# On Evacuation Assisting Vehicular Ad Hoc Networks

by

Cong Pu, B.S.

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
Of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Dr. Sunho Lim
Chair of Committee

Dr. Noe Lopez-Benitez

Dominick Casadonte
Dean of the Graduate School

August, 2013

# ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Sunho Lim, for his continuous support during my Master's studies. He was always there to listen and to give advice. He has taken pains to go through my thesis and make necessary suggestions as and when needed.

Besides my advisor, I would like to thank my thesis defense committee member Dr. Neo Lopez-Benitez, for his kind co-ordination and his valuable time.

I would also like to thank my parents for constantly encouraging me in tough situations throughout my Master's Degree. Last but not least, I also owe sincere thanks to my girlfriend for her understanding and support.

# TABLE OF CONTENTS

# ABSTRACT

Recent natural and man-made disasters such as Hurricane Sandy (2012) and the Fukushima nuclear power plant (2011) make efficient evacuation route planning and routing more important than ever. A conventional sign-based evacuation route and its information are limited to use in a life-threatening environment. In this paper, we first investigate a least travel time based shortest path approach to minimize the evacuation time in a Vehicular Ad hoc Network (VANET). Since the travel time changes in the presence of time-varying traffic congestions, frequent and timely updates of the shorted path during the evacuation period are essential. Thus, we also investigate the evacuation assisting VANET to efficiently update the shortest path on wheel by deploying vehicle-to-infrastructure (V2I) and vehicle-to-vehicle (V2V) communications. Then we propose VANET assisting schemes, *V2RSU* and *V2RSU+V2V*. We also implement the shortest path based schemes to work in VANETs, *Upperbound* and *W/O Update*. We compare the performance of four schemes as a function of network size and number of vehicles, sources, destinations, and congestions. The simulation results indicate that the proposed schemes integrated with a VANET can reduce the evacuation time significantly.

# LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

With increasing risks from natural and man-made disasters, efficient evacuation plans play a critical role for life-saving in disaster and emergency preparedness. For example, a tsunami caused by a 9.0 earthquake damaged three nuclear reactors at the Fukushima Daiichi nuclear power plant in 2011. Over 170,000 people evacuated and dispersed from the surrounding areas [1]. Global warming impacts the creation of super storms like Hurricane Katrina (2005), Rita (2005), and Sandy (2012). Recently Hurricane Sandy affected the East Coast and 24 US states, particularly damaging New York, which is the largest city in US, and required the mandatory evacuations of over 375,000 people residing in seashore and low-lying areas [2]. Although several US states (i.e., Louisiana, Mississippi, Florida, Texas, etc.) primarily use a signed evacuation route, which simply shows a direction and radio channels for information, its effectiveness is very limited in a life-threatening environment.

For efficient evacuation route planning and routing, contraflow (or lane reversal) based approaches have been investigated to increase outbound evacuation route capacity and ultimately minimize the evacuation time. Researchers in academia and the US Department of Transportation (US-DOT) have been investigating how to operate and manage contraflow (i.e., changing or merging lanes and roads, controlling traffic lights, etc.) for maximizing the utilization of already-built infrastructures [3], [4]. On the other hand, [5], [6], [7], [8], [9], [10] propose diverse algorithms to find evacuation routes with the minimized computation in the presence of multiple sources (i.e., locations where evacuees evacuate from) and destinations (i.e., shelters where evacuees drive to). In addition, an intelligent highway infrastructure is proposed to support planned evacuations [11], where evacuation information can be disseminated based on a Vehicular Ad hoc Network (VANET) via limited vehicle-to-infrastructure (V2I) and/or vehicle-to-vehicle (V2V) communications. Note that since traffic

condition is often time-varying during an evacuation period, frequent and timely updates of evacuation information to evacuees are critical.

In this paper, we investigate an evacuation assisting VANET to minimize the evacuation time by efficiently updating the shortest path to the evacuation routes. A VANET is an instance of a mobile ad hoc network (MANET) and consists of a set of mobile vehicles equipped with computing and communication capabilities. A VANET flexibly supports either V2I or V2V communications and it is well suited for facilitating flexible accessibility and information availability. Due to rapid advances in high speed wireless Internet, being on-line without interruption on wheel has been already realized. Our contribution is two-fold:

• First, we model a transportation network as a graph form and investigate a least travel time based shortest path approach to minimize the evacuation time in a VANET environment. The proposed approach is extended by integrating with V2I and V2V communications to further reduce the evacuation time.

• Second, we develop a customized simulator and implement the proposed shortest path update schemes to work in VANETS: *Upperbound*, *W/O Update*, *V2RSU*, and *V2RSU+V2V*.

An extensive simulation study has been conducted for multidimensional analyses. We compare the performance of the schemes in terms of network size, and number of vehicles, sources, destinations, and congestions. The proposed schemes integrated with VANET, V2RSU and V2RSU+V2V, can reduce the evacuation time significantly and are a viable approach for expediting the evacuation.

The rest of this paper is organized as follows. The prior study is reviewed and analyzed in Section II. A system model and its assumptions and the proposed communication protocols and techniques are presented in Section III. Section IV is devoted to performance evaluation and analysis. Finally, we conclude the paper with future directions in Section V.

# CHAPTER II

# BACKGROUND AND RELATED WORK

In this section, we review and analyze prior algorithms and communication protocols in evacuation route planning and VANETs, respectively.

## 2.1. Evacuation Route Planning and Routing

Due to the size of transportation network and its evacuation factors, heuristic approach is often used to produce a suboptimal and efficient evacuation plan. Unlike prior heuristic approach, calculating the shortest distance from a source to the closest destination, time-varying route capacity constraints are considered to compute the evacuation route in Capacity Constrained Routing Planner (CCRP) [5], [6], [7] and its extension (CCRP++) [12]. In the CCRP, evacuees are divided into a set of groups and each group is allocated to a route and time schedule based on its earliest arrival time to the destination. Since the route capacity changes represented as a time-expanded graph, a generalized Dijkstra's shortest path search algorithm repeatedly calculates a quickest route available to each group from all sources in each iteration. The CCRP++ does not use the time-expanded graph but reduces the number of shortest path search operations by reusing the search results in previous iterations. [13] does not use the time-expanded graph either but quickest paths are searched that can reduce the evacuation time, instead of searching shortest path based on the travel time. In [9], a scalable evacuation routing algorithm is proposed based on synchronized flows instead of using the time-expanded graph. The synchronized flows are the path of evacuees from sources to destinations and the evacuees are distributed over the paths to have the same evacuation time. Virtual evacuees are also added to the synchronized flows to make the evacuation time equal.

A good number of algorithms have been proposed for contraflow-aware evacuation route planning to minimize the evacuation time by increasing outbound evacuation route capacity. Researchers in academia and US-DOT have been focusing on the operating and managing lanes and roads and controlling traffic lights [3], [4]

but they cannot flexibly incorporate with diverse evacuation factors and thus, efficient contraflow road segments cannot be produced [14]. [8] defines the contraflow problem based on graph theory and proposes two heuristic algorithms by considering road capacity constraints, multiple sources, congestion factors, and scalability: Greedy and Bottleneck relief. The greedy algorithm needs the flow history of the original network to output a contraflow reconfigured network. The bottleneck relief algorithm only needs the original network and produce a contraflow reconfigured network by using a minimum cut. A contraflow evacuation routing algorithm is proposed based on reverse shortest path and maximum throughput flow [10]. A single running of reverse shortest path can provide a set of shortest paths from all sources to the destination and thus, the path-finding time reduces. Then the proposed maximum throughput flow scheme allocates as full as available route capacity into the shortest paths for minimum evacuation time.

## 2.2. Vehicular Ad Hoc Networks Assisted Evacuation

In VANET, drivers can monitor/share real-time traffic condition, send/receive Emergency Warning Messages (EWM), and avoid an accident, such as an intersection collision or a chain collision. Diverse applications and operations built for VANETS are available in both short and extended communication ranges. To realize and disseminate these techniques, government, industry, and academic community have been putting a lot of efforts on the combining wireless technologies with VANETs [16], [17].

Based on the aforementioned computing and communication capabilities, VANET has been integrated with evacuation process. An intelligent highway infrastructure is proposed to support a planned evacuation [11] by embedding piezoelectric pressure sensor belts in the road at regular intervals. Since traffic condition is often time-varying during evacuation period, frequent and timely updates of evacuation information to evacuees are critical. Vehicles communicate with the belts by uploading and download traffic-related information. Roadside units (RSUs) [18] or access points (APs) are combined with the belts. They can query vehicles for

the travel time and disseminate this aggregated information through the belts to the vehicles as they pass over the belts. This infrastructure can also alert drivers for incoming contraflows.

In VANET, the RSUs can play a key role in evacuation process by disseminating information and assisting the communication between vehicles and the Internet. Here, a RSU can be mounted on the top of a signal light, a road lamp, a gas station, or an intersection, and it is connected with a wired network and operates as a router for vehicles to connect the Internet. Compared to the 3G/4G and satellite networks, the RSUs can provide location-dependent and real-time information with high bandwidth and low cost. [18] proposes several scheduling schemes for data access between vehicles and the RSUs. In [19], a data dissemination scheme is proposed by periodically broadcasting data to the vehicles in the road. This scheme is further improved by buffering and broadcasting data at/from the RSUs located at the intersection so that vehicles isolated from others and later arriving vehicles can still access available data.

In summary, relatively little effort has been made in developing communication protocols and related techniques for assisting evacuation route planning and routing in a VANET, which becomes critical in disaster and emergency preparedness.

# CHAPTER III

# THE PROPOSED EVACUATION-ASSISTING VEHICULAR AD HOC NETWORKS

In this section, we first present the system model and then propose a communication protocol and algorithm for evacuation assisting VANETs.

## 3.1. System Model

In VANET, vehicles are powered by their own built-in battery and execute computing and communicating operations without concerning of energy conservation. Vehicles are equipped with communication facilities such as an IEEE 802.11-based Dedicated Short Range Communication (DSRC) transceiver. Thus, vehicles can communicate with other vehicles and the Internet flexibly through V2V or V2I communications. In the V2V communication, vehicles can communicates with other vehicles directly or indirectly through a multi-hop message relay without the assistance of any fixed infrastructure, such as a RSU [18]. In the V2I communication, however, vehicles are limited to a single-hop communication with a RSU. Bypassing vehicles can pour their data into a RSU that can temporary store and forward it to the following vehicles for improving data delivery. Vehicles also equip a built-in navigation system integrated with a Global Positioning System (GPS), in which a digital map is loaded to show the roads around the current location and direction, the shortest path to the destination, and location-dependent information. Vehicles are enable to monitor real-time traffic conditions, transceiver Emergency Warning Messages (EWMs), and avoid accidents such as an intersection collision or a chain collision. Due to high speed, vehicles may experience frequent disconnections or isolations from the RSUs or other vehicles. The movement of vehicles is restricted by underlying fixed roads with speed limits and traffic lights.

In Figure 3.1, a subset of transportation network is represented as a graph form in terms of edge and vertex. In this paper, we consider a mesh network for the sake of simplicity. A RSU can be located in an intersection for sharing traffic and evacuation

information. Evacuation routes are located at the bottom with the limited number of entrance points. Upon evacuation, vehicles located in the multiple sources move to the multiple entrance points. In addition, a macroscopic network flow model is deployed to model the movement of vehicles, represented as a flow on the graph. This macroscopic model is preferred because it is effective to represent most capacity of a given transportation network such as road density, weighted mean speed, etc. [8].

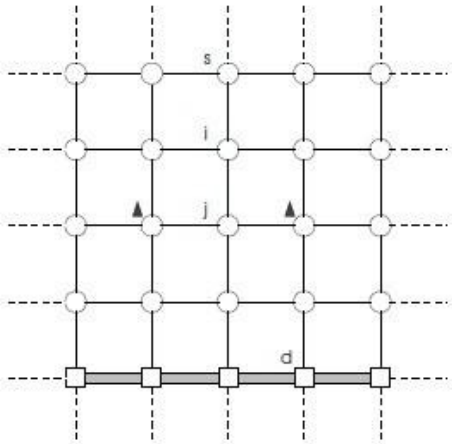


Figure 3.1. A graph representation of a subset of the transportation network.

### 3.2. Least Travel Time-based Shortest Path

During the evacuation period, a set of evacuation routes and its related information will be disseminated to vehicles through the 3G network. The information at least contains state roads or interstate highways, their available entrance points, and a set of destinations. Note that not all the entrance points may be available to vehicles because lanes or roads can be changed or merged to expedite the evacuation. Evacuation routes are calculated by recent evacuation route planning algorithms [8], [9], [10], [12], [13] to achieve the minimized evacuation time in a given disaster area. Since the algorithms consider diverse factors including road size, capacity, and condition, contraflow, traffic condition, weather, etc., evacuation routes are often pre-calculated in an off-line and ready for dissemination not to delay the evacuation. Thus, evacuation routes are seldom changed or updated in a real-time fashion.

We first investigate how quickly each vehicle can reach to one of target entrance points of the evacuation routes with the minimized evacuation time in a given transportation network. Here, the evacuation time is measured from when a first vehicle leaves a source to when a last vehicle arrives at one of entrance points of evacuation routes. Any evacuation time elapsed after reaching the entrance point of evacuation routes is not considered. In the transportation network, each road is characterized by its road capacity ($c_{road}$) and travel time ($t_{trav}$). The road capacity is measured by the number of travel vehicles per a unit period. As shown in Fig. 1, when the number of vehicles ($n_{vehicle}$) located at source ($s$) moves to destination ($d$), the evacuation time ($t_{evac}$) is calculates by,

$$t_{evac} = t_{trav} + \lfloor \frac{n_{vehicle}}{c_{road}} \rfloor - 1 \text{ [13].}$$

Under the macroscopic methodology, the road capacity can be modeled by two methods: (i) continuous entering and (ii) occupy and empty [8]. In the continuous entering method, the number of vehicles equal to the road capacity travels the road as long as the road is available. In occupy and empty method, however, the number of vehicles equal to the road capacity occupies the road for the travel time. During the travel period, the road is not available to other vehicles. In this paper, we deploy the continuous entering method because it represents the movement of vehicles more realistic.

When a vehicle receives evacuation information, it sets up a path from the current location to one of entrance points of the evacuation routes using a shortest path algorithm. Here, a shortest path is displayed in a pre-loaded area map on wheel navigation system. Because of diverse road capacities in the network, a shortest path based on the physical distance between source and destination is not considered. Instead, the least travel time will be considered by using the Dijkstra's algorithm. The pseudo code of the proposed shortest path algorithm is shown in Figure 3.2. The algorithm is terminated as soon as the next vertex is a destination.

**Notations:**

- $G_t$: The transportation graph consisting of vertices and edges for a given evacuation area.
- $v_s$, $v_d$: Source and destination vertices where vehicles starts to and evacuate to, respectively.
- trav[$i$], trav[$i$, $j$], prev_trav[$i$]: Travel times from $v_s$ to $v_i$ and from $v_i$ to $v_j$, respectively, and known travel time from $v_s$ to $v_i$.
- $L_{s,d}$: List of paths from $v_s$ to $v_d$.

◇ When a vehicle, $n$, receives an evacuation notice, $E$, through the 3G network:

```
for each v_i ∈ G_t do
    trav[i] = prev_trav[i];
end for
trav[s] = 0;
queue(v_i ∈ G_t);
while (Q_s ≠ ∅) do
    dequeue v_i with one-hop apart and smallest travel time;
    if (v_i == v_d ) then
        break;
    for (each one-hop neighbor v_j of v_i) do
        t = trav[i] + trav[i, j];
        if (t < trav[j]) then
            trav[j] = t;
    end do
    put v_j into L_{s,d};
end while
```

Figure 3.2. The pseudo code of the shortest path based on the least travel time.

### 3.3. V2I and V2V Communications Assistance

The aforementioned least travel time based shortest path is calculated once before vehicles move to a destination. Then the path is never updated during the evacuation period. One of implicit assumptions in this approach is that the travel time is fixed. However, the travel time can frequently be changed during the evacuation period because of traffic congestions. Thus, we also investigate *how a* VANET can assist in calculating least travel time based shortest path and achieve the minimized evacuation time in the presence of time-varying traffic congestions in a given transportation network.

In VANET, vehicles can communicate with a RSU to update the shortest path. A set of RSUs is installed in the intersections and plays a role as a gateway to the Internet. We consider both V2I and V2V communications. In V2I communication, as

shown in Fig. 3, when a vehicle (i.e., $n_p$) is located in the communication range of a RSU, it sends a *Request* message piggybacked with recorded travel times to the RSU. Upon receiving the message, the RSU replies an *Update* message containing updated travel times in the transportation network. Then the vehicle can recalculate a shortest path from the current location to the destination based on the updated travel times. The vehicle will replace the pre-calculated shortest path with the updated path, if the travel time can be reduced. A possible drawback of this approach is that if a vehicle does not meet any RSU during the evacuation period, it cannot update the shortest path. To increase the chance of updating the shortest path, V2V communication is also considered in which vehicles located far away from a RSU can still update their shortest path through multi-hop relays. In Figure 3.3, when a vehicle (i.e., $n_q$) is approaching to a RSU, it can receive an *Update* message from the RSU after two hop relays. Thus, the vehicle can update its shortest path and avoid traffic congestions early.

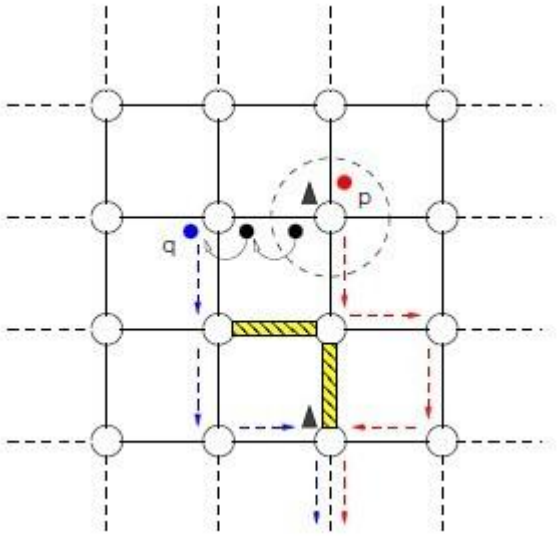


Figure 3.3. Both V2I and V2V communications assist in updating the shortest path

# CHAPTER IV

# PERFORMANCE EVALUATIONS

## 4.1. The Simulation Testbed

In this paper, we develop a customized simulator to conduct our experiments. We use a simple mesh network to model a transportation network. A set of mesh networks are deployed by changes network size, where a set of vehicles is allocated to multiple sources located in the middle of network. Two evacuation routes with multiple entrance points are located in the top and bottom of the network, respectively. Upon evacuation, vehicles move to one of entrance points of evacuation routes based on the least travel time in the presence of traffic congestion. We measure the evacuation time by changing the number of sources, destinations, vehicles, and congestions to measure the evacuation time. The simulation parameters are summarized in Table 4.1.

Table 4.1. Simulation Parameters

| Parameter | Values |
|---|---|
| Network size | 100 (10 × 10), 196 (14 × 14), 324 (18 × 18), 400 (20 × 20), 484 (22 × 22), 576 (24 × 24), 676 (26 × 26), 784 (28 × 28) |
| Number of sources | 2, 4, 6, 8, 10, 12, 14, 16 |
| Number of destinations | 4, 6, 8, 10, 12, 14, 16 |
| Number of vehicles | 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 |
| Number of congestions | 2, 4, 6, 8, 10, 12, 14, 16 |

## 4.2. Simulation Results

To compare the performance of proposed approaches, we first evaluate an ideal case where there is no traffic congestion. This case will achieve the minimum evacuation time based on the shortest path and it is used as the performance upper bound. It is denoted as *Upperbound*. Second, we consider a case where each vehicle follows its initial shortest path without update during the evacuation period in the presence of traffic congestion. It is denoted as *W/O Update*. Third, updating the initial

shortest path whenever possible via only V2I communication in a VANET is considered and it is denoted as *V2RSU*. Finally, combination of V2I and V2V communications are considered to update the shortest path directly or indirectly through multi-hop relays, and it is denoted as *V2RSU+V2V*. Here, unless otherwise specify, we use 8 sources, 4 destinations, 2,000 vehicles, and 4 or 8 congestions in the network.

1) Impact of Network Size: In Figure 4.1, we first compare the evacuation time by changing the network size and number of congestions. As the network size increases, the evacuation time increases almost linearly. With more number of traffic congestions, higher evacuation time is observed as shown in Figure 4.1(b) compared to Figure 4.1(a). The W/O Update scheme shows the highest evacuation time for entire network sizes because of lack of updates on the shortest path. In particular, more evacuation time is witnessed compared to other three schemes in Subfig. 4.1(b). Both V2RSU and V2RSU+V2V schemes show a competitive performance compared to the Upperbound scheme. Here, due to the number of congestions, smaller network sizes are not considered in Subfig. 4.1(b).

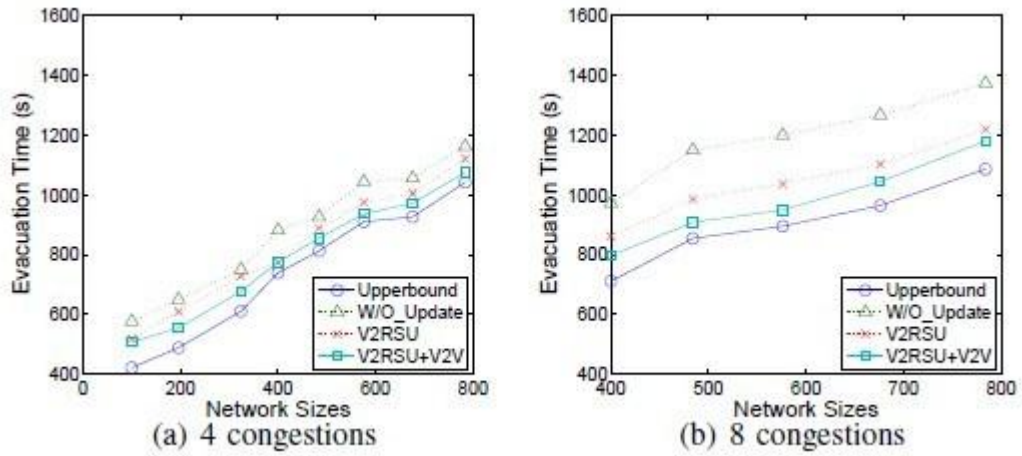


Figure 4.1. Evacuation time against network sizes.

2) Impact of Number of Vehicles: Second, we compare the evacuation time by changing the number of vehicles in Figure 4.2. A set of vehicles is allocated to multiple sources located in the middle of transportation network before initiating the

evacuation. Under the macroscopic flow model, the movement of vehicles is represented as a flow. As the number of vehicles increases, the evacuation time increases due to the limited capacity in the network. The performance gap between the W/O Update scheme and both V2RSU and V2RSU+V2V schemes increases as the number of congestion increases.

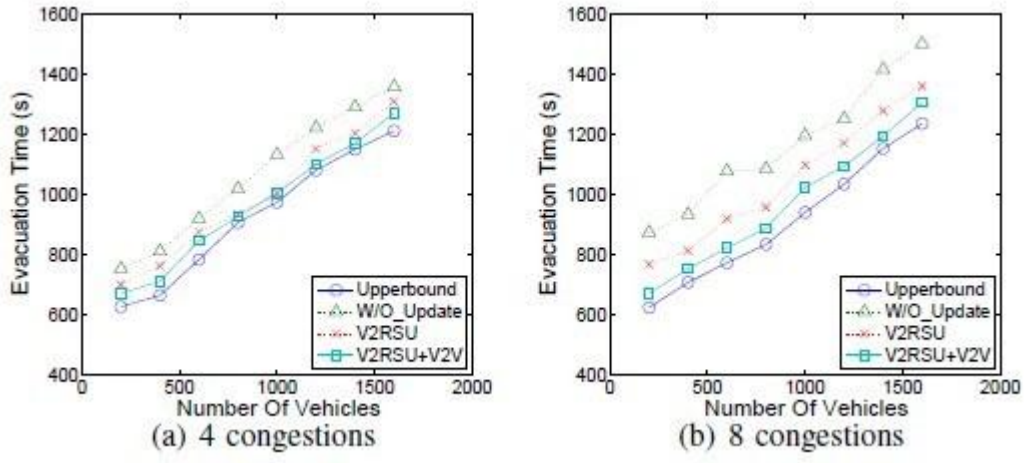


Figure 4.2. Evacuation time against the number of vehicles.

3) Impact of Number of Sources: Third, we compare the evacuation time by changing the number of sources in Figure 4.3. The equal number of vehicles are allocated to the designated number of sources. As the number of sources increases, the evacuation time decreases because vehicles are spread into the network before initiating the evacuation and the less number of vehicles is conflicted during the evacuation period. Both V2RSU and V2RSU+V2V schemes shows lower evacuation time.

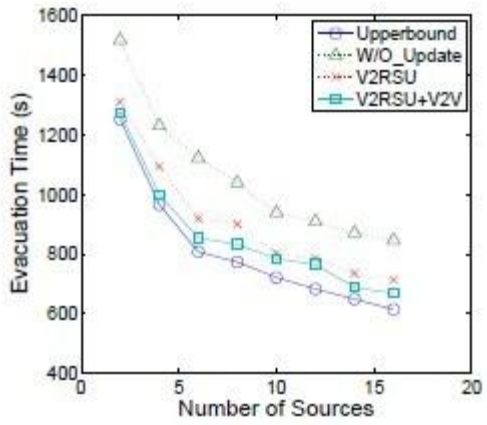


Figure 4.3. Evacuation time against the number of sources.

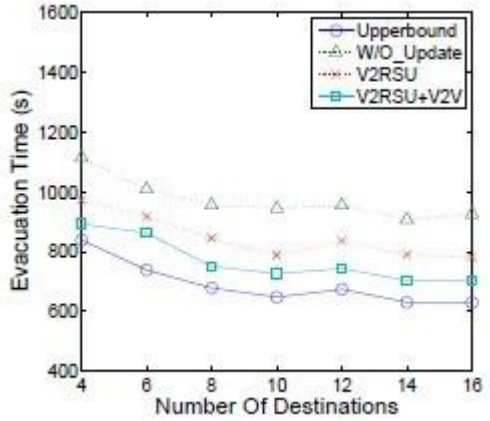


Figure 4.4. Evacuation time against the number of destinations.

4) Impact of Number of Destinations: Fourth, we compare the evacuation time by changing the number of destinations in Figure 4.4. The destination is one of entrance points of the evacuation routes. Each vehicle setups its shortest path from the current location to the destination, which is closely located. As the number of destinations increases, each vehicle has more chance to choose the shortest path with less evacuation time to the destination, and thus overall evacuation times decrease.

5) Impact of Number of Congestions: Finally, we compare the evacuation time by changing the number of congestions in Figure 4.6. Since the Upperbound scheme does not affect to the congestions, it shows a stable evacuation time. The W/O Update scheme is congestion sensitive and shows a steep increase of the evacuation time. However, both V2RSU and V2RSU+V2V schemes show low evacuation time because they can avoid traffic congestions by updating the shortest path. The V2RSU scheme shows higher evacuation time than that of the V2RSU+V2V scheme because it opportunistically updates the shortest path whenever it meets the RSU during the evacuation period. Unlike the V2RSU+V2V scheme, the V2RSU scheme cannot update the shortest path in case of missing the RSU according to the shortest path.

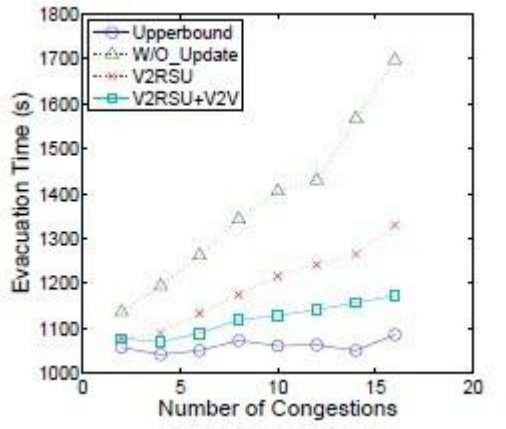


Figure 4.5. Evacuation time against the number of congestions.

# CHAPTER V

# CONCLUSION AND FUTURE RESEARCH DIRECTION

In this paper, we investigated a least travel time based shortest path approach in a transportation network and its enhancement by deploying VANET communications to minimize the evacuation time. The VANET assisting schemes, V2RSU and V2RSU+V2V, can reduce the evacuation time significantly and they show a competitive performance compared to the Upperbound scheme. We plan to extend the proposed techniques by considering a prediction mechanism for time-varying traffic congestions. Since a RSU is not always available in every intersection, vehicles may not update their shortest path frequently or in a timely manner. Thus, when vehicles meet the RSU, they predict traffic congestions based on the updated travel times and calculate the shortest path accordingly just in case of missing the RSU.

# REFERENCES

[1] The Canadian Press: IAEA says 170,000 people evacuated from area near damaged Japan nuclear plant, Google, 3-13-2011.

[2] NYC shutting down transit, evacuating 375,000, The Wall Street Journal, 10-29-2012.

[3] B. Wolshon, "One-way-out: Contraflow freeway operation for hurricane evacuation," Natural Hazards Review, vol. 2, no. 3, pp. 105–112, 2001.

[4] G. Ford, R. Henk, and P. barricklow, "Interstate highway 37 reverse flow analysis - technical memorandum," Texas Transportation Institute, Tech. Rep., 2000.

[5] Q. Lu, Y. Huang, and S. Shekhar, "Evacuation Planning: A Capacity Constrained Routing Approach," Lecture Note on Computer Science,vol. 2665, pp. 111–125, 2003.

[6] Q. Lu, B. George, and S. Shekhar, "Capacity Constrained Routing Algorithms for Evacuation Planning: A Summary of Results," Lecture Note on Computer Science, vol. 3633, pp. 291–307, 2005.

[7] S. Kim, S. Shekhar, and B. George. "Evacuation Planning: Scalable Heuristics," in Proc. Int'l Symposium on Advances in Geographic Information Systems, 2007.

[8] S. Kim, S. Shekhar, and M. Min, "Contraflow Transportation Network Reconfiguration for Evacuation Route Planning," IEEE Trans. on Knowledge and Data Engineering, vol. 20, no. 8, pp. 1115–1129, 2008.

[9] M. Min, "Synchronized Flow-Based Evacuation Route Planning," in Proc. WASA, 2012, pp. 411–422.

[10] M. Min and J. Lee, "Maximum Throughput Flow-Based Contraflow Evacuation Routing Algorithm," in Proc. Workshop on Pervasive Networks for Emergency Management (PerNEM) in conjunction with IEEE PerCom, 2013.

[11] M. C. Weigle and S. Olariu, "Intelligent Highway Infrastructure for Planned Evacuation," in Proc. IPCCC, 2007, pp. 594–599.

[12] D. Yin, "A Scalable Heuristic for Evacuation Planning in Large Road Network," in Int'l Workshop on Computational Transportation Science, 2009, pp. 19–24.

[13] M. Min and B. C. Neupane, "An Evacuation Planner Algorithm in Flat Time Graphs," in Proc. Ubiquitous Information Management and Communication, 2011.

[14] Florida's One-Way Evacuation Operation, http://www.onewayflorida.org, Florida Department of Transportation, 2012.

[15] X. Yang, J. Liu, F. Zhao, and N. Vaidya, "A Vehicle-to-Vehicle Communication Protocol for Cooperative Collision Warning," in Proc. On Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous 2004), 2004, pp. 114–123.

[16] Dedicated Short Range Communications (DSRC) Home, http://www.leearmstrong.com/DSRC/DSRCHomeset.htm.

[17] IEEE Std. 802.11p Working Group, Wireless Access for the Vehicular Environment (WAVE), http://grouper.ieee.org/groups/802/11/Reports/tgp update.htm.

[18] Y. Zhang, J. Zhao, and G. Cao, "On Scheduling Vehicle-Roadside Data Access," in ACM VANET, 2007, pp. 9–18.

[19] J. Zhao, Y. Zhang, and G. Cao, "Data Pouring and Buffering on The Road: A new Data Dissemination Paradigm for Vehicular Ad Hoc Networks," IEEE Trans. on Vehicular Technology, vol. 56, no. 6, pp. 3266–3277, 2007.

# APPENDIX

## Code Implementation

This chapter discusses some of the important functions that are used in implementing this experiment. We evaluate the performance of the proposed algorithm using our customized simulator to conduct our experiments by using C.

In the functionFile.h file, the functions describe the vehicles behavior based on the different conditions. We first evaluate an ideal case where there is no traffic congestion. This case will achieve the minimum evacuation time based on the shortest path and it is used as the performance upper bound. Second, we consider a case where each vehicle follows its initial shortest path without update during the evacuation period in the presence of traffic congestions. Third, updating the initial shortest path whenever possible via only V2I communication in a VANET is considered. Finally, combination of V2I and V2V communications are considered to update the shortest path directly or indirectly through multi-hop relays.

```
#define DEFAULT_VERTEX_NUM 400

#define INIFINITE 10000

#define RANGE 2

#define NUMBER_OF_SOURCE 4

#define NUMBER_OF_VEHICLE 1000

#define GROUP_OF_VEHILE 250

#define NUMBER_OF_DESTINATION 8

#define NUMBER_OF_CONGESTION 4

#define NUMBER_OF_ACCESSPOINT 4


typedef struct{
```

```
        int travelTime;
}GraphEdge, *PGraphEdge;
typedef struct{
        int vertexID;
        int numberOfVehicle;
}GraphVertex, *PGraphVertex;
typedef struct{
        int vertexNum;
        GraphVertex vertex[DEFAULT_VERTEX_NUM];
        GraphEdge
arcsMatrix[DEFAULT_VERTEX_NUM][DEFAULT_VERTEX_NUM];
}GraphMatrix, *PGraphMatrix;
typedef struct{
        int topSource[NUMBER_OF_SOURCE];
        int topDestination[NUMBER_OF_DESTINATION];
        int bottomSource[NUMBER_OF_SOURCE];
        int bottomDestination[NUMBER_OF_DESTINATION];
        int topID;
        int bottomID;
}SourceDestination, *PSourceDestination;



typedef struct{
        int time;
```

```
        int APcommunication;
        int Vcommunication;
        int Totalcommunication;
}TimeCommunication, *PTimeCommunication;
typedef struct{
        int pathTime;
        int *pathPointer;
        int pathSize;
}PathInfo, PPathInfo;
int randomTime(){
        int result;
        result = rand()%40 + 30;
        return result;
}
float randomCapacity(){
        time_t t;
        srand((unsigned)time(&t));
        float result;
        result = rand()%5 + 1;
        return result;
}
float randomCapacity_Ideal(int timeSeed){
        srand(timeSeed);
        float result;
```

```
	result = rand()%2 + 3;
	return result;
}

float randomCapacity_CongestionAvoidance(int timeSeed){
	srand(timeSeed);
	float result;
	result = rand()%2 + 1;
	return result;
}

int congestionTime(int timeSed){
	int i;
	int tempTime = 0;
	int finalCongestionTime = 0;
	for(i = 0; i < NUMBER_OF_CONGESTION; i++){
		srand(timeSed + i);
		if((timeSed + i + tempTime)%2==0){
			tempTime = rand()%40 + 30;
		}
		else{
			tempTime = rand()%30 + 70;
		}
		finalCongestionTime = finalCongestionTime + tempTime;
```

```
        }
        return finalCongestionTime;
}

int congestionTimeVehicle(int timeSed){
        int i;
        int tempTime = 0;
        int finalCongestionTime = 0;
        for(i = 0; i < (NUMBER_OF_CONGESTION / 2); i++){
                srand(timeSed + i);
                if((timeSed + i + tempTime)%2==0){
                        tempTime = rand()%40 + 30;
                }
                finalCongestionTime = finalCongestionTime + tempTime;
        }
        return finalCongestionTime;
}


int congestionTimeVehicleAP(int timeSed){
        int i;
        int tempTime = 0;
        int finalCongestionTime = 0;
        tempTime = rand()%20;
```

```
        finalCongestionTime = finalCongestionTime + tempTime;
        return finalCongestionTime;
}

int newTraveltime(int timeSeed){
        int result;
        srand(timeSeed);
        result = rand()%30 + 70;
        return result;
}

float randomProbability(int timeSeed){
        float result;
        srand(timeSeed);
        result = (rand()%8 + 1)/10.0;
        return result;
}

void createArcsMatrix(PGraphMatrix pGraph){
        int i;
        int j;
        int tempTravelTime;
        time_t t;
        srand((unsigned)time(&t));
```

```
        FILE *fp = fopen("travelTime.txt","w");
        if(!fp){
                printf("create and open file failed\n");
        }
        pGraph->vertexNum = DEFAULT_VERTEX_NUM;
        for(i = 0; i < pGraph->vertexNum; i++){
                pGraph->vertex[i].vertexID = i;
                pGraph->vertex[i].numberOfVehicle = 0;
        }
        for(i = 0; i < pGraph->vertexNum; i++){
                for(j = 0; j < pGraph->vertexNum; j++){
                        pGraph->arcsMatrix[i][j].travelTime = INIFINITE;
                }
        }
        for(i = 0; i < (DEFAULT_VERTEX_NUM - (int)
sqrt(DEFAULT_VERTEX_NUM)); i++){
                if(((i % (int) sqrt(DEFAULT_VERTEX_NUM)) == (int)
sqrt(DEFAULT_VERTEX_NUM) - 1)){
                        tempTravelTime = randomTime();
                        pGraph->arcsMatrix[i][i + (int)
sqrt(DEFAULT_VERTEX_NUM)].travelTime = tempTravelTime;
                        pGraph->arcsMatrix[i + (int)
sqrt(DEFAULT_VERTEX_NUM)][i].travelTime = tempTravelTime;
                }
                else{
```

```
                    tempTravelTime = randomTime();
                    pGraph->arcsMatrix[i][i + 1].travelTime = tempTravelTime;
                    pGraph->arcsMatrix[i + 1][i].travelTime = tempTravelTime;

                    tempTravelTime = randomTime();
                    pGraph->arcsMatrix[i][i + (int)
sqrt(DEFAULT_VERTEX_NUM)].travelTime = tempTravelTime;
                    pGraph->arcsMatrix[i + (int)
sqrt(DEFAULT_VERTEX_NUM)][i].travelTime = tempTravelTime;
                }
        }
        for(i = DEFAULT_VERTEX_NUM - sqrt(DEFAULT_VERTEX_NUM); i <
DEFAULT_VERTEX_NUM - 1; i++){
                tempTravelTime = randomTime();
                pGraph->arcsMatrix[i][i + 1].travelTime = tempTravelTime;
                pGraph->arcsMatrix[i + 1][i].travelTime = tempTravelTime;
        }
        for(i = 0; i < pGraph->vertexNum; i++){
                for(j = 0; j < pGraph->vertexNum; j++){
                        if(j < pGraph->vertexNum - 1){
                                fprintf(fp,"%d,",pGraph->arcsMatrix[i][j].travelTime);
                        }
                        else{
                                fprintf(fp,"%d\n",pGraph->arcsMatrix[i][j].travelTime);
```

```
                    }
                }
            }
            fclose(fp);
}

SourceDestination generateSourceDestinationPoint(GraphMatrix pGraph){
            GraphMatrix temppGraph = pGraph;
            SourceDestination tempSourceDestination;
            int i;
            int j;
            int k;
            int temp;
            int tempID;
            int topTempID[NUMBER_OF_SOURCE];
            int bottomtTempID[NUMBER_OF_SOURCE];
            int temRandom = 0;
            int repeat = 0;
            int sideLength;
            time_t t;
            srand((unsigned)time(&t));
            int frontTopID_Source;
            int backTopID_Source;
            int frontTopID_Destination;
```

```
int backTopID_Destination;
int frontBottomID_Source;
int backBottomID_Source;
int frontBottomID_Destination;
int backBottomID_Destination;
sideLength = (int)sqrt(DEFAULT_VERTEX_NUM);

frontTopID_Source = (sideLength/2 - 1)*sideLength;
backTopID_Source = frontTopID_Source + (sideLength - 1);
tempSourceDestination.topID = backTopID_Source;
frontTopID_Destination = 0;
backTopID_Destination = frontTopID_Destination + (sideLength - 1);


frontBottomID_Source = (sideLength/2)*sideLength;
backBottomID_Source = frontBottomID_Source + (sideLength - 1);
tempSourceDestination.bottomID = frontBottomID_Source;
frontBottomID_Destination = (sideLength - 1)*sideLength;
backBottomID_Destination = frontBottomID_Destination + (sideLength - 1);
//TOP:generate the source point
for(i = 0; i < NUMBER_OF_SOURCE; i++){
        repeat = 0;
        temRandom = rand()%((backTopID_Source-RANGE)-
(frontTopID_Source+RANGE)+1)+(frontTopID_Source+RANGE);
```

```
                for(j = 0; j < i; j++){
                        if(temRandom ==
tempSourceDestination.topSource[j]){
                                repeat = 1;
                                break;
                        }
                }
                if(repeat == 0){
                        tempSourceDestination.topSource[i] = temRandom;
                }
                else{
                        i = i - 1;
                }
        }
        //TOP:generate the destination point
        for(i = 0; i < NUMBER_OF_DESTINATION; i++){
                repeat = 0;
                temRandom = rand()%((backTopID_Destination-RANGE)-
(frontTopID_Destination+RANGE)+1)+(frontTopID_Destination+RANGE);
                for(j = 0; j < i; j++){
                        if(temRandom ==
tempSourceDestination.topDestination[j]){
                                repeat = 1;
                                break;
```

```
                }
            }
            if(repeat == 0){
                tempSourceDestination.topDestination[i] = temRandom;
            }
            else{
                i = i - 1;
            }
        }
        //BOTTOM:generate the source point and assign the number of vehicle to it
        for(i = 0; i < NUMBER_OF_SOURCE; i++){
            repeat = 0;
            temRandom = rand()%((backBottomID_Source-RANGE)-
(frontBottomID_Source+RANGE)+1)+(frontBottomID_Source+RANGE);
            for(j = 0; j < i; j++){
                if(temRandom ==
tempSourceDestination.bottomSource[j]){
                    repeat = 1;
                    break;
                }
            }
            if(repeat == 0){
                tempSourceDestination.bottomSource[i] = temRandom;
            }
```

```
                else{
                        i = i - 1;
                }
        }
        //BOTTOM:generate the destination point
        for(i = 0; i < NUMBER_OF_DESTINATION; i++){
                repeat = 0;
                temRandom = rand()%((backBottomID_Destination-RANGE)-
(frontBottomID_Destination+RANGE)+1)+(frontBottomID_Destination+RANGE);
                for(j = 0; j < i; j++){
                        if(temRandom ==
tempSourceDestination.bottomDestination[j]){
                                repeat = 1;
                                break;
                        }
                }
                if(repeat == 0){
                        tempSourceDestination.bottomDestination[i] =
temRandom;
                }
                else{
                        i = i -1;
                }
        }
```

```
	return tempSourceDestination;
}

TimeCommunication topDijkstraAlgorithm_Ideal(GraphMatrix pGraph, int source, int destination, int backTopID_Source){
	GraphMatrix tempToppGraph = pGraph;
	TimeCommunication tempTimeCommunication;
	int i;
	int j;
	int buffer;
	int topID = backTopID_Source + 1;
	int flag[topID];
	int shortestPath[topID];
	int tempMinWeight;
	int tempMinID;
	int temp;
	int topTempTravelTimeMatrix[topID][topID];
	Queue *record[topID];
	for(i = 0; i < topID; i++){
		for(j = 0; j < topID; j++){
			topTempTravelTimeMatrix[i][j] = tempToppGraph.arcsMatrix[i][j].travelTime;
		}
```

```
        }
        for(i = 0; i < topID; i++){
                flag[i] = 0;
                shortestPath[i] = topTempTravelTimeMatrix[source][i];
                record[i] = InitQueue();
                EnQueue(record[i], source);
        }
        flag[source] = -1;
        shortestPath[source] = 0;

        for(i = 1; i < topID; i++){
                tempMinWeight = INIFINITE;
                for(j = 0; j < topID; j++){
                        if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
                                tempMinWeight = shortestPath[j];
                                tempMinID = j;
                        }
                }
                flag[tempMinID] = -1;
                EnQueue(record[tempMinID],tempMinID);
                for(j = 0; j < topID; j++){
                        temp = tempMinWeight +
topTempTravelTimeMatrix[tempMinID][j];
                        if(j != source && temp < shortestPath[j]){
```

```
                                shortestPath[j] = temp;
                                ClearQueue(record[j]);
                                EnQueue(record[j], source);
                                buffer = DeQueue(record[tempMinID]);
                                EnQueue(record[tempMinID], buffer);
                                while(GetFront(record[tempMinID]) != source){
                                        buffer = DeQueue(record[tempMinID]);
                                        EnQueue(record[j], buffer);
                                        EnQueue(record[tempMinID], buffer);
                                }
                        }
                }
        }
        for(i=0;i<topID;i++){
                DestroyQueue(record[i]);
        }
        tempTimeCommunication.time =  shortestPath[destination];
        tempTimeCommunication.APcommunication =
tempToppGraph.vertex[source].numberOfVehicle*NUMBER_OF_ACCESSPOINT;
        return tempTimeCommunication;
}

TimeCommunication bottomDijkstraAlgorithm_Ideal(GraphMatrix pGraph, int
source, int destination, int downTempID){
```

```
GraphMatrix tempBottompGraph = pGraph;
TimeCommunication tempTimeCommunication;
int i;
int j;
int buffer;
int flag[DEFAULT_VERTEX_NUM-downTempID];
int shortestPath[DEFAULT_VERTEX_NUM-downTempID];
int tempMinWeight;
int tempMinID;
int temp;
int bottomTempTravelTimeMatrix[DEFAULT_VERTEX_NUM-
downTempID][DEFAULT_VERTEX_NUM-downTempID];
Queue *record[DEFAULT_VERTEX_NUM-downTempID];
for(i = downTempID; i < DEFAULT_VERTEX_NUM; i++){
        for(j = downTempID; j < DEFAULT_VERTEX_NUM; j++){
                bottomTempTravelTimeMatrix[i - downTempID][j -
downTempID] = tempBottompGraph.arcsMatrix[i][j].travelTime;
        }
}
for(i = 0; i < DEFAULT_VERTEX_NUM-downTempID; i++){
        flag[i] = 0;
        shortestPath[i] = bottomTempTravelTimeMatrix[source-
downTempID][i];
        record[i] = InitQueue();
```

```
            EnQueue(record[i], source);
        }
        flag[source-downTempID] = -1;
        shortestPath[source-downTempID] = 0;
        for(i = 1; i < DEFAULT_VERTEX_NUM-downTempID; i++){
            tempMinWeight = INIFINITE;
            for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
                if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
                    tempMinWeight = shortestPath[j];
                    tempMinID = j;
                }
            }
            flag[tempMinID] = -1;
            EnQueue(record[tempMinID],tempMinID+downTempID);
            for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
                temp = tempMinWeight +
bottomTempTravelTimeMatrix[tempMinID][j];
                if(j != source && temp < shortestPath[j]){
                    shortestPath[j] = temp;
                    ClearQueue(record[j]);
                    EnQueue(record[j], source);
                    buffer = DeQueue(record[tempMinID]);
                    EnQueue(record[tempMinID], buffer);
                    while(GetFront(record[tempMinID]) != source){
```

```
                                    buffer = DeQueue(record[tempMinID]);
                                    EnQueue(record[j], buffer);
                                    EnQueue(record[tempMinID], buffer);
                                }
                            }
                        }
                    }
                    for(i = 0; i < DEFAULT_VERTEX_NUM-downTempID; i++){
                        DestroyQueue(record[i]);
                    }
                    tempTimeCommunication.time =  shortestPath[destination-downTempID];
                    tempTimeCommunication.APcommunication =
tempBottompGraph.vertex[source].numberOfVehicle*NUMBER_OF_ACCESSPOIN
T;
                    return tempTimeCommunication;
}

PathInfo topDijkstraAlgorithm_PathSearch(GraphMatrix pGraph, int source, int
destination, int backTopID_Source){
                    GraphMatrix tempToppGraph = pGraph;
                    PathInfo tempPathInfo;
                    int i;
                    int j;
                    int buffer;
```

```
int topID = backTopID_Source + 1;
int flag[topID];
int shortestPath[topID];
int tempMinWeight;
int tempMinID;
int temp;
int *trace = NULL;
int topTempTravelTimeMatrix[topID][topID];
Queue *record[topID];
for(i = 0; i < topID; i++){
        for(j = 0; j < topID; j++){
                topTempTravelTimeMatrix[i][j] =
tempToppGraph.arcsMatrix[i][j].travelTime;
        }
}
for(i = 0; i < topID; i++){
        flag[i] = 0;
        shortestPath[i] = topTempTravelTimeMatrix[source][i];
        record[i] = InitQueue();
        EnQueue(record[i], source);
}
flag[source] = -1;
shortestPath[source] = 0;
for(i = 1; i < topID; i++){
```

```
                tempMinWeight = INIFINITE;
                for(j = 0; j < topID; j++){
                        if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
                                tempMinWeight = shortestPath[j];
                                tempMinID = j;
                        }
                }
                flag[tempMinID] = -1;
                EnQueue(record[tempMinID],tempMinID);
                for(j = 0; j < topID; j++){
                        temp = tempMinWeight +
topTempTravelTimeMatrix[tempMinID][j];
                        if(j != source && temp < shortestPath[j]){
                                shortestPath[j] = temp;
                                ClearQueue(record[j]);
                                EnQueue(record[j], source);
                                buffer = DeQueue(record[tempMinID]);
                                EnQueue(record[tempMinID], buffer);
                                while(GetFront(record[tempMinID]) != source){
                                        buffer = DeQueue(record[tempMinID]);
                EnQueue(record[j], buffer);
                EnQueue(record[tempMinID], buffer);
                                }
                        }
```

```
            }
        }
        tempPathInfo.pathSize = GetSize(record[destination]);

        tempPathInfo.pathTime =  shortestPath[destination];
        tempPathInfo.pathPointer = (int *)malloc(tempPathInfo.pathSize*sizeof(int));
        for(i = 0; i < tempPathInfo.pathSize; i++){
            tempPathInfo.pathPointer[i] = DeQueue(record[destination]);
        }
        for(i=0;i<topID;i++){
            if(i != destination){
                DestroyQueue(record[i]);
            }
        }
        return tempPathInfo;

}

PathInfo bottomDijkstraAlgorithm_PathSearch(GraphMatrix pGraph, int source, int
destination, int downTempID){
        GraphMatrix tempBottompGraph = pGraph;
        PathInfo tempPathInfo;
        int i;
        int j;
```

```
int buffer;
int flag[DEFAULT_VERTEX_NUM-downTempID];
int shortestPath[DEFAULT_VERTEX_NUM-downTempID];
int tempMinWeight;
int tempMinID;
int temp;
int *trace = NULL;
int bottomTempTravelTimeMatrix[DEFAULT_VERTEX_NUM-
downTempID][DEFAULT_VERTEX_NUM-downTempID];
Queue *record[DEFAULT_VERTEX_NUM-downTempID];
for(i = downTempID; i < DEFAULT_VERTEX_NUM; i++){
    for(j = downTempID; j < DEFAULT_VERTEX_NUM; j++){
        bottomTempTravelTimeMatrix[i - downTempID][j -
downTempID] = tempBottompGraph.arcsMatrix[i][j].travelTime;
    }
}
for(i = 0; i < DEFAULT_VERTEX_NUM-downTempID; i++){
    flag[i] = 0;
    shortestPath[i] = bottomTempTravelTimeMatrix[source-
downTempID][i];
    record[i] = InitQueue();
    EnQueue(record[i], source);
}
flag[source-downTempID] = -1;
```

```
shortestPath[source-downTempID] = 0;
for(i = 1; i < DEFAULT_VERTEX_NUM-downTempID; i++){
    tempMinWeight = INIFINITE;
    for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
        if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
            tempMinWeight = shortestPath[j];
            tempMinID = j;
        }
    }
    flag[tempMinID] = -1;
    EnQueue(record[tempMinID],tempMinID+downTempID);

    for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
        temp = tempMinWeight +
bottomTempTravelTimeMatrix[tempMinID][j];
        if(j != source && temp < shortestPath[j]){
            shortestPath[j] = temp;
            ClearQueue(record[j]);
            EnQueue(record[j], source);
            buffer = DeQueue(record[tempMinID]);
            EnQueue(record[tempMinID], buffer);
            while(GetFront(record[tempMinID]) != source){
                buffer = DeQueue(record[tempMinID]);
                EnQueue(record[j], buffer);
```

```
                                        EnQueue(record[tempMinID], buffer);
                                }
                        }
                }
        }
        tempPathInfo.pathSize = GetSize(record[destination-downTempID]);
        tempPathInfo.pathTime =  shortestPath[destination-downTempID];
        tempPathInfo.pathPointer = (int *)malloc(tempPathInfo.pathSize*sizeof(int));
        for(i = 0; i < tempPathInfo.pathSize; i++){
                tempPathInfo.pathPointer[i] = DeQueue(record[destination-
downTempID]);
        }
        for(i = 0; i < DEFAULT_VERTEX_NUM-downTempID; i++){
                if(i != (destination-downTempID)){
                        DestroyQueue(record[i]);
                }
        }
        return tempPathInfo;
}

TimeCommunication topDijkstraAlgorithm_CongestionAvoidance(GraphMatrix
pGraph, int source, int destination, int backTopID_Source, PathInfo pathInformation,
int *accessPointP){
        GraphMatrix tempToppGraph = pGraph;
```

```
TimeCommunication tempTimeCommunication;
PathInfo tempPathInformation = pathInformation;
int *tempAccessPointP = accessPointP;
int i;
int j;
int k;
int buffer;
int count = 0;
int topID = backTopID_Source + 1;
int flag[topID];
int shortestPath[topID];
int tempMinWeight;
int tempMinID;
int temp;
int newSource;
int numOfAP;
int queueSize;
int *trace = NULL;
int topTempTravelTimeMatrix[topID][topID];
Queue *record[topID];
newSource = tempPathInformation.pathPointer[1];
numOfAP = NUMBER_OF_SOURCE * NUMBER_OF_ACCESSPOINT * 2;
for(i = 0; i < topID; i++){
        for(j = 0; j < topID; j++){
```

```
                    topTempTravelTimeMatrix[i][j] =
tempToppGraph.arcsMatrix[i][j].travelTime;
                }
        }
        for(i = 0; i < topID; i++){
                flag[i] = 0;
                shortestPath[i] = topTempTravelTimeMatrix[newSource][i];
                record[i] = InitQueue();
                EnQueue(record[i], newSource);
        }
        flag[newSource] = -1;
        shortestPath[newSource] = 0;
        for(i = 1; i < topID; i++){
                tempMinWeight = INIFINITE;
                for(j = 0; j < topID; j++){
                        if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
                                tempMinWeight = shortestPath[j];
                                tempMinID = j;
                        }
                }
                flag[tempMinID] = -1;
                EnQueue(record[tempMinID],tempMinID);

                for(j = 0; j < topID; j++){
```

```
                        temp = tempMinWeight +
topTempTravelTimeMatrix[tempMinID][j];
                        if(j != newSource && temp < shortestPath[j]){
                                shortestPath[j] = temp;
                                ClearQueue(record[j]);
                                EnQueue(record[j], newSource);
                                buffer = DeQueue(record[tempMinID]);
                                EnQueue(record[tempMinID], buffer);
                                while(GetFront(record[tempMinID]) != newSource){
                                        buffer = DeQueue(record[tempMinID]);
                                        EnQueue(record[j], buffer);
                                        EnQueue(record[tempMinID], buffer);
                                }
                        }
                }
        }
        queueSize = GetSize(record[destination]);
        trace = (int *)malloc(queueSize*sizeof(int));
        for(i = 0; i < queueSize; i++){
                trace[i] = DeQueue(record[destination]);
        }

        for(i = 0; i < numOfAP; i++){
                for(j = 0; j < queueSize; j++){
```

```
                    if(tempAccessPointP[i] == trace[j]){
                        count++;
                    }
                }
            }
            tempTimeCommunication.time = shortestPath[destination] +
topTempTravelTimeMatrix[tempPathInformation.pathPointer[0]][tempPathInformatio
n.pathPointer[1]];
            tempTimeCommunication.APcommunication = 0;
            tempTimeCommunication.Vcommunication = 0;
            for(i = 0; i < count; i++){
                tempTimeCommunication.APcommunication =
randomProbability(tempTimeCommunication.time +
i)*tempToppGraph.vertex[source].numberOfVehicle +
tempTimeCommunication.APcommunication;
                tempTimeCommunication.Vcommunication = (1 -
randomProbability(tempTimeCommunication.time +
i))*tempToppGraph.vertex[source].numberOfVehicle +
tempTimeCommunication.Vcommunication;
            }
            tempTimeCommunication.Totalcommunication =
tempToppGraph.vertex[source].numberOfVehicle * count;


            free(trace);
            return tempTimeCommunication;
}
```

```
TimeCommunication bottomDijkstraAlgorithm_CongestionAvoidance(GraphMatrix pGraph, int source, int destination, int downTempID, PathInfo pathInformation, int *accessPointP){
    GraphMatrix tempBottompGraph = pGraph;
    TimeCommunication tempTimeCommunication;
    PathInfo tempPathInformation = pathInformation;
    int *tempAccessPointP = accessPointP;
    int i;
    int j;
    int k;
    int buffer;
    int count = 0;
    int flag[DEFAULT_VERTEX_NUM-downTempID];
    int shortestPath[DEFAULT_VERTEX_NUM-downTempID];
    int tempMinWeight;
    int tempMinID;
    int temp;
    int newSource;
    int numOfAP;
    int queueSize;
    int *trace = NULL;
    int bottomTempTravelTimeMatrix[DEFAULT_VERTEX_NUM-downTempID][DEFAULT_VERTEX_NUM-downTempID];
```

```
        Queue *record[DEFAULT_VERTEX_NUM-downTempID];
        newSource = tempPathInformation.pathPointer[1];
        numOfAP = NUMBER_OF_SOURCE * NUMBER_OF_ACCESSPOINT * 2;
        for(i = downTempID; i < DEFAULT_VERTEX_NUM; i++){
                for(j = downTempID; j < DEFAULT_VERTEX_NUM; j++){
                        bottomTempTravelTimeMatrix[i - downTempID][j -
downTempID] = tempBottompGraph.arcsMatrix[i][j].travelTime;
                }
        }
        for(i = 0; i < DEFAULT_VERTEX_NUM-downTempID; i++){
                flag[i] = 0;
                shortestPath[i] = bottomTempTravelTimeMatrix[newSource-
downTempID][i];
                record[i] = InitQueue();
                EnQueue(record[i], newSource);
        }
        flag[newSource-downTempID] = -1;
        shortestPath[newSource-downTempID] = 0;
        for(i = 1; i < DEFAULT_VERTEX_NUM-downTempID; i++){
                tempMinWeight = INIFINITE;
                for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
                        if(flag[j] != -1 && shortestPath[j] < tempMinWeight){
                                tempMinWeight = shortestPath[j];
                                tempMinID = j;
```

```
                    }
                }
                flag[tempMinID] = -1;
                EnQueue(record[tempMinID],tempMinID+downTempID);
                for(j = 0; j < DEFAULT_VERTEX_NUM-downTempID; j++){
                    temp = tempMinWeight +
bottomTempTravelTimeMatrix[tempMinID][j];
                    if(j != newSource && temp < shortestPath[j]){
                        shortestPath[j] = temp;
                        ClearQueue(record[j]);
                        EnQueue(record[j], newSource);
                        buffer = DeQueue(record[tempMinID]);
                        EnQueue(record[tempMinID], buffer);
                        while(GetFront(record[tempMinID]) != newSource){
                            buffer = DeQueue(record[tempMinID]);
                            EnQueue(record[j], buffer);
                            EnQueue(record[tempMinID], buffer);
                        }
                    }
                }
            }
            queueSize = GetSize(record[destination-downTempID]);
            trace = (int *)malloc(queueSize*sizeof(int));
            for(i = 0; i < queueSize; i++){
```

```
            trace[i] = DeQueue(record[destination-downTempID]);
        }
        for(i = 0; i < numOfAP; i++){
            for(j = 0; j < queueSize; j++){
                if(tempAccessPointP[i] == trace[j]){
                    count++;
                }
            }
        }

        tempTimeCommunication.time = shortestPath[destination-downTempID] +
bottomTempTravelTimeMatrix[tempPathInformation.pathPointer[0] -
downTempID][tempPathInformation.pathPointer[1] - downTempID];
        tempTimeCommunication.APcommunication = 0;
        tempTimeCommunication.Vcommunication = 0;
        for(i = 0; i < count; i++){
            tempTimeCommunication.APcommunication =
randomProbability(tempTimeCommunication.time +
i)*tempBottompGraph.vertex[source].numberOfVehicle +
tempTimeCommunication.APcommunication;
            tempTimeCommunication.Vcommunication = (1 -
randomProbability(tempTimeCommunication.time +
i))*tempBottompGraph.vertex[source].numberOfVehicle +
tempTimeCommunication.Vcommunication;
        }
```

```
        tempTimeCommunication.Totalcommunication =
tempBottompGraph.vertex[source].numberOfVehicle * count;
        free(trace);
        return tempTimeCommunication;
}
```