CPE 400
Project #6

Software Defined Networks for Traffic Management

Program Documentation

December 6, 2017

Bryan Kline
Ryan Lieu
Robert Watkins

Table of Contents:

Program Description:

The program is written in C++, terminal-based, and is intended to simulate a smart city which is organized like a network with intersections acting as nodes or routers, roads between intersections acting as edge or links, and autonomous vehicles which are routed through the system acting as packets. The user first adds any number of intersections to the city, specifying how many cars are at that intersection when the system begins the simulation. The user then adds roads connecting the intersections and for each road a weight must also be added which specifies the amount of time needed for a car to traverse that road. Intersections may have up to four roads connecting it with other intersections, and the time associated with each road must be greater than zero.

After adding intersections, cars at each intersection, and roads connecting intersections, the user must then assign each car in the system a destination intersection so that each car has a source intersection and a destination intersection. The user may then add a delay to the system; the user will select one intersection and then assign a delay to this intersection which adds that amount of time to each road connected to it. The purpose of this is to simulate an event, such as road construction or lane closure, which causes the weights of all roads coming from that intersection to increase. The system then will output tables displaying the shortest paths of all cars in the system to their destinations without the delay, the paths of all cars with the delay without re-routing of cars to avoid these new costs, and the new shortest paths of all cars taking these new costs into consideration and their new routes to possibly avoid the delays. This delay acts to simulate link failure in a network and how this error might be handled in a smart city. In addition to the tables, the shortest time, longest time, and average time for each table is displayed so that each can be compared.

After the delay and its effects on the network are simulated, an additional simulation is carried out which models the effects of traffic congestion on the network. The city, prior to having the delay added, has each of its cars' trajectories stepped through in time and checked against those of all other cars to determine whether or not they occupy the same intersection at the same time. If too many cars occupy the same intersection at the same time then they affect one another's times to reach their

subsequent destination intersections. This is intended to simulate the effect of traffic congestion on the system. A novel approach to correcting for traffic congestion is simulated whereby cars which are at the same intersection at the same time but whose *next* destination intersection is also the same are considered to be one car, as autonomous cars controlled globally may maneuver moment-to-moment in unison and if they are leaving that intersection together then they don't impose wait times on one another. Tables for the system prior to simulation of traffic congestion, the system with traffic congestion costs without correction, and the system with congestion and congestion correction are displayed, along with each table's shortest, longest, and average times.

The user interacts with the program in the Linux terminal by way of a menu system which displays prompts for intersections, cars, roads, car trajectories, and a delay all to be added to the system. The user may then choose to display the tables for the system with delay and with traffic congestion. The user may also choose to display a summary of the the system map which shows the total number of intersections, total number of cars, then one by one for each intersection all roads coming from that intersection, where each leads, the total cars that start there, their trajectories, their shortest paths through the system, and a table showing the shortest path from that intersection to every other one in the system. The user may also choose whether or not the output of the program should be written to a file, which is always called simply "output.txt", in addition to printing the output to the terminal. A convenient way for the user to interact with the system is to redirect a text file into the program which feeds in the menu commands to build a map and run the simulations. Some sample maps are provided in the software package in addition to a readme file for compiling and running the program. The program is meant to be built and run only once per map, the program should be restarted if a new map is desired.

Program Structure Overview:

main.cpp

The main driver for the program creates two class objects of type System, which is the city map. One of these is the city without the delay, and the traffic congestion simulation, and the other is the city with the delay added. The user interface is largely handled in the main driver which displays a menu system, handled by a switch case block in a loop, each menu option corresponding to an action the user can take to build the system and run the simulations. The main driver also uses a few free functions to assist in building the system map and running the simulations.

System.h
System.cpp

The System class contains the city map and handles the way in which the map is built and simulated, with a number of methods to add Intersections, Roads, Car trajectories, run the simulations, and output the system summary and all system tables. The System class maintains a linked list of Intersection class objects, which in turn are composed of Road, Car, and DijkstraNode structs, as well as three two dimensional arrays of Path structs that hold the Paths that the Cars take in the system at various times.

Intersection.h

Intersection.cpp

The Intersection class contains all the information relevant to a particular Intersection, including the number of Roads there, the number of Cars that begin there, an array of Roads which connect it with other Intersections, an array of Cars which are the those that begin at that Intersection, and a two dimensional array of DijkstraNode structs which is the table built using Dijkstra's algorithm from that Intersection to compute the shortest paths, based on Road weights, to every other Intersection in the system.

Program Units:

main.cpp

Free Functions:

name: initializeDestinations
description: takes in a string array and frees allocated memory in the array if it exists and sets the char pointers to NULL; the array if used to load Car destinations from the user which is then sent to the System class to set all the Car trajectories
parameters: takes in the string array to be initialized and a bool corresponding to whether or not the array currently has memory allocated to its elements or not
return: void
prototype: void initializeDestinations(char* destinationList[], bool initialize);

name: setDestinations
description: loads a string array with all the destinations for a particular intersection which is then used passed to a System method to set the Car destinations
parameters: takes in a string array which takes the destinations, a string containing the current Intersection, and an int corresponding to the number of cars there
return: void
prototype: void setDestinations(char* destinationList[], char* intersectionName, int totalCars);

name: integerToAlphaNumeric
description: takes in an int and converts it to a string
parameters: the int to be converted, the string where it is be stored, the size of the string
return: void
prototype: void integerToAlphaNumeric(int value, char* buffer, int lastIndex);

Intersection.h

Structs:

name: Road

attributes: an int corresponding to the amount of time needed to traverse that Road, an Intersection pointer array of size two holding the references to the starting and ending Intersections for that Road
purpose: connects Intersection class objects together, data member of the Intersection class

name: Car
attributes: a string holding the name of the car, all names are the name of the Intersection the Car starts at and its number, a string array of size two which holds the names of the Car's starting and ending Intersections, a pointer to a Path struct which points to a list of Paths which constitute that Car's shortest path through the map based only on Road weights
purpose: acts as an individual car or packet in the system which is routed to its destination, data member of the Intersection class

name: Path
attributes: an int which is the cost in terms of time to reach that position in the Car's path, a bool corresponding to whether, in the traffic congestion simulation, that portion of the Car's path can be combined with another Car, a string which is the name of the Intersection the Car is at, a pointer to a Path which is the next Path struct in the list
purpose: an individual node which is used to build the path of a Car through the system, each node being a move the Car makes, holding the Intersection name at that point as well as the cumulative cost to get there, also used by the System class to make tables of all cars to display their paths through the system, data member of Intersection and System classes

name: DijkstraNode
attributes: an int which is the cost of a Car to get to a particular Intersection from its initial Intersection, a string which is the name of the last Intersection through which a Car would need to pass to get to the current Intersection, a bool corresponding to whether or not the shortest path up to a particular has been calculated
purpose: a node in an OSPF table which is built in order to calculate Dijkstra's algorithm from a particular Intersection to find the shortest paths to all other Intersections in the system, data member of the Intersection class

Classes:

name: Intersection
attributes: a string holding the Intersection's name, and int for the number of Cars starting at that Intersection, an int for the number of Roads at that Intersection, an array of Roads which connect the Intersection with other Intersections, a Car pointer which is the head of a list of Cars, an Intersection pointer which points to the next Intersection in the System's list of Intersections, a double pointer to a DijkstraNode which is the start of a two dimensional array which is the OSPF table from that Intersection to every other Intersection in the system; friend of System class
purpose: acts as an individual node in a list of Intersections maintained by the System class to model the city map

Intersection.cpp

Class Methods:

name: Intersection
description: parameterized constructor, allocates memory for the Car array then allocates memory for strings for each car, and sets the Road array to defaults, the start of the Road points at this intersection, the other end points to NULL
parameters: takes in a string corresponding to the name of the Intersection and an int for the number of Car objects there
return: none
prototype: Intersection(char* intersectionName, int totalCars);

name: ~Intersection
description: destructor, frees all dynamically allocated memory
parameters: none
return: none
prototype: ~Intersection();

Free Functions:

name: integerToAlphaNumericCoversion
description: takes in an int and converts it to a string
parameters: the int to be converted, the string where it is be stored, the size of the string
return: void
prototype: void integerToAlphaNumericCoversion(int value, char* buffer, int lastIndex)

System.h

Structs:

none

Classes:

name: System
attributes: ints for the total number of Intersection, the total number of Cars, the longest Car path in the system, the latest time a Car travels for, the traffic threshold for the system which sets the number of Cars at an Intersection at which traffic begins to congest, an int to control the order which methods are called, disallowing certain methods to be called if certain others haven't been called yet, three double pointers to Path structs which make up three tables which correspond to the shortest paths of all the Cars in the system based just on Road weights, shortest paths based on a delay introduced into the system, and shortest paths based on traffic congestion, each table also has three ints corresponding to the lowest, highest, and average times for Cars in that table

purpose: the city map which maintains all Intersections, Cars, and tables of Paths which allows for simulations to be run on the city for both link failure, delay, and traffic congestion; friend of Intersection class

System.cpp

Class Methods:

name: System
description: default constructor, sets data members to default values
parameters: none
return: none
prototype: System();

name: ~System
description: destructor, frees all dynamically allocated memory
parameters: none
return: none
prototype: ~System();

name: isEmpty
description: checks whether or not the Intersection list exists or not
parameters: none
return: returns a bool corresponding to whether or not the list exists
prototype: bool isEmpty();

name: addIntersection
description: creates an Intersection object, names it the string passed in and sets the number of cars to the int passed in, and connects the newly created Intersection to the Intersection list
parameters: takes in a string corresponding to the name of the new Intersection and an int for the number of Car objects at that Intersection
return: returns an int which is an error code that reports what error was encountered
prototype: int addIntersection(char* newIntersection, int cars);

name: addRoad
description: takes in to and from Intersection names and connects the Road array at the from Intersection to the to Intersection, and sets the time it takes to traverse that Road
parameters: takes in two strings, the to and the from Intersection names, and the time it takes to traverse that Road
return: returns an int which is an error code which reports what error was encountered
prototype: int addRoad(char* toIntersection, char* fromIntersection, int time);

name: setCarTrajectories
description: sets the destinations of the Cars in the Car array at an Intersection to all the destinations entered as the string array parameter

parameters: takes in a string which is the name of the current Intersection as well as a string array which holds the names of the other Intersections to which the Cars at that Intersection are traveling
return: returns an int which is an error code which reports what error was encountered
prototype: int setCarTrajectories(char* fromIntersection, char* destinationList[]);

name: checkCarTrajectories
description: iterates through the string array passed in as a parameter to test whether or not all destinations in the array are valid from the current Intersection
parameters: takes in a string which is the name of the current Intersection as well as a string array which holds the names of the other Intersections to which the Cars at that Intersection are traveling
return: returns an int which is an error code which reports what error was encountered
prototype: int checkCarTrajectories(char* fromIntersection, char* destinationList[]);

name: addDelay
description: takes in the name of an Intersection, moves to that Intersection, and iterates through the Roads at that Intersection and adds a delay to each one
parameters: takes in a string corresponding to the Intersection at which the delay occurs and an int which is the amount of time the delay occurs for
return: returns an error code as an int which reports whether or not the addition of the delay was successful
prototype: int addDelay(char* fromIntersection, int delay);

name: getTotalCars
description: returns the number of Cars at a given Intersection
parameters: takes in a string corresponding to the name of the Intersection from which to get the number of Cars
return: returns and int which is either an error code which reports what error was or the number of Cars at that Intersection encountered
prototype: int getTotalCars(char* intersectionName);

name: getSystemCars
description: returns the total number of Cars in the system
parameters: none
return: returns an int corresponding to the total number of Car objects in the entire system
prototype: int getSystemCars();

name: getIntersection
description: takes in the name of an Intersection and returns a reference to it if it exists, otherwise NULL is returned
parameters: takes in a string which is the name of the Intersection for which a reference is requested
return: returns a pointer to an Intersection which is the Intersection requested if it exists, NULL if it doesn't exist
prototype: Intersection* getIntersection(char* intersectionName);

name: OSPF
description: all Intersections in the system are iterated through, the shortest paths to every other intersection are calculated using Dijkstra's algorithm and stored in a table after initializing it, then the shortest paths for all Cars in the system are calculated using each Intersection's OSPF table, and then the system time tables are populated with all Cars' shortest paths
parameters: none
return: void
prototype: void OSPF();

name: initializeOSPFTable
description: dynamically allocates memory for the OSPF table at a given Intersection, sets each DijkstraNode in the table to default values, then the Intersection list is moved through and the name of each Intersection is added to the first row of the table which acts as column labels
parameters: takes in a string corresponding to the name of the Intersection whose OSPF table is to be initialized
return: void
prototype: void initializeOSPFTable(char* intersectionName);

name: runDijkstra
description: runs Dijkstra's algorithm at a given Intersection, calculating the shortest paths from that Intersection to all other Intersections and populating the OSPF table at the Intersection with all the costs to every other Intersection
parameters: takes in a string which corresponds to the Intersection at which Dijkstra's algorithm shall be run
return: void
prototype: void runDijkstra(char* intersectionName);

name: calculatePaths
description: iterates through the Intersection list and creates Path lists for the Cars at that intersection and then consults the OSPF table at that Intersection to create the shortest path for that Car and the Path list is filled in with the intermediate Intersections and costs for each move that Car makes through the system
parameters: none
return: void
prototype: void calculatePaths();

name: getIndex
description: checks the OSPF table at a given Intersection to see if a requested Intersection is in the table and if so its index in the table is returned
parameters: takes in a string which is the current Intersection and another string which is the Intersection whose index in the table should be returned
return: returns an int corresponding to the index in the OSPF table at a given Intersection another Intersection resides at, -1 is returned if either the current Intersection doesn't exists or the requested table Intersection doesn't exist
prototype: int getIndex(char* intersectionName, char* label);

name: getTime
description: takes in the names of two Intersections and returns the time needed to traverse the Road between them if it exists, zero is returned if it doesn't
parameters: takes in two strings corresponding to the start and end Intersections of a Road
return: returns an int corresponding to the time needed to traverse a given Road, if the Road doesn't exist then zero is returned
prototype: int getTime(char* fromIntersection, char* toIntersection);

name: initializeSystemTable
description: three copies of a table of Car trajectories are created of size Cars in the system, rows, by the longest path, columns, by iterating through all Cars in the system adding their shortest paths to each table
parameters: none
return: void
prototype: void initializeSystemTable();

name: applyDelay
description: one of the system tables with the default shortest Car paths loaded into it is iterated through and changed in such a way so that delays are added to the times in the table
parameters: takes in a string corresponding to the Intersection at which the delay occurs and an int which is the amount of time the delay occurs for
return: void
prototype: void applyDelay(char* fromIntersection, int delay);

name: simulateTraffic
description: the primary system time table containing all Cars in the system and their shortest paths, and the amount of traffic at each Intersection, at each time, for each Car, is compared with a traffic threshold and if it’s above the threshold a cost ripples forward in the table increasing each Car's path times; the same process also takes place on another table which has had Cars with similar trajectories combined in order to correct for traffic congestion
parameters: none
return: void
prototype: void simulateTraffic();

name: combineCars
description: the system table which will hold the Car travel times after congestion correction has taken place is iterated through and if a Car is at an Intersection at a given time and shares a portion of their path with another Car at that same Intersection at that exact time, then they are combined into one in order lessen the effects of traffic on congestion on the system
parameters: none
return: void
prototype: void combineCars();

name: timeSummary

description: the three system tables which hold the simple shortest paths, the paths with delay times added, and the paths with traffic congestion, are iterated through and the lowest, highest, and average travel times, are calculated and stored in class data members
parameters: none
return: void
prototype: void timeSummary();

name: displaySystem
description: the Intersection list is moved through and the information for each Intersection is printed to the screen, and written to a file
parameters: takes in a bool corresponding to whether or not the Intersection data should be written to a file and an ofstream object which is the stream connected to the file
return: void
prototype: void displaySystem(bool toFile, ofstream &outFile);

name: outputSystemTables
description: the system tables are iterated through and their contents, as well as their average, lowest, and highest travel times, are printed to the screen and written to a file
parameters: takes in a bool corresponding to whether or not the tables should be written to a file, an ofstream object which is the stream connected to the file, and an int corresponding to which secondary table, either the table holding delay times or the table holding traffic congestion times, should be displayed in addition to the primary table
return: void
prototype: void outputSystemTables(bool toFile, ofstream &outFile, int outputCopy);

name: errorControl
description: takes in an error code and prints to the screen the type of error encountered
parameters: takes in an int corresponding to the error code to be reported
return: void
prototype: void errorControl(int errorCode);

Free Functions:

none

makefile

Contains compilation directives, assuming g++ as the C++ compiler.

Program Use Cases:

The software package has, in addition to documentation and compilation and execution instructions, two files with user input pre-loaded which can be redirected into the program in the terminal. These files each construct a particular city map, complete with intersections, roads, road weights, car counts,

and car trajectories. Visual representations of the city maps constructed by the input files, named "input0" as shown in Figure 1, and "input1" as shown in Figure 2, are presented below.

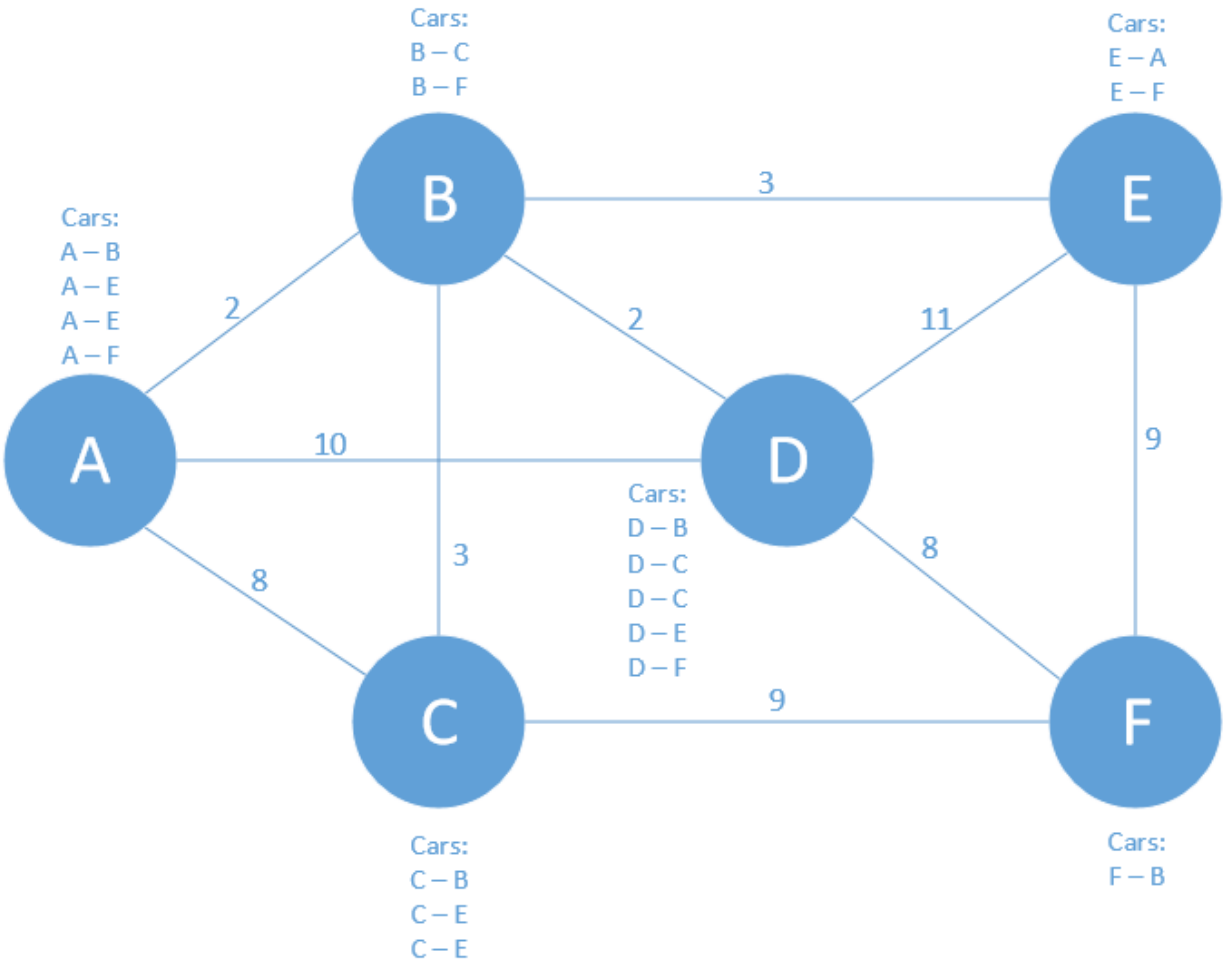

Figure 1: The city map constructed by redirecting the file "input0" into the program.

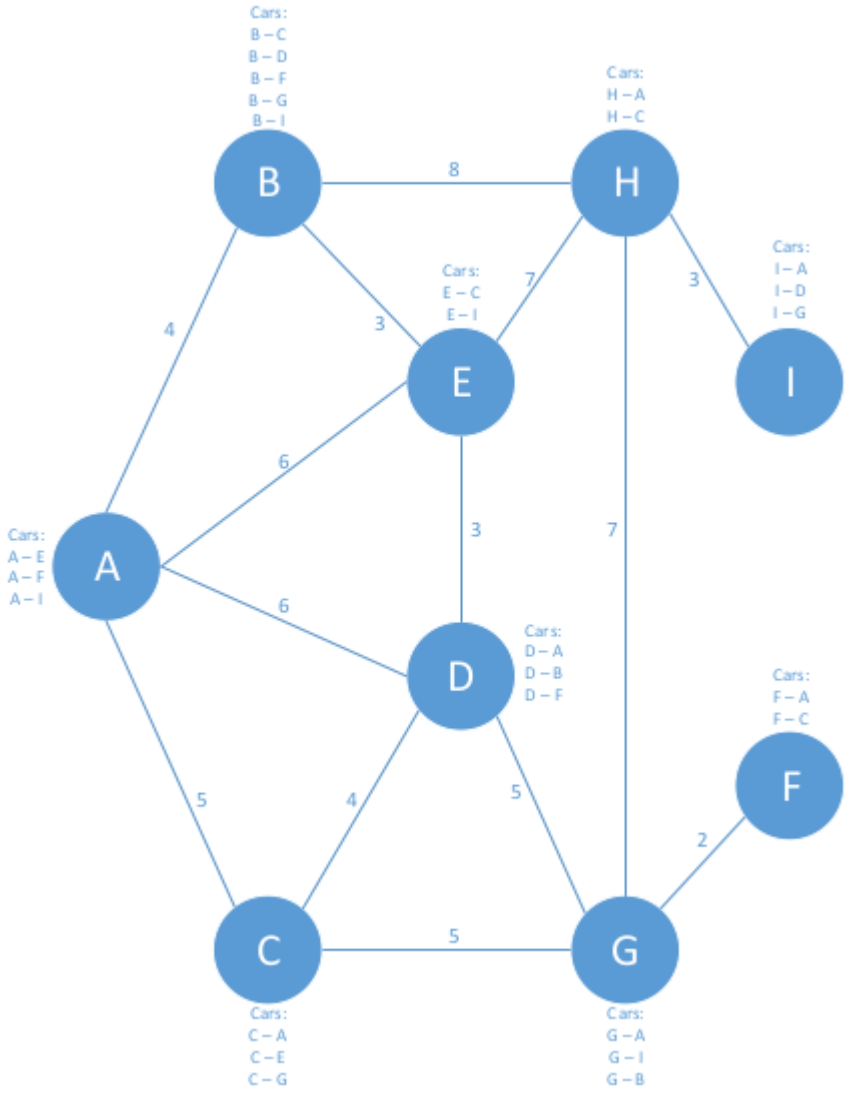

Figure 2: The city map constructed by redirecting the file "input1" into the program.