalysis(&trialRun, inputNodeControlFileName);
    writeNodeControlFile(trialRun.network, outputNodeControlFileName);

    /* Step 5: Clean up by deleting temporary files */
    if (remove(TEMP_INTERSECTION_FILENAME)) fatalError("Unable to delete temporary control file %s\n", TEMP_INTERSECTION_FILENAME);
    if (remove(TEMP_PARAMETERS_FILENAME)) fatalError("Unable to delete temporary parameters file %s\n", TEMP_PARAMETERS_FILENAME);
  cleanUpDTARun(&trialRun);
}

void generateTemporaryParametersFile(char *parametersFileName, char *newControlFileName, char *newParametersFileName) {
    parameters_type tempRun;
     readParametersFile(&tempRun, parametersFileName);
    strncpy(tempRun.nodeControlFileName, newControlFileName, STRING_SIZE);
    writeParametersFile(&tempRun, newParametersFileName);
}

void performWarrantAnalysis(parameters_type *run, char *originalControlFileName) {
```

```
   int i;
   network_type *network = run->network;
   declareVector(bool, isUnknown, network->numNodes);

   /* Read original node controls to see which nodes need a warrant
      analysis conducted */
   scanControlFileForUnknown(originalControlFileName, isUnknown,
      network->numNodes);

   /* First ensure that all centroids are properly labeled */
   for (i = 0; i < network->numZones; i++) {
      network->node[i].control = CENTROID;
   }

   /* Now process all other nodes */
   for (; i < network->numNodes; i++) {
      if (isUnknown[i] == TRUE) {
         analyzeNode(run, i);
      }
   }

   deleteVector(isUnknown);
}

/* i is the node to analyze */
void analyzeNode(parameters_type *run, int i) {
   network_type *network = run->network;

   arc_type *majorApproach1 = NULL, *majorApproach2 = NULL, *
      minorApproach = NULL;
   /* First determine appropriate control type */
   classifyApproaches(&(network->node[i]), network->timeHorizon, &
      majorApproach1, &majorApproach2, &minorApproach);
   network->node[i].control = warrantedControl(&(network->node[i]),
      network->timeHorizon, run->timeHorizon, majorApproach1,
      majorApproach2, minorApproach);

   /* Then perform any additional analysis needed for that control type
       */
   switch (network->node[i].control) {
   case NONHOMOGENEOUS:
   case FOUR_WAY_STOP: /* Here no additional information is needed */
      break;
   case TWO_WAY_STOP: /* Here use angles to determine order of
      priorities */
      createTwoWayStop(&(network->node[i]), majorApproach1,
         majorApproach2);
```

```
      break;
   case BASIC_SIGNAL: /* Put basic signal timing here */
      createBasicSignal(&(network->node[i]), majorApproach1, majorApproach2, minorApproach, network->timeHorizon, run->timeHorizon);
      break;
   default:
      fatalError("Unknown return value %d from warrantedControl!", network->node[i].control);
   }
}


/*
Creates a "two-phase" signal.  Phase 1 is for majorApproach1 and majorApproach2.  Phase 2 is for minorApproach and everything else.
 1. For major approach: X factor is max(v/c for major approach 1 and major approach 2)
 2. For minor approach: X factor is v/c for minorApproach
 3. Calculate cycle length: C = min(5/(1 - X_maj - X_min), MAX_LENGTH)
 4. Then each phase's green time is proportionate X_maj / (X_maj + X-min) * C
This procedure IGNORES: clearance/lost time; saturation flow adjustments (including for turning); pedestrian crossing times
*/
void createBasicSignal(node_type *node, arc_type *majorApproach1, arc_type *majorApproach2, arc_type *minorApproach, int timeSteps, float timeHorizon) {
   int cycleLength, majorGreenTime, minorGreenTime;
   float majorSaturation, minorSaturation;
   double majorApproach1volume, majorApproach2volume, minorApproachVolume;
   basicSignal_type *basicSignalControl;
   arcLinkedListElt *upstreamArc, *downstreamArc;

   displayMessage(DEBUG, "Creating basic signal for node %d\n", node->ID);

   /* 1. If there are 2 approaches, make the second "major approach" the minor approach */
   if (minorApproach == NULL) {
      majorApproach2 = majorApproach1;
      minorApproach = majorApproach2;
   }

   /* 2. Determine critical lane volumes, converting units as appropriate */
```

```
majorApproach1volume = majorApproach1->downstreamCount[timeSteps - 1] / timeHorizon;
majorApproach2volume = majorApproach2->downstreamCount[timeSteps - 1] / timeHorizon;
minorApproachVolume = minorApproach->downstreamCount[timeSteps - 1] / timeHorizon;
majorSaturation = max(majorApproach1volume / majorApproach1->capacity, majorApproach2volume / majorApproach2->capacity);
minorSaturation = minorApproachVolume / minorApproach->capacity;
displayMessage(DEBUG, "Major and minor saturation levels are %f and %f\n", majorSaturation, minorSaturation);

/* 3. Calculate cycle length using Webster's formula */
displayMessage(DEBUG, "Webster cycle length for this signal is %f\n", 5.0 / (1 - majorSaturation - minorSaturation));
if (majorSaturation + minorSaturation > 1 - 5.0 / MAX_CYCLE_LENGTH) {
   cycleLength = MAX_CYCLE_LENGTH;
} else {
   cycleLength = max(MIN_CYCLE_LENGTH, rint(5 / (1 - majorSaturation - minorSaturation)));
}

/* 4. Allocate green times */
majorGreenTime = (majorSaturation / (majorSaturation + minorSaturation)) * cycleLength;
minorGreenTime = cycleLength - majorGreenTime;

/* 5. Set up signal data structure; note that green times are entered in the same order as in createAllPossibleMovements */
node->controlData = newScalar(basicSignal_type);
basicSignalControl = (basicSignal_type *)(node->controlData);
basicSignalControl->cycleLength = cycleLength;
basicSignalControl->greenTime = createLinkedList();
for (upstreamArc = node->reverseStar->head; upstreamArc != NULL; upstreamArc = upstreamArc->next) {
   for (downstreamArc = node->forwardStar->head; downstreamArc != NULL; downstreamArc = downstreamArc->next) {
      if (downstreamArc->arc->head->ID == upstreamArc->arc->tail->ID) continue; /* Skip U-turns */
      if (upstreamArc->arc == majorApproach1 || upstreamArc->arc == majorApproach2) {
         insertLinkedList(basicSignalControl->greenTime, majorGreenTime, NULL);
      } else {
         insertLinkedList(basicSignalControl->greenTime, minorGreenTime, NULL);
```

```
        }
      }
    }
}

/*
Two-way stop: tier 3.  tier 1 is a right turn or through from major; tier 2 is a let turn from major; tier 3 is anything else
*/
void createTwoWayStop(node_type *node, arc_type *majorApproach1, arc_type *majorApproach2) {
    float majorAngle1, majorAngle2, trialAngle;
    priorityLinkedListElt *priority1, *priority2, *priority3;
    twoWayStop_type *stopData;
    turningLinkedListElt *curMovement;
    arcLinkedListElt *upstreamArc, *downstreamArc;

    /* 1. Determine angles */
    majorAngle1 = atan2(majorApproach1->tail->Y - node->Y, majorApproach1->tail->X - node->X);
    majorAngle2 = atan2(majorApproach2->tail->Y - node->Y, majorApproach2->tail->X - node->X);

    /* 2. Set up stop data structure */
    node->controlData = newScalar(twoWayStop_type);
    stopData = (twoWayStop_type *)(node->controlData);
   stopData->minStopPriority = 3;
    stopData->priorityList = createPriorityLinkedList();
    priority1 = insertPriorityLinkedList(stopData->priorityList, 1, NULL);
    priority2 = insertPriorityLinkedList(stopData->priorityList, 2, priority1);
    priority3 = insertPriorityLinkedList(stopData->priorityList, 3, priority2);
    stopData->saturationFlow = majorApproach1->capacity;

    /* 3. Assign each movement to an appropriate priority list:
        major approach through/right = 1; major approach left = 2; everything else = 3
        Cycles through list in the same order as movements are created in createAllPossibleMovements
    */
    curMovement = node->turnMovements->head;
    for (upstreamArc = node->reverseStar->head; upstreamArc != NULL; upstreamArc = upstreamArc->next) {
        for (downstreamArc = node->forwardStar->head; downstreamArc != NULL; downstreamArc = downstreamArc->next) {
```

```
if (downstreamArc->arc->head->ID == upstreamArc->arc->tail->ID
   ) continue; /* Skip U-turns */
if (upstreamArc->arc == majorApproach1) {
   trialAngle = atan2(downstreamArc->arc->head->Y - node->Y,
      downstreamArc->arc->head->X - node->X);
   if (majorAngle1 <= majorAngle2) {
      if (majorAngle1 <= trialAngle && trialAngle <=
         majorAngle2) { /* Right turn */
         insertTurningLinkedList(priority1->movements,
            curMovement->movement, NULL);
      } else {

         /* Left turn */
         insertTurningLinkedList(priority2->movements,
            curMovement->movement, NULL);
      }
   } else {
      if (majorAngle2 <= trialAngle && trialAngle <=
         majorAngle1) {  /* Left turn */
         insertTurningLinkedList(priority2->movements,
            curMovement->movement, NULL);
      } else {

         /* Right turn */
         insertTurningLinkedList(priority1->movements,
            curMovement->movement, NULL);
      }
   }
} else if (upstreamArc->arc == majorApproach2) {
   trialAngle = atan2(downstreamArc->arc->head->Y - node->Y,
      downstreamArc->arc->head->X - node->X);
   if (majorAngle2 <= majorAngle1) {
      if (majorAngle2 <= trialAngle && trialAngle <=
         majorAngle1) { /* Right turn */
         insertTurningLinkedList(priority1->movements,
            curMovement->movement, NULL);
      } else {

         /* Left turn */
         insertTurningLinkedList(priority2->movements,
            curMovement->movement, NULL);
      }
   } else {
      if (majorAngle1 <= trialAngle && trialAngle <=
         majorAngle2) {  /* Left turn */
         insertTurningLinkedList(priority2->movements,
            curMovement->movement, NULL);
```

```c
                    } else {

                        /* Right turn */
                        insertTurningLinkedList(priority1->movements, curMovement->movement, NULL);
                    }
                }
            } else {
                insertTurningLinkedList(priority3->movements, curMovement->movement, NULL);
            }
            curMovement = curMovement->next;
        }
    }
}


void classifyApproaches (node_type *node, int timeSteps, arc_type **majorApproach1, arc_type **majorApproach2, arc_type **minorApproach) {
    double majorApproach1volume, majorApproach2volume, minorApproachVolume, curVolume;
    arcLinkedListElt *curArc;

    /* Initialize search */
    *majorApproach1 = NULL;
    *majorApproach2 = NULL;
    *minorApproach = NULL;
    majorApproach1volume = -INFINITY;
    majorApproach2volume = -INFINITY;
    minorApproachVolume = -INFINITY;

    for (curArc = node->reverseStar->head; curArc != NULL; curArc = curArc->next) {
        curVolume = curArc->arc->downstreamCount[timeSteps - 1];
        if (curVolume > majorApproach1volume) {
            minorApproachVolume = majorApproach2volume;
            majorApproach2volume = majorApproach1volume;
            majorApproach1volume = curVolume;
            *minorApproach = *majorApproach2;
            *majorApproach2 = *majorApproach1;
            *majorApproach1 = curArc->arc;
        } else if (curVolume > majorApproach2volume) {
            minorApproachVolume = majorApproach2volume;
            majorApproach2volume = curVolume;
            *minorApproach = *majorApproach2;
            *majorApproach2 = curArc->arc;
```

```c
        } else if (curVolume > minorApproachVolume) {
            minorApproachVolume = curVolume;
            *minorApproach = curArc->arc;
        }
    }

    /* Avoid compiler warning about parameter 'minorApproach' set but not used (the pointer is used by another function) */
    if (*minorApproach == NULL) return;
}

/* Use volumes to classify approaches, and thereby determine appropriate control type */
intersection_type warrantedControl(node_type *node, int timeSteps, float timeHorizon, arc_type *majorApproach1, arc_type *majorApproach2, arc_type *minorApproach) {
    int majorApproachLanes, minorApproachLanes;
    double majorApproachVolume, minorApproachVolume, majorApproachFFS;

    /* First handle easy cases */
    if (node->reverseStar->size <= 0 || node->forwardStar->size <= 0) {
        fatalError("Non-centroid node %d either has no approaches or no exits.", node->ID);
    } else if (node->reverseStar->size == 1) {
        if (node->forwardStar->size == 1) {
            return NONHOMOGENEOUS;
        } else {
            return DIVERGE;
        }
    } else if (node->forwardStar->size == 1) {
        return MERGE;
    }

    /* Need to add MERGE and DIVERGE as well */
    if (majorApproach2 == NULL) return NONHOMOGENEOUS;

    /* If there are 2 approaches, make the second "major approach" the minor approach */
    if (minorApproach == NULL) {
        majorApproachVolume = majorApproach1->downstreamCount[timeSteps - 1];
        minorApproachVolume = majorApproach2->downstreamCount[timeSteps - 1];
        majorApproachLanes = rint(majorApproach1->capacity / (BASE_SATURATION_FLOW / HOURS));
        minorApproachLanes = rint(majorApproach2->capacity / (BASE_SATURATION_FLOW / HOURS));
```

```
      majorApproachFFS = majorApproach1->length / majorApproach1->freeFlowTime;
   } else { /* Usual case */
      majorApproachVolume = max(majorApproach1->downstreamCount[timeSteps - 1], majorApproach2->downstreamCount[timeSteps - 1]);
      minorApproachVolume = minorApproach->downstreamCount[timeSteps - 1];
      majorApproachLanes = rint(max(majorApproach1->capacity, majorApproach2->capacity) / (BASE_SATURATION_FLOW / HOURS));
      minorApproachLanes = rint(minorApproach->capacity / (BASE_SATURATION_FLOW / HOURS));
      majorApproachFFS = max(majorApproach1->length / majorApproach1->freeFlowTime, majorApproach2->length/ majorApproach2->freeFlowTime);
   }

   /* Convert parameters to hourly units for MUTCD */
   majorApproachVolume *= (HOURS / timeHorizon);
   minorApproachVolume *= (HOURS / timeHorizon);
   majorApproachFFS /= (MILES / HOURS);
   displayMessage(DEBUG, "Major and minor approach volumes are %f and %f\n", majorApproachVolume, minorApproachVolume);

   /* Now do warrant analysis based on volumes */
   if (signalWarranted(majorApproachLanes, minorApproachLanes, majorApproachVolume, minorApproachVolume, majorApproachFFS) == TRUE) return BASIC_SIGNAL;
   if (fourWayStopWarranted(majorApproachVolume, minorApproachVolume) == TRUE) return FOUR_WAY_STOP;
   if (twoWayStopWarranted(majorApproachVolume) == TRUE) return TWO_WAY_STOP;

   return FOUR_WAY_STOP; /* Default control */
}

/* Based on MUTCD volume warrant  */
bool signalWarranted(int majorLanes, int minorLanes, double majorVolume, double minorVolume, double majorApproachFFS) {
   if (minorLanes <= 1) {
      if (majorLanes <= 1) {
         if (majorVolume > 500 && minorVolume > 150) return TRUE;
         if (majorVolume > 750 && minorVolume > 75) return TRUE;
         if (majorVolume > 600 && minorVolume > 120) return TRUE;

         if (majorApproachFFS <= 40) return FALSE;

```

```
         if (majorVolume > 350 && minorVolume > 105) return TRUE;
         if (majorVolume > 525 && minorVolume > 53) return TRUE;
         if (majorVolume > 420 && minorVolume > 84) return TRUE;
         return FALSE;
      } else { /* More than one major lane */
         if (majorVolume > 600 && minorVolume > 150) return TRUE;
         if (majorVolume > 900 && minorVolume > 75) return TRUE;
         if (majorVolume > 720 && minorVolume > 120) return TRUE;

         if (majorApproachFFS <= 40) return FALSE;

         if (majorVolume > 420 && minorVolume > 105) return TRUE;
         if (majorVolume > 630 && minorVolume > 53) return TRUE;
         if (majorVolume > 504 && minorVolume > 84) return TRUE;
         return FALSE;
      }
   } else { /* More than one minor lane */
      if (majorLanes <= 1) {
         if (majorVolume > 500 && minorVolume > 200) return TRUE;
         if (majorVolume > 750 && minorVolume > 100) return TRUE;
         if (majorVolume > 600 && minorVolume > 160) return TRUE;

         if (majorApproachFFS <= 40) return FALSE;

         if (majorVolume > 350 && minorVolume > 140) return TRUE;
         if (majorVolume > 525 && minorVolume > 70) return TRUE;
         if (majorVolume > 420 && minorVolume > 112) return TRUE;
         return FALSE;
      } else { /* More than one major lane */
         if (majorVolume > 600 && minorVolume > 200) return TRUE;
         if (majorVolume > 900 && minorVolume > 100) return TRUE;
         if (majorVolume > 720 && minorVolume > 160) return TRUE;

         if (majorApproachFFS <= 40) return FALSE;

         if (majorVolume > 420 && minorVolume > 140) return TRUE;
         if (majorVolume > 630 && minorVolume > 70) return TRUE;
         if (majorVolume > 504 && minorVolume > 112) return TRUE;
         return FALSE;
      }
   }
}

bool fourWayStopWarranted(double majorVolume, double minorVolume) {
   if (majorVolume > 300 && minorVolume > 200) {
      return TRUE;
   } else {
```

```
      return FALSE;
   }
}

bool twoWayStopWarranted(double majorVolume) {
   if (majorVolume > 750) {
      return TRUE;
   } else {
      return FALSE;
   }
}

void setAllNodesTo4WayStop(network_type *network) {
   int i;

   /* Take care of centroids... */
   for (i = 0; i < network->numZones; i++) {
      network->node[i].control = CENTROID;
   }

   /* Then generate all other possible movements, without creating U-turns */
   for (; i < network->numNodes; i++) {
      if (network->node[i].control != UNKNOWN_CONTROL) continue; /* Don't disturb known intersections */
      network->node[i].control = FOUR_WAY_STOP;
      createAllPossibleMovements(&(network->node[i]));
   }

}

void createAllPossibleMovements(node_type *node) {
   arcLinkedListElt *upstreamArc, *downstreamArc;
   turning_type *newMovement;

   for (upstreamArc = node->reverseStar->head; upstreamArc != NULL; upstreamArc = upstreamArc->next) {
      for (downstreamArc = node->forwardStar->head; downstreamArc != NULL; downstreamArc = downstreamArc->next) {
         if (downstreamArc->arc->head->ID == upstreamArc->arc->tail->ID) continue; /* Skip U-turns */
         newMovement = newScalar(turning_type);
         createMovement(newMovement, upstreamArc->arc->tail->ID, node->ID, downstreamArc->arc->head->ID, node);
         newMovement->saturationFlow = upstreamArc->arc->capacity;
      }
   }
```

```
}

void scanControlFileForUnknown(char *controlFileName, bool *isUnknown, int numNodes) {
   int i, status;
   char fullLine[STRING_SIZE], trimmedLine[STRING_SIZE], controlText[STRING_SIZE];
   FILE *controlFile = openFile(controlFileName, "r");

   while (!feof(controlFile)) {
    if (fgets(fullLine, STRING_SIZE, controlFile) == NULL) break;
    status = parseLine(fullLine, trimmedLine);
    if (status == BLANK_LINE || status == COMMENT) continue;
    if (strncmp(trimmedLine, "Node", 4) == 0) {
      sscanf(trimmedLine, "Node %d : %s", &i, controlText);
      }
    if (i < 1 || i > numNodes) {
         warning(FULL_NOTIFICATIONS, "Node out of range in control file.  Ignoring input line:\n%s\n", fullLine);
         continue;
    }
    if (strcmp(controlText, "UNKNOWN") == 0) {
         isUnknown[i-1] = TRUE;
      } else {
         isUnknown[i-1] = FALSE;
      }
   }
   fclose(controlFile);
}
```

D.3.2 `warrant.h`

```
#ifndef _WARRANT_H_
#define _WARRANT_H_

#include "dta.h"
#include "fileio.h"
#include "network.h"
#include "datastructures.h"
#include "utils.h"
#include <math.h>

#define BASE_SATURATION_FLOW 1900

#define MIN_CYCLE_LENGTH 20
#define MAX_CYCLE_LENGTH 120

```

```
#define TEMP_INTERSECTION_FILENAME "~TEMP.ICF"
#define TEMP_PARAMETERS_FILENAME "~TEMP_PARAMETERS.TXT"

void generateBasicNodeControls(char *networkFileName, char *inputNodeControlFileName, char *outputNodeControlFileName);
void generateWarrantNodeControls(char *parametersFileName, char *networkFileName, char *inputNodeControlFileName, char *outputNodeControlFileName);
void generateTemporaryParametersFile(char *parametersFileName, char *newControlFileName, char *newParametersFileName);

void classifyApproaches (node_type *node, int timeSteps, arc_type **majorApproach1, arc_type **majorApproach2, arc_type **minorApproach);
void performWarrantAnalysis(parameters_type *run, char *originalControlFileName);
intersection_type warrantedControl(node_type *node, int timeSteps, float timeHorizon, arc_type *majorApproach1, arc_type *majorApproach2, arc_type *minorApproach);

bool signalWarranted(int majorLanes, int minorLanes, double majorVolume, double minorVolume, double majorApproachFFS);
bool fourWayStopWarranted(double majorVolume, double minorVolume);
bool twoWayStopWarranted(double majorVolume);

void createBasicSignal(node_type *node, arc_type *majorApproach1, arc_type *majorApproach2, arc_type *minorApproach, int timeSteps, float timeHorizon);
void createTwoWayStop(node_type *node, arc_type *majorApproach1, arc_type *majorApproach2);

void setAllNodesTo4WayStop(network_type *network);
void createAllPossibleMovements(node_type *node);
void scanControlFileForUnknown(char *controlFileName, bool *isUnknown, int numNodes);

void analyzeNode(parameters_type *run, int i);
#endif
```

D.4 Graphics module

D.4.1 main_graphics.c

```
#include "main_graphics.h"

```

```
/* Primary main file for running simulation.  Uncomment #if 0 and #endif to compile this one. */
int main(int numArgs, char *args[]) {


  parameters_type run;
   graphicsParameters_type graphicsParameters;

   /* int t; */
   verbosity = FULL_NOTIFICATIONS;

   #ifdef DEBUG_MODE /* Debug mode enables extra logging.  Define this macro in utils.h */
      debugFile = openFile("logfile.txt", "w");
      /* verbosity = FULL_DEBUG; */
      displayMessage(DEBUG, "Starting new run.\n");
   #endif

   if (numArgs != 2) displayUsage();

  initializeDTARun(&run, args[1]);
   initializeGraphics(&graphicsParameters, &run);
   readCumulativeCounts(run.network, run.countsFileName);

   if (graphicsParameters.snapshotMode == TRUE) {
      for (t = 0; t < run.network->timeHorizon; t++) {
         generateBitmap(&graphicsParameters, run.network, t);
      }
   } else {
      generateFinalBitmap(&graphicsParameters, &run);
   }

  cleanUpDTARun(&run);
   cleanUpGraphics(&graphicsParameters);

   #ifdef DEBUG_MODE
      fclose(debugFile);
   #endif

  return (EXIT_SUCCESS);
}

void displayUsage() {
   fatalError("Program requires exactly one argument - run parameters file.");
}
```

D.5 main_graphics.h

```
/*
 Hierarchy of header files (bottom-up):

 utils.h
 datastructures.h
 sampling.h
 network.h
 cell.h
 vehicle.h
 node.h
 fileio.h
 dta.h
 main.h

 Declarations referring to lower-level headers can use typedefs;
    declarations referring to higher-level headers must use structs
*/


#include <stdlib.h>
#include "cell.h"
#include "fileio.h"
#include "utils.h"
#include "graphics.h"
#include "warrant.h"

void displayUsage();
```

D.5.1 graphics.c

```
#include "graphics.h"
#include "characters.h"

/******************************
 ** Image conversion routines **
 ******************************/

void absolute2relative(graphicsParameters_type *graphicsParameters,
    float absoluteX, float absoluteY, int *relativeX, int *relativeY,
    int minX, int minY) {
  *relativeX = graphicsParameters->borderWidth + graphicsParameters->
     imageWidth  * ((absoluteX - minX) / graphicsParameters->
     absoluteXrange);
  *relativeY = graphicsParameters->borderWidth + graphicsParameters->
```

```
    imageHeight * (1 - (absoluteY - minY) / graphicsParameters->absoluteYrange);

}

int density2red(int density, int jamDensity) {
  return density * 255 / jamDensity;
}

int density2green(int density, int jamDensity) {
  return (jamDensity - density) * 255 / jamDensity;
}

int density2blue(int density, int jamDensity) {
  return (density > 0) ? 0 : 128;
  return 0;
  return 0 * (density + jamDensity); /* Never reached, but silences compiler warnings about unused arguments. */
}

#if 0
void generateFinalNodeBitmap(graphicsParameters_type * graphicsParameters, parameters_type *run, node_type *node) {
   int i, ij;
   char filename[STRING_SIZE];
   network_type *network = run->network;
   bitmap_type bitmap;

   displayMessage(FULL_NOTIFICATIONS, "Writing image file ");
   bitmap.height = graphicsParameters->imageHeight + 2 * graphicsParameters->borderWidth;
   bitmap.width = graphicsParameters->imageWidth + 2 * graphicsParameters->borderWidth;
   bitmap.pixels = newVector(bitmap.height * bitmap.width, pixel_type);
   resetBitmap(&bitmap, 0, 0, 0);

   /* Draw arcs adjacent only to a single node */
   for (ij = 0; ij < network->numArcs; ij++) {
      if (network->arc[ij].head == node ) {
         drawOverallArc(&bitmap, graphicsParameters, ij, run);
         drawNode(&bitmap, graphicsParameters, ptr2node(network->arc[ij].tail));
      }
      if (network->arc[ij].tail == node) {
         drawOverallArc(&bitmap, graphicsParameters, ij, run);
         drawNode(&bitmap, graphicsParameters, ptr2node(network->arc[ij].head));
```

```
      }
   }

   /* Write file */
   sprintf(filename, "%s_node%d.png", graphicsParameters->graphicsRoot, node->ID);
   writePNG(&bitmap, filename);
   deleteVector(bitmap.pixels);
   displayMessage(FULL_NOTIFICATIONS, "%s complete.\n", filename);
}
#endif

#define PLACEHOLDER_TIME 0
void generateFinalBitmap(graphicsParameters_type *graphicsParameters, parameters_type *run) {
   int i, ij, t;
   int oldUpstream, oldDownstream;
   int upstream, downstream;
   char filename[STRING_SIZE];
   int startTime = run->warmUpLength / run->tickLength, endTime = (run->timeHorizon - run->coolDownLength) / run->tickLength;
  int numPeriods = endTime - startTime;
  network_type *network = run->network;
   bitmap_type bitmap;


   displayMessage(FULL_NOTIFICATIONS, "Writing image file ");
  if (numPeriods < 1) {
     warning(LOW_NOTIFICATIONS, "Can't generate final bitmap file, entire run is warm-up or cool-down.\n");
     return;
  }

   bitmap.height = graphicsParameters->imageHeight + 2 * graphicsParameters->borderWidth;
   bitmap.width = graphicsParameters->imageWidth + 2 * graphicsParameters->borderWidth;
   bitmap.pixels = newVector(bitmap.height * bitmap.width, pixel_type);
   resetBitmap(&bitmap, 0, 0, 0);


   /* Draw arcs */
   for (ij = 0; ij < network->numArcs; ij++) {
      oldUpstream = network->arc[ij].upstreamCount[PLACEHOLDER_TIME];
      oldDownstream = network->arc[ij].downstreamCount[PLACEHOLDER_TIME];
      upstream = 0; downstream = 0;
```

```
      for (t = startTime; t < endTime; t++) {
         upstream += network->arc[ij].upstreamCount[t];
         downstream += network->arc[ij].downstreamCount[t];
      }
      network->arc[ij].upstreamCount[PLACEHOLDER_TIME] = upstream /
         numPeriods;
      network->arc[ij].downstreamCount[PLACEHOLDER_TIME] = downstream /
          numPeriods;
      drawArc(&bitmap, graphicsParameters, ij, network,
         PLACEHOLDER_TIME);
      network->arc[ij].upstreamCount[PLACEHOLDER_TIME] = oldUpstream;
      network->arc[ij].downstreamCount[PLACEHOLDER_TIME] =
         oldDownstream;
   }

   /* Draw nodes */
   for (i = network->numZones; i < network->numNodes; i++) {
      drawNode(&bitmap, graphicsParameters, i);
   }

   /* Write file */
   sprintf(filename, "%s_final.png", graphicsParameters->graphicsRoot);
   writePNG(&bitmap, filename);
   deleteVector(bitmap.pixels);
   displayMessage(FULL_NOTIFICATIONS, "%s complete.\n", filename);
}

void generateBitmap(graphicsParameters_type *graphicsParameters,
   network_type *network, int t) {
   int i, ij, numDigits = ceil(log10(network->timeHorizon));
   char filename[STRING_SIZE], formatString[STRING_SIZE];
   bitmap_type bitmap;

   displayMessage(FULL_NOTIFICATIONS, "Writing image file ");
   bitmap.height = graphicsParameters->imageHeight + 2 *
      graphicsParameters->borderWidth;
   bitmap.width = graphicsParameters->imageWidth + 2 *
      graphicsParameters->borderWidth;
   bitmap.pixels = newVector(bitmap.height * bitmap.width, pixel_type);
   resetBitmap(&bitmap, 127, 127, 127);

   /* Draw arcs */
   for (ij = 0; ij < network->numArcs; ij++) {
      if (network->arc[ij].head->controlType != CENTROID && network->
         arc[ij].tail->controlType != CENTROID)
         drawArc(&bitmap, graphicsParameters, ij, network, t);
   }
```

```c

   /* Draw nodes */
   for (i = network->numZones; i < network->numNodes; i++) {
      drawNode(&bitmap, graphicsParameters, i);
   }

   /* Write file */
   sprintf(formatString, "%%s%%0%dd.png", numDigits);
   sprintf(filename, formatString, graphicsParameters->graphicsRoot, t);
   writePNG(&bitmap, filename);
   deleteVector(bitmap.pixels);
   displayMessage(FULL_NOTIFICATIONS, "%s complete.\n", filename);
}

/***** Drawing routines *****/

void drawArc(bitmap_type *bitmap, graphicsParameters_type * graphicsParameters, int ij, network_type *network, int t) {
   int x, y, baseX, baseY;
   int tail = ptr2node(network, network->arc[ij].tail);
   int head = ptr2node(network, network->arc[ij].head);
   int red, green, blue;
   int numVehicles = network->arc[ij].upstreamCount[t] - network->arc[ij].downstreamCount[t];
   int maxVehicles = network->arc[ij].jamDensity * network->arc[ij].length;

   displayMessage(DEBUG, "Drawing arc (%d,%d)\n", network->arc[ij].tail->ID, network->arc[ij].head->ID);
   /* Identify colors */
   if (maxVehicles == 0) { /* Shade white for links with no allowable vehicles */
      red = 127;
      green = 127;
      blue = 127;
   } else {
      red = density2red(numVehicles, maxVehicles);
      green = density2green(numVehicles, maxVehicles);
      blue = density2blue(numVehicles, maxVehicles);
   }

   switch(graphicsParameters->arcSlope[ij]) {
   case SHALLOW: /* Iterate over x direction, finding appropriate y. arcRelativeDX must be nonzero */
      if (graphicsParameters->arcRelativeDX[ij] > 0) { /* Arc moves from left-to-right, draw on bottom */
```

```
            for (x = BOTTOM_RIGHT_X(tail); x <= BOTTOM_LEFT_X(head); x++)
               {
               if (BOTTOM_LEFT_X(head) == BOTTOM_RIGHT_X(tail)) continue;
               baseY = BOTTOM_RIGHT_Y(tail) + (x - BOTTOM_RIGHT_X(tail)) *
                    (BOTTOM_LEFT_Y(head) - BOTTOM_RIGHT_Y(tail)) / (
                  BOTTOM_LEFT_X(head) - BOTTOM_RIGHT_X(tail));
               for (y = baseY; y >= baseY - graphicsParameters->linkWidth;
                   y--) {
                  setPixel(bitmap, x, y, red, green, blue);
               }
            }
         } else { /* Arc moves from right-to-left, draw on top */
            for (x = TOP_LEFT_X(tail); x >= TOP_RIGHT_X(head); x--) {
               if (TOP_RIGHT_X(head) == TOP_LEFT_X(tail)) continue;
               baseY = TOP_LEFT_Y(tail) + (x - TOP_LEFT_X(tail)) * (
                  TOP_RIGHT_Y(head) - TOP_LEFT_Y(tail)) / (TOP_RIGHT_X(
                  head) - TOP_LEFT_X(tail));
               for (y = baseY; y <= baseY + graphicsParameters->linkWidth;
                   y++) {
                  setPixel(bitmap, x, y, red, green, blue);
               }
            }
         }
         break;
      case STEEP: /* Iterative over y direction, finding appropriate x.
         arcRelativeDY must be nonzero */
         if (graphicsParameters->arcRelativeDY[ij] < 0) { /* Arc moves
            from bottom-to-top, draw on right */
            for (y = TOP_RIGHT_Y(tail); y >= BOTTOM_RIGHT_Y(head); y--) {
               if (BOTTOM_RIGHT_Y(head) == TOP_RIGHT_Y(tail)) continue;
               baseX = TOP_RIGHT_X(tail) + (y - TOP_RIGHT_Y(tail)) * (
                  BOTTOM_RIGHT_X(head) - TOP_RIGHT_X(tail)) / (
                  BOTTOM_RIGHT_Y(head) - TOP_RIGHT_Y(tail));
               for (x = baseX; x >= baseX - graphicsParameters->linkWidth;
                   x--) {
                  setPixel(bitmap, x, y, red, green, blue);
               }
            }
         } else { /* Arc moves from top-to-bottom, draw on left */
            for (y = BOTTOM_LEFT_Y(tail); y <= TOP_LEFT_Y(head); y++) {
               if (TOP_LEFT_Y(head) == BOTTOM_LEFT_Y(tail)) continue;
               baseX = BOTTOM_LEFT_X(tail) + (y - BOTTOM_LEFT_Y(tail)) * (
                  TOP_LEFT_X(head) - BOTTOM_LEFT_X(tail)) / (TOP_LEFT_Y(
                  head) - BOTTOM_LEFT_Y(tail));
               for (x = baseX; x <= baseX + graphicsParameters->linkWidth;
                   x++) {
                  setPixel(bitmap, x, y, red, green, blue);
```

```
            }
         }
      }
      break;
   default:
      fatalError("Unknown arc slope!");
   }
}

void drawNode(bitmap_type *bitmap, graphicsParameters_type * graphicsParameters, int i) {
   int x, y, nodeRadius = graphicsParameters->nodeRadius;
   for (x = graphicsParameters->relativeNodeX[i] - nodeRadius; x <= graphicsParameters->relativeNodeX[i] + nodeRadius; x++) {
      for (y = graphicsParameters->relativeNodeY[i] - nodeRadius; y <= graphicsParameters->relativeNodeY[i] + nodeRadius; y++) {
         setPixel(bitmap, x, y, 255, 255, 255);
      }
   }
   plopLabel(bitmap, i+1, BOTTOM_LEFT_X(i), BOTTOM_LEFT_Y(i) + graphicsParameters->nodeRadius, 255, 0, 0);
}

void plopLabel(bitmap_type *bitmap, long label, int upperLeftX, int upperLeftY, int red, int green, int blue) {
   int ptr = 0, digit;
   int x = upperLeftX;
   char buffer[STRING_SIZE];
   sprintf(buffer, "%ld", label);
   while (buffer[ptr] != '\0') {
      digit = buffer[ptr] - '0'; /* Assumes character encoding has numbers sequentially after 0 */
      plopDigit(bitmap, digit, x, upperLeftY, red, green, blue);
      x += CHAR_WIDTH + 1;
      ptr++;
   }
}

void plopDigit(bitmap_type *bitmap, int digit, int upperLeftX, int upperLeftY, int red, int green, int blue) {
   int x, y;
   for (x = 0; x < CHAR_WIDTH; x++) {
      for (y = 0; y < CHAR_HEIGHT; y++) {
         if (digitFont[digit][y][x] != 0) setPixel(bitmap, x + upperLeftX, y + upperLeftY, red, green, blue);
      }
   }
```

```c
}

void resetBitmap(bitmap_type *bitmap, unsigned char red, unsigned char green, unsigned char blue) {
  int x, y;
  for (y = 0; y < bitmap->height; y++) {
    for (x = 0; x < bitmap->width; x++) {
      setPixel(bitmap, x, y, red, green, blue);
    }
  }
}

void setPixel(bitmap_type *bitmap, int x, int y, unsigned char red, unsigned char green, unsigned char blue) {
  pixel_type *pixel = coord2pixel(bitmap, x, y);
  if (x < 0 || x >= bitmap->width || y < 0 || y >= bitmap->height) return;

  pixel->red = red;
  pixel->green = green;
  pixel->blue = blue;
}

/***** Handle graphics data structures *****/

void cleanUpGraphics(graphicsParameters_type *graphicsParameters) {
  deleteVector(graphicsParameters->relativeNodeX);
  deleteVector(graphicsParameters->relativeNodeY);
  deleteVector(graphicsParameters->arcRelativeDX);
  deleteVector(graphicsParameters->arcRelativeDY);
  deleteVector(graphicsParameters->arcSlope);
}

void initializeGraphics(graphicsParameters_type *graphicsParameters, parameters_type *run) {
  int i, ij;
  float minX = INFINITY, maxX = -INFINITY, minY = INFINITY, maxY = -INFINITY;
   network_type *network = run->network;

   if (strlen(run->graphicsFileName) == 0) fatalError("Missing graphics parameter file!");
   readGraphicsParametersFile(graphicsParameters, run->graphicsFileName);

  graphicsParameters->relativeNodeX = newVector(network->numNodes, int);
```

```c
  graphicsParameters->relativeNodeY = newVector(network->numNodes, int);
  graphicsParameters->arcRelativeDX = newVector(network->numArcs, int);
  graphicsParameters->arcRelativeDY = newVector(network->numArcs, int);
  graphicsParameters->arcSlope = newVector(network->numArcs, slope_type);

  for (i = 0; i < network->numNodes; i++) {
    minX = min(minX, network->node[i].X);
    maxX = max(maxX, network->node[i].X);
    minY = min(minY, network->node[i].Y);
    maxY = max(maxY, network->node[i].Y);
  }
  graphicsParameters->absoluteXrange = maxX - minX + 1; /* Add one to ensure absolute ranges are at least 1 */
  graphicsParameters->absoluteYrange = maxY - minY + 1;

  for (i = 0; i < network->numNodes; i++) {
     absolute2relative(graphicsParameters, network->node[i].X, network->node[i].Y, &(graphicsParameters->relativeNodeX[i]), &(graphicsParameters->relativeNodeY[i]), minX, minY);
  }

  for (ij = 0; ij < network->numArcs; ij++) {
     graphicsParameters->arcRelativeDX[ij] = graphicsParameters->relativeNodeX[network->arc[ij].head->ID-1] - graphicsParameters->relativeNodeX[network->arc[ij].tail->ID-1];
     graphicsParameters->arcRelativeDY[ij] = graphicsParameters->relativeNodeY[network->arc[ij].head->ID-1] - graphicsParameters->relativeNodeY[network->arc[ij].tail->ID-1];
     if (abs(graphicsParameters->arcRelativeDX[ij]) > abs(graphicsParameters->arcRelativeDY[ij]))
         graphicsParameters->arcSlope[ij] = SHALLOW;
      else
         graphicsParameters->arcSlope[ij] = STEEP;
  }

}

void readGraphicsParametersFile(graphicsParameters_type *graphicsParameters, char *graphicsParametersFileName) {
  int status;
  char fullLine[STRING_SIZE];
  char metadataTag[STRING_SIZE], metadataValue[STRING_SIZE];
  FILE *graphicsParametersFile = openFile(graphicsParametersFileName, "r");

```

```
/* Set default parameter values */
graphicsParameters->imageWidth = 500;
graphicsParameters->imageHeight = 500;
graphicsParameters->borderWidth = 50;
graphicsParameters->nodeRadius = 5;
graphicsParameters->linkWidth = 2;
graphicsParameters->graphicsRoot[0] = '\0';
graphicsParameters->snapshotMode = FALSE;

/* Process parameter file */
while (!feof(graphicsParametersFile)) {
  do {
    if (fgets(fullLine, STRING_SIZE, graphicsParametersFile) == NULL) break;
    status = parseMetadata(fullLine, metadataTag, metadataValue);
  } while (status == BLANK_LINE || status == COMMENT);
  if        (strcmp(metadataTag, "IMAGE WIDTH") == 0) {
    graphicsParameters->imageWidth = atoi(metadataValue);
  } else if (strcmp(metadataTag, "IMAGE HEIGHT") == 0) {
    graphicsParameters->imageHeight = atoi(metadataValue);
  } else if (strcmp(metadataTag, "BORDER WIDTH") == 0) {
    graphicsParameters->borderWidth = atoi(metadataValue);
  } else if (strcmp(metadataTag, "NODE RADIUS") == 0) {
    graphicsParameters->nodeRadius = atoi(metadataValue);
  } else if (strcmp(metadataTag, "LINK WIDTH") == 0) {
    graphicsParameters->linkWidth = atoi(metadataValue);
  } else if (strcmp(metadataTag, "PNG ROOT") == 0) {
    strcpy(graphicsParameters->graphicsRoot, metadataValue);
  } else if (strcmp(metadataTag, "SNAPSHOTS") == 0) {
    graphicsParameters->snapshotMode = TRUE;
  } else {
    warning(MEDIUM_NOTIFICATIONS, "Ignoring unknown metadata tag in parameters file - %s\n", metadataTag);
  }
}

/* Check mandatory elements are present and validate input */
if (graphicsParameters->imageWidth <= 0) fatalError("Image width must be positive!");
if (graphicsParameters->imageHeight <= 0) fatalError("Image height must be positive!");
if (graphicsParameters->borderWidth < 0) fatalError("Border width must be nonnegative!");
if (graphicsParameters->nodeRadius <= 0) fatalError("Node radius must be positive!");
if (graphicsParameters->linkWidth <= 0) fatalError("Link width must be positive!");
```

```
  if (strlen(graphicsParameters->graphicsRoot) == 0) warning(FULL_NOTIFICATIONS, "Graphics root is empty.");

  fclose(graphicsParametersFile);
  displayMessage(DEBUG, "Finished reading graphics parameters file.\n");

}


/************************
 ** PNG writing routines **
 ************************/

pixel_type *coord2pixel(bitmap_type *bitmap, int x, int y) {
  return bitmap->pixels + bitmap->width * y + x;
}

void writePNG(bitmap_type *bitmap, char *pngFilename) {
  int x, y;
  png_byte *row, **rowPtr = NULL;
  pixel_type *pixel;
  FILE *pngFile = openFile(pngFilename, "wb");
  png_structp png = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
  png_infop info = png_create_info_struct(png);

  if (png == NULL || info == NULL) fatalError("Error allocating memory for PNG structures!");
  if (setjmp(png_jmpbuf(png))) fatalError("Error writing PNG file!");

  png_set_IHDR (png, info, bitmap->width, bitmap->height, DEPTH, PNG_COLOR_TYPE_RGB, PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);

  rowPtr = png_malloc(png, bitmap->height * sizeof(png_byte *));
  for (y = 0; y < bitmap->height; y++) {
    row = png_malloc(png, sizeof(unsigned char) * bitmap->width * PIXEL_SIZE);
    rowPtr[y] = row;
    for (x = 0; x < bitmap->width; x++) {
     pixel = coord2pixel(bitmap, x, y);
     *row++ = pixel->red;
     *row++ = pixel->green;
     *row++ = pixel->blue;
    }
  }
```

```c

  png_init_io(png, pngFile);
  png_set_rows(png, info, rowPtr);
  png_write_png(png, info, PNG_TRANSFORM_IDENTITY, NULL);

  for (y = 0; y < bitmap->height; y++) {
    png_free(png, rowPtr[y]);
  }
  png_free(png, rowPtr);
  png_destroy_write_struct (&png, &info);
  fclose(pngFile);
}
```

D.5.2 `graphics.h`

```c
#ifndef _GRAPHICS_H_
#define _GRAPHICS_H_

#include <math.h>
#include <png.h>
#include <zlib.h>
#include "network.h"
#include "utils.h"

#define PIXEL_SIZE 3
#define DEPTH 8

/* Relative coordinate points for a node i.  To use these, graphicsParameters must be a pointer to the relevant graphics struct */
#define TOP_RIGHT_X(i)     (graphicsParameters->relativeNodeX[i] + graphicsParameters->nodeRadius)
#define TOP_RIGHT_Y(i)     (graphicsParameters->relativeNodeY[i] - graphicsParameters->nodeRadius)
#define TOP_LEFT_X(i)      (graphicsParameters->relativeNodeX[i] - graphicsParameters->nodeRadius)
#define TOP_LEFT_Y(i)      (graphicsParameters->relativeNodeY[i] - graphicsParameters->nodeRadius)
#define BOTTOM_RIGHT_X(i) (graphicsParameters->relativeNodeX[i] + graphicsParameters->nodeRadius)
#define BOTTOM_RIGHT_Y(i) (graphicsParameters->relativeNodeY[i] + graphicsParameters->nodeRadius)
#define BOTTOM_LEFT_X(i)  (graphicsParameters->relativeNodeX[i] - graphicsParameters->nodeRadius)
#define BOTTOM_LEFT_Y(i)  (graphicsParameters->relativeNodeY[i] + graphicsParameters->nodeRadius)

```

```
typedef enum {
  SHALLOW,
  STEEP
} slope_type;

typedef struct graphicsParameters_s {
  int imageWidth; /* Image width and height *excludes* border */
  int imageHeight;
  int borderWidth;
  int nodeRadius;
  int linkWidth;
  int *relativeNodeX; /* Arrays containing node X and Y coordinates in relative coordinates */
  int *relativeNodeY; /* Note relative Y direction is reversed... positive downwards, while absolute Y is positive upwards */
  int *arcRelativeDX;
  int *arcRelativeDY;
  slope_type *arcSlope;
  char graphicsRoot[STRING_SIZE];
  char graphicsParametersFilename[STRING_SIZE];
  float absoluteXrange;
  float absoluteYrange;
  bool  snapshotMode;
} graphicsParameters_type;

typedef struct {
  unsigned char red;
  unsigned char green;
  unsigned char blue;
} pixel_type;

typedef struct {
  pixel_type *pixels;
  int width;
  int height;
} bitmap_type;

#include "fileio.h" /* Dependencies require this to be included after declaration of graphicsParameters_type */


/***** Image conversion routines *****/

void absolute2relative(graphicsParameters_type *graphicsParameters, float absoluteX, float absoluteY, int *relativeX, int *relativeY, int minX, int minY);
int density2red(int density, int jamDensity);
```

```c
int density2green(int density, int jamDensity);
int density2blue(int density, int jamDensity);
void generateBitmap(graphicsParameters_type *graphicsParameters, network_type *network, int t);
void generateFinalBitmap(graphicsParameters_type *graphicsParameters, parameters_type *run);

/***** Drawing routines *****/

void drawArc(bitmap_type *bitmap, graphicsParameters_type *graphicsParameters, int ij, network_type *network, int t);
void drawNode(bitmap_type *bitmap, graphicsParameters_type *graphicsParameters, int i);
void plopLabel(bitmap_type *bitmap, long label, int upperLeftX, int upperLeftY, int red, int green, int blue);
void plopDigit(bitmap_type *bitmap, int digit, int upperLeftX, int upperLeftY, int red, int green, int blue);
void resetBitmap(bitmap_type *bitmap, unsigned char red, unsigned char, unsigned char blue);
void setPixel(bitmap_type *bitmap, int x, int y, unsigned char red, unsigned char green, unsigned char blue);

/***** Handle graphics data structures *****/

void cleanUpGraphics(graphicsParameters_type *graphicsParameters);
void initializeGraphics(graphicsParameters_type *graphicsParameters, parameters_type *run);
void readGraphicsParametersFile(graphicsParameters_type *graphicsParameters, char *graphicsParametersFileName);

/***** PNG manipulation routines *****/

pixel_type *coord2pixel(bitmap_type *bitmap, int x, int y);
void writePNG(bitmap_type *bitmap, char *pngFilename);

#endif
```

Acknowledgements

The research described in this report owes much to a number of individuals assistance. The following graduate and undergraduate students assisted with developing prototypes and collecting data: Rebecca Franke, Chris Melson, Promothes Saha, Ruoyu Liu, Sadeq Safaripoor, and Ravi Venkatraman. In particular, a substantial portion of Ruoyu Lius MS thesis was supported by this project.

The Wyoming Department of Transportation provided valuable assistance through the help of project advisors Lee Roadifer and Sherm Wiseman, as well as that of Chad Matthews, who provided the TransCAD files needed to generate data for the Casper case study.