Gone Before the Wind

Summary

Hurricanes can be devastating natural disasters, having serious adverse consequences not only from their physical force but also as a result of the traffic and housing congestions resulting from evacuation of the coastal areas. This paper analyzes the problem of routing inland huge numbers of vehicles in a relatively short time period, specifically from South Carolina. We modeled the system abstractly as a graph with edges representing roads and nodes representing cities and road intersections. Since the number of nodes grows extremely quickly when adding roads (because of additional road intersections), we limited our model to a few major highways and a few key cities. We recognized this problem as a problem of maximizing flow and analyzed our graph using the Ford-Fulkerson Algorithm. Even our overly simplified model resulted in satisfactory evacuation times, so we concluded that if given a full model of the state’s cities, populations, and road systems, much better evacuation times would be achieved. The scale of this model is readily extensible to huge graphs and should provide efficient evacuation routes. The model has several drawbacks that are discussed, but they seem to be relatively minor.

Restatement of the Problem

The problem is to find the optimal method of evacuating a huge number of people inland when a hurricane threatens. The several components of the problem involve deciding when to turn the flow of traffic on two-way highways into one-way traffic, deciding on evacuation timing for the coastal counties, deciding whether or not to add more shelters to certain metropolitan areas, deciding when to restrict people’s freedom to take as many vehicles as they would like when they evacuate, and analyzing the impact of Georgians and Floridians driving north on South Carolina highways. Setting up appropriate nodes and edges can reduce these questions to a relatively simple graph-theoretical problem of optimal flow. Using the Ford-Fulkerson algorithm, every aspect of this problem can be analyzed in a straightforward manner.

Assumptions

1. The life of every resident of South Carolina is worth saving.

2. Everyone from the thirteen coastal counties (Dillon, Marion, Horry, Florence, Williamsburg, Georgetown, Berkeley, Dorchester, Charleston, Colleton, Hampton, Beaufort, and Jasper) must reach safety within twenty-four hours of the issuance of a hurricane warning.

3. People from counties farther inland are already safe.

4. The actual number of people in the coastal region is about the same as the official population of that area (i.e. the number of non-residents visiting the area is roughly the same as the number of residents of the area who are not currently there).
5. The proportion of county populations to households has remained approximately constant.

6. The average household in the South Carolina coastal counties has two vehicles.

7. People will attempt to save as many vehicles as possible when they are ordered to evacuate.

8. Each metropolitan area to which we evacuate people has the capacity to hold as many people as already live there (i.e. we can effectively double a city’s population).

9. The rate of traffic flow is proportional to the number of lanes open for travel on any given highway.

10. Traffic can be treated reasonably as a fluid.

11. Drivers on average follow the “two-second rule” for following other vehicles.

12. We have greatly reduced the number of roads and cities used in our model. Actual evacuation routes could be calculated using this method by adding all the relevant roads and cities of departure and destination.

Model Design

To get a more accurate representation of the population distribution of South Carolina’s coastal areas, we first obtained demographic information for each of the thirteen relevant counties (see attached table). We calculated the number of households per county in 1999 using the 1999 populations and the 1998 populations and numbers of households. We then found the numbers of vehicles per county we expected to need to evacuate based on the numbers of households in each county. To conduct our analysis using a graph, we lumped counties together and represented them by one city. Dillon, Marion, Florence, and Williamsburg were lumped together and represented by the city of Florence. Horry and Georgetown were lumped together and represented by Myrtle Beach. Charleston, Dorchester, and Berkeley were lumped together and represented by the city of Charleston. Finally, Colleton, Hampton, Jasper, and Beaufort were lumped together and represented by Hilton Head. We then calculated (using Microsoft Streets & Trips 2001) the fastest routes under normal driving conditions between each of these “lumps” and the possible destinations (Atlanta, Augusta, Columbia, Spartanburg, and Charlotte).

We use only three fundamental characteristics to describe large-scale traffic motion: speed, density, and flow rate. The definitions of the first two terms are fairly obvious. Speed is the magnitude of the velocity of the individual cars, which is equivalent to the velocity of the traffic “fluid” as a whole. For this model, we ignore the effects of differences in the velocities of individual cars and assume that all of the cars are moving at the mean velocity. This seems to be a reasonable assumption on large enough scales and on roads where traffic is not confined to one lane. In these circumstances, we feel that micro-scale processes will automatically minimize the effects of these small differences in speeds.
Density is defined as the number of cars within a given distance (we use units of cars/lane-mile) and is given by the equation:

\[ K = \frac{N}{D} = 1/d \]  

(1)

where \( K \) is density, \( N \) is the number of cars per lane on a section of road, \( D \) is the length of the section of road, and \( d \) is the average effective length of a car (mean length of a car plus the mean distance between one car and another).

Flow rate is defined as the number of cars that pass a given point in a given time (we use units of cars/lane-hour). Flow rate is given by the equation:

\[ q = U \cdot K \]  

(2)

where \( U \) is the speed in units of miles/hour, and \( K \) is the density in units of cars/lane-mile. If we use equation (1), we get:

\[ q = \left(\frac{U}{d}\right) \cdot 5280 \text{ feet/mile}, \]  

(3)

where \( d \) is given in units of lane-feet.

It is now clear that to maximize \( q \), we need to maximize the ratio of \( U \) to \( d \). However, \( U \) has a maximum equal to the posted speed limit (70 miles/hour in the case of most divided highways in South Carolina). We also know that the minimum safe value for \( d \) is given by the two-second rule. The two-second rule states that the minimum safe distance between a car and the car in front of it is the distance that it covers in 2 seconds. So if we assume 12 feet is the mean length of a vehicle and \( U \) is 70 miles/hour, we get the equation:

\[ q_{\text{max}} = \frac{(70 \text{ miles/hour} \cdot 5280 \text{ feet/mile})}{(70 \text{ miles/hour} \cdot 5280 \text{ feet/mile} \cdot 2 \text{ sec} \cdot 1/3600 \text{ hour/sec}) + 12 \text{ feet}} \]  

(4)

\[ q_{\text{max}} = 1700 \text{ cars/hour} \]  

(5)

If we instead use 60 miles/hour for \( U \), \( q_{\text{max}} \) falls to 1685 cars/lane-hour. So we see that near highway speeds \( q_{\text{max}} \) is only weakly dependent on the speed limit (a decrease of more than 14% in \( U \) corresponds to a decrease of less than 1% in \( q_{\text{max}} \) in this case). Therefore, for the sake of simplicity, we used 1700 cars/hour multiplied by the number of lanes in a highway to calculate the maximum safe capacity and hence the optimum load of the highways that we used in our evacuation routes.

This relationship between speed and flow rate is of vital importance in situations where large numbers of vehicles need to be moved in as little time as possible. In these situations it is vital that traffic keeps moving. As the graph shows, flow rate is still above 90% of \( q_{\text{max}} \) at 25 miles/hr., but \( q \) falls off very quickly at speeds less than that.

So we see that the two most important factors in maintaining maximal capacity are (A) make as many lanes as possible useable for traffic (each highway lane adds about 1700 cars/hour to the optimum capacity) and (B) make sure that traffic stays above 25 miles/hour.

We have modeled the flow of cars as a weighted directed graph. In our case, the vertices of the graph represent towns or intersections of roads (i.e. places where one can change roads), and the edges represent the roads themselves (directed to allow for one-way roads; two-way roads can be represented by pairs of edges). We have seen that each road has
a certain capacity (in cars/hour) and that our graph will have a set of sources (points of origination) and sinks (points of destination) of cars. Furthermore, we may prefer certain edges over others by the traversal time (the weight of a corresponding edge). Our goal is to maximize the flow rate from the sources to the sinks while minimizing the average traversal time. In general, the max-flow problem (for one source and one sink) is solved with the Ford-Fulkerson Algorithm (1956). The Ford-Fulkerson Algorithm essentially states that the maximum flow can be obtained by running through the following loop:

For each path from source to sink, add as much flow as possible going through that path without exceeding the capacity of any edge.

After running through every path, we will have the maximum flow. There are a few problems with this algorithm that are easily resolvable. First, it might seem that being limited to a single sink and a single source is not realistic. To eliminate this problem, we created a virtual source and a virtual sink. We gave the virtual source a directed edge to each of the other sources with capacities equal to the populations of the sources. Similarly, we created edges from each of the sinks to the virtual sink. This way, we have not only avoided the problem, but we have also included finite population considerations into the algorithm. Another major problem is that the algorithm does not take into account the traversal times of the various paths (any path is as good as any other). To force the algorithm to “prefer” certain paths, we gave the paths to the algorithm in order of increasing traversal time. Thus, the shorter paths would be chosen first. After these few modifications, we have a simple, viable algorithm to give escape routes from each desired city.

**Verification and Testing**

Again in favor of simplicity, we prepared source code for a program that would use our algorithm to find the “best” evacuation routes. The source code (in C++) as well as sample input and output for a simplified coastal model is attached.

**Discussion**

Let us begin by addressing certain practical points specific to South Carolina’s geography. Clearly it is almost always beneficial to turn the two coastal-bound lanes of I-26 into two lanes of Columbia-bound traffic, since the evacuation flow from Charleston (a massive population center) would double via that route. It has been suggested that staggering the evacuation may improve traffic flow. However, we feel that the change would not be significant because the routes from different regions do not interfere with one another. It would certainly help to reverse evacuation flow on smaller highways besides I-26. In particular, I-20’s traffic flow would be a major benefit to the flow of the system. However, smaller roads would probably not change the flow, since most drivers would not choose these roads in the first place. However, if the algorithm required a higher flow this is where it would come from. Another possible benefit would be to increase Columbia’s capacity by adding more temporary shelters. Again, our sample input and output suggest that although the average traversal time would certainly decrease (since Columbia is closer to the coast than, say, Atlanta), unless the capacities of the other major sinks were to decrease dramatically, there would be no need for more support in this node. One clear benefit of the algorithm is that it deals
with certain management questions as how many cars per family should be allowed by simply stating the maximum flow out of a given city. Another benefit is the ability to simulate a larger-scale evacuation, such as evacuation of Georgia and Florida that occurred in Hurricane Floyd by creating a source that input large flows into I-95 and large sinks beyond I-95 that would represent the northern safe-havens.

There seems to be a strong intuitive connection between traffic patterns and the motions of fluids. We speak of “traffic flow” and “traffic ripples” or “waves” naturally and without consciously thinking about it. Consequently, our choice to model traffic as a fluid seemed natural.

The simple fluid model is exactly that—simple. It gives us an estimate of the optimal capacities of roads by assuming an equilibrium state where drivers maintain a constant speed in order to preserve a constant density. It completely ignores all of the micro-scale processes that cause the equilibrium. By introducing these processes into the model, much more precise information about what is happening on the roads could be obtained.

In reality, speed is a function of density. For example, if we have a car with a speed slightly above the mean, the inter-car distance will decrease beneath the optimum of a two-second following distance. When the driver notices that he is getting too close to the car in front of him, he will reduce speed to avoid a collision. If, on the other hand, we have a driver who is driving slowly, the car behind him will start getting too close. This may be an incentive to drive faster, but it is more likely that on a highway, the car behind will simply pass the slower one (this is especially effective if the highway has dedicated passing lanes so that the vehicle needn’t temporarily disrupt traffic in the other lane). By these processes, the average speeds and densities remain near equilibrium when there are no obstacles that affect the road capacity.

When we have obstacles that decrease the maximum flow of a lane either permanently or temporarily, we have new problems that our model may not be able to describe adequately. For example, a construction site can cause the flow rate of a lane to go to zero, cause no change in the flow rate, or decrease the flow to any intermediate value. If we have a flagger allowing cars to pass through an area at a specific rate, the maximum flow for that entire lane drops to the new flow rate. But this cannot happen instantaneously. Instead, the first car will be forced to slow down, causing the car behind to slow down and so on until the entire lane has regained equilibrium at the new, lower flow rate. This phenomenon is called a shockwave. Because the optimum capacity is achieved at equilibrium and shockwaves are effects of a non-equilibrium transition, shockwaves will reduce the efficiency of the roads in addition to the drop in capacity that the obstacle has already caused. This is clearer in the case where the obstacle is temporary. For example, a crash might close a lane for a period of an hour. The flow rate decreases quickly after the accident has occurred, but because the backward-moving shockwave travels slower than a “recovery” shockwave, the flow rates take slightly longer to return to normal. This results in a net decrease in mean flow rate in addition to the decrease due to the obstacle itself.

Since our model ignores shockwaves, the capacities that we have estimated for our highways are overly optimistic. If we have few incidents that cause shockwaves on our routes, then the difference will be negligible. If, on the other hand, we have many of these incidents, the correction might be significant. Unfortunately, we do not have a reliable method of predicting the severity of the effect that incidents like these will have.
However, we maintain that our model is perfectly usable in finding the relative differences between the capacities of two roads. For two identical roads, the expected corrections for micro-scale processes should be identical. One could postulate that there are a certain number of incidents per lane-mile (possibly modeled by a Poisson distribution) that lead to these corrections. Therefore, the corrections should be proportional to the length of the route. So our method of giving precedence to shorter routes and otherwise treating roads exactly alike should, in theory, be perfectly acceptable. While our time estimates might be overly optimistic, the routes we determine to be optimal should be completely acceptable.

We favor the Ford-Fulkerson algorithm for its simplicity. However, a potential problem with this model is its speed. A higher-scale model would almost certainly necessitate a fast algorithm rather than a simple one. For this purpose, we would suggest the Goldberg-Tarjan Algorithm (1985). However, its complexity makes it inappropriate and unreasonable for our model. Furthermore, both the Ford-Fulkerson and the Goldberg-Tarjan Algorithms can be modified to handle discrete time-dependent flow models. This would certainly be more accurate and produce more efficient routes.

References


Appendices

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "graph.h"

/* File: speedo.cpp
 * Description: Finds the optimum route for evacuation of people
 * threatened by a hurricane. Optimum is based on the
 * average time it takes to get to a safe location
 * and on the number of people saved.
 */

/* The two function prototypes countallpaths and findallpaths
 * are for functions defined at the end of this file. The functions
 * respectively count and store all the paths from source to sink
 * in our graph.
 */

void
countallpaths(Node *current, Node *sink);
void
findallpaths(Node *current, Node *sink, Path *accumulator, Edge *last);

// These are global variables used by the functions.
// They must be global since the functions are recursive.
int numpaths, tempath;
Path **paths;

int
main(int argc, char** argv, char** envp) {
    // These variables are temporary storage variables.
    Edge **edges;
    Path temp_patha, *temp_path;
    FILE *input, *output;
    char ch, space[100], spacea[100], spaceb[100];
    int lines, i, j, numnodes, numedges, sum, temp_flow;
    float temp_time;

    // User friendly user interface %^-)
    if(argc < 3) {
        printf("Syntax error. Proper syntax:\n\t%s <nodes_file>
"
<edges_file> [edges_output]n", argv[0]);

  return -1;
}

// Input the Nodes file
input = fopen(argv[1], "r");
for(lines = 0; (ch = getc(input)) != EOF; lines += (ch=='\n')?1:0);
rewind(input); nodes = new Node*[++lines];

for(numnodes=0; numnodes<lines && !feof(input); numnodes++) {
  fscanf(input, "%s", space);
  nodes[numnodes] = new Node(space, numnodes);
  printf("%sn", space);
  if(strlen(space) == 0)
    source = nodes[numnodes];
  else if(!strcasecmp(space, "T"))
    sink = nodes[numnodes];
}
fclose(input);

// Input the Edges File
input = fopen(argv[2], "r");
for(lines = 0; (ch = getc(input)) != EOF; lines += (ch=='\n')?1:0);
rewind(input); edges = new Edge*[++lines]; numedges = lines;

for(i=0; i<lines && !feof(input); i++) {
  fscanf(input, "%s %s %s %f %d", space, spacea, spaceb, &temp_time, &temp_flow);
  tempa = NULL; tempb = NULL;
  for(j=0; j<numnodes; j++) {
    if(!strcasecmp(space, nodes[j]->name))
      tempa = nodes[j];
    if(!strcasecmp(spacea, nodes[j]->name))
      tempb = nodes[j];
  }
  if((tempa == NULL) && (tempb == NULL))
    printf("Line %d ignored. Node \"%s\" and \"%s\" undefined.n",
           i+1, space, spacea);
  else if(tempa == NULL)
    printf("Line %d ignored. Node \"%s\" undefined.n", i+1, space);
  else if(tempb == NULL)
    printf("Line %d ignored. Node \"%s\" undefined.n", i+1, spacea);
  if((tempa == NULL) || (tempb == NULL)) continue;

  edges[i] = new Edge(spaceb, temp_flow, temp_time, i);
edges[i]->src = tempa; edges[i]->dst = tempb;
tempa->AddEdge(edges[i]);
}

// sort out the quickest paths based on the first weight (traversal time)
// find the number of paths
numpaths = 0;
countallpaths(source, sink);

// allocate space for the paths
paths = new Path*[numpaths];

// find and store the paths, while counting path lengths
tempath = 0;
findallpaths(source, sink, new Path(), NULL);

// sort them (using simple insertion sort) by path length
// This sort algorithm is inefficient O(N^2) and if this
// program were being used for a large scale application,
// it would be replaced with Quicksort
for (j=1; j<numpaths; j++) {
temp_time = paths[j]->totaltime;
temp_path = paths[j];
for(i = j-1; i >= 0; i--) {
    if(paths[i]->totaltime < temp_time) break;
    paths[i+1] = paths[i];
}
paths[i+1] = temp_path;
}

// use Ford-Fulkerson to figure out the maximum flow
sum = 0;
for(i=0; i<numpaths; i++) {
    sum += paths[i]->AddMaxCap();
}

// output the results
printf("Flow = %d\n", sum);
if(argc > 3) {
    output = fopen(argv[3], "w");
    for(i=0; i<numpaths; i++) {
        fprintf(output, "\nFlow = %d. Time = %f. Path: ", paths[i]->flow,
            paths[i]->totaltime);
        paths[i]->print(output);
    }
    fflush(output); fclose(output);
void
countallpaths(Node *current, Node *sink) {
    int i;
    if (current == sink) {
        ++numpaths; return;
    }
    for (i = 0; i < current->numouts; i++)
        countallpaths(current->next[i]->dst, sink);
}
void
findallpaths(Node *current, Node *sink, Path *accumulator, Edge *last) {
    int i; Path *a = new Path(accumulator);
    if (last != NULL) a->AddInt(last);
    if (current == sink) {
        paths[tempath++] = a;
        return;
    }
    for (i = 0; i < current->numouts; i++)
        findallpaths(current->next[i]->dst, sink, a, current->next[i]);
}
/* graph.h : Graph related data structures (C++)
* Node  - a node on the graph; in the case of the South Carolina
*        hurricane weather escape routes, cities or intersections.
* Edge  - an edge on the graph; in our case, a road
* Path  - a sequence of consecutive edges from the source to the sink.
*/

// Class Declarations
class Node;
class Edge;
class Path;

class Node {
public:
    char *name; // Name of the Node
    Edge **next; // Array of Edges leaving the node
    int numouts, numalloc; // The number of used cells, and the total
                          // number of cells
    int NodeID; // The index of the Node in the main array

    Node(char* name_in, int id); // Constructor
    void AddEdge(Edge *dst); //
}; // End Node Class

class Edge {
public:
    Node *src; // Source Node of the Edge
    Node *dst; // Destination Node
    char *name; // Name of the Edge (like "I-95" :)
    int capacity; float time; // The Capacity of the Edge, and
                             // the length of the edge (traversal time)
    int EdgeID; // Index of the Edge in the main array
    int used_cap; // Capacity Used Out of the Edge

    Edge(char* name_in, int cap, float time_in, int id); // Constructor
}; // End Edge Class

class Path {
public:
    Edge **points; // Edges in the path sequence
    int numused, numalloc, flow; // Number of edges in the path,
                                 // the number we have space for,
                                 // and the strength of the path, resp.
    float totaltime; // The length of the path (traversal time)
Path(); // Default Constructor
Path(Path *a); // Copy Constructor
void AddInt(Edge* x); // Append an edge to the path
int AddMaxCap(); // Increase the flow of the path as much as possible
void print(FILE* output); // Print the path to the file "output"
};

// Class Member Function Implementations

Node::Node(char* name_in, int id) {
    name = new char[strlen(name_in)+1];
    strcpy(name, name_in);
    next = new Edge*[3];
    numalloc = 3; numouts = 0;
    NodeID = id;
};
void
Node::AddEdge(Edge *dst) {
    if(numouts >= numalloc) {
        Edge** tempos = new Edge*[2*numalloc];
        for(int i=0; i<numalloc; i++)
            tempos[i] = next[i];
        delete [] next;
        next = tempos;
        numalloc = 2*numalloc;
    }
    next[numouts++] = dst;
};

Edge::Edge(char* name_in, int cap, float time_in, int id) {
    name = new char[strlen(name_in)+1];
    strcpy(name, name_in);
    capacity = cap; time = time_in;
    EdgeID = id;
    used_cap = 0;
};

Path::Path() {
    points = new Edge*[10];
    numused = 0; numalloc = 10;
    totaltime = 0; flow = 0;
};
Path::Path(Path *a) {
    points = new Edge*[a->numalloc];
    numalloc = a->numalloc;
numused = a->numused;
totaltime = a->totaltime;
for(int i=0; i<numused; i++) {
    points[i] = a->points[i];
} 
flow = a->flow;
};
void
Path::AddInt(Edge* x) {
    if(numused >= numalloc) {
        Edge** tempos = new Edge*[2*numalloc];
        for(int i=0; i<numalloc; i++)
            tempos[i] = points[i];
        delete [] points;
        points = tempos;
        numalloc = 2*numalloc;
    }
    points[numused++] = x;
    if(x != NULL)
        totaltime += x->time;
};
int
Path::AddMaxCap() {
    int addable, i, y;
    addable=points[0]->capacity;
    for(i=0; i<numused; i++) {
        y = points[i]->capacity-points[i]->used_cap;
        if(addable > y) addable = y;
    }
    for(i=0; i<numused; i++)
        points[i]->used_cap += addable;

    return (flow = addable);
};
void
Path::print(FILE* output) {
    for(int i=0; i<numused; i++) {
        fprintf(output, "\n\t %s -> %s via %s", points[i]->src->name,
                     points[i]->dst->name, points[i]->name);
    }
}
File: nodes.txt

S
Charleston
Florence
Hilton_Head
Myrtle_Beach
T
Augusta
Atlanta
Columbia
Spartanburg
Charlotte
File: edges.txt

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Charleston 0</td>
<td>391093</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Florence   0</td>
<td>163329</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Myrtle_Beach 0</td>
<td>189245</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Hilton_Head 0</td>
<td>138140</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Augusta     T T 0</td>
<td>14112</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atlanta     T T 0</td>
<td>118119</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Columbia    T T 0</td>
<td>15809</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Spartanburg T T 0</td>
<td>28466</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Charlotte   T T 0</td>
<td>43400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Charleston</td>
<td>Augusta US78</td>
<td>1.986</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Charleston</td>
<td>Columbia I26</td>
<td>1.614</td>
<td>13600</td>
</tr>
<tr>
<td></td>
<td>Florence</td>
<td>Charleston US52</td>
<td>1.571</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Florence</td>
<td>Columbia I20</td>
<td>1.129</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Florence</td>
<td>Charlotte US74</td>
<td>1.614</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Myrtle_Beach</td>
<td>Columbia US378</td>
<td>1.971</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Myrtle_Beach</td>
<td>Charlotte US501</td>
<td>2.614</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Myrtle_Beach</td>
<td>Florence US501</td>
<td>0.971</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Hilton_Head</td>
<td>Augusta US278</td>
<td>2.143</td>
<td>3400</td>
</tr>
<tr>
<td></td>
<td>Hilton_Head</td>
<td>Columbia I26</td>
<td>2.357</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Augusta</td>
<td>Atlanta I20</td>
<td>2.129</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Columbia</td>
<td>Augusta I20</td>
<td>1.000</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Columbia</td>
<td>Spartanburg I26</td>
<td>1.343</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Columbia</td>
<td>Charlotte I77</td>
<td>1.329</td>
<td>6800</td>
</tr>
<tr>
<td></td>
<td>Spartanburg</td>
<td>Atlanta I85</td>
<td>2.457</td>
<td>6800</td>
</tr>
</tbody>
</table>
Flow = 6800. Time = 1.129000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> T via T
Flow = 3400. Time = 1.614000. Path:
S -> Florence via S
Florence -> Charlotte via US74
Charlotte -> T via T
Flow = 9009. Time = 1.614000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> T via T
Flow = 0. Time = 1.971000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> T via T
Flow = 3400. Time = 1.986000. Path:
S -> Charleston via S
Charleston -> Augusta via US78
Augusta -> T via T
Flow = 0. Time = 2.100000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> T via T
Flow = 0. Time = 2.129000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> T via T
Flow = 3400. Time = 2.143000. Path:
S -> Hilton_Head via S
Hilton_Head -> Augusta via US278
Augusta -> T via T
Flow = 0. Time = 2.357000. Path:
S -> Hilton_Head via S
Hilton_Head -> Columbia via I26
Columbia -> T via T
Flow = 0. Time = 2.458000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> Charlotte via I77
Charlotte -> T via T
Flow = 0. Time = 2.472000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> T via T
Flow = 0. Time = 2.585000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Charlotte via US74
Charlotte -> T via T
Flow = 4591. Time = 2.614000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> Augusta via I20
Augusta -> T via T
Flow = 3400. Time = 2.614000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Charlotte via US501
Charlotte -> T via T
Flow = 0. Time = 2.700000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> T via T
Flow = 0. Time = 2.943000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> Charlotte via I77
Charlotte -> T via T
Flow = 0. Time = 2.957000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> Spartanburg via I26
Spartanburg -> T via T
Flow = 2209. Time = 2.971000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> Augusta via I20
Augusta -> T via T
Flow = 0. Time = 3.100000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> T via T
Flow = 0. Time = 3.185000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Charlotte via US74
Charlotte -> T via T

Flow = 1191. Time = 3.300000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> Charlotte via I77
Charlotte -> T via T

Flow = 0. Time = 3.314000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> Spartanburg via I26
Spartanburg -> T via T

Flow = 0. Time = 3.357000. Path:
S -> Hilton_Head via S
Hilton_Head -> Columbia via I26
Columbia -> Augusta via I20
Augusta -> T via T

Flow = 0. Time = 3.429000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> Charlotte via I77
Charlotte -> T via T

Flow = 0. Time = 3.443000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> T via T

Flow = 5609. Time = 3.686000. Path:
S -> Hilton_Head via S
Hilton_Head -> Columbia via I26
Columbia -> Charlotte via I77
Charlotte -> T via T

Flow = 0. Time = 3.700000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> T via T
Flow = 1191. Time = 3.700000. Path:
S -> Hilton Head via S
Hilton Head -> Columbia via I26
Columbia -> Spartanburg via I26
Spartanburg -> T via T
Flow = 0. Time = 4.029000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> Charleston via I77
Charleston -> T via T
Flow = 0. Time = 4.043000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> T via T
Flow = 0. Time = 4.115000. Path:
S -> Charleston via S
Charleston -> Augusta via US78
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 4.258000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 4.272000. Path:
S -> Hilton Head via S
Hilton Head -> Augusta via US278
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 4.743000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 4.929000. Path:
S -> Florence via S
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T
Flow = 0. Time = 5.100000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 5.229000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 5.414000. Path:
S -> Charleston via S
Charleston -> Columbia via I26
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T
Flow = 0. Time = 5.486000. Path:
S -> Hilton_Head via S
Hilton_Head -> Columbia via I26
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 5.771000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Columbia via US378
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T
Flow = 0. Time = 5.829000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> Augusta via I20
Augusta -> Atlanta via I20
Atlanta -> T via T
Flow = 0. Time = 5.900000. Path:
S -> Myrtle_Beach via S
Myrtle_Beach -> Florence via US501
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T
Flow = 0. Time = 6.157001. Path:
S -> Hilton Head via S
Hilton Head -> Columbia via I26
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T
Flow = 0. Time = 6.500000. Path:
S -> Charleston via S
Charleston -> Florence via US52
Florence -> Columbia via I20
Columbia -> Spartanburg via I26
Spartanburg -> Atlanta via I85
Atlanta -> T via T