A Route Assignment Approach to Vehicle Emergency Evacuation

Dalila Lima, Miguel Sandim, Paula Fortuna, Rosaldo J. F. Rossetti
Faculdade de Engenharia da Universidade do Porto
email: {ei12043, ei12061, ei12025, rossetti}@fe.up.pt

KEYWORDS

Emergency Evacuation, Traffic Network Modelling, Graph Theory, Traffic Flow Theory

ABSTRACT

The aim of this study is to develop a strategy to address the problem of assigning routes to vehicles in an emergency situation. In order to minimize the network travel time we use the Ford-Fulkerson Maximum Flow and its generalization with the Minimum Cost Flow. Both algorithms worked as an initial starting point to our solution. However due to our travel time definition using the BPR function, the use of the Minimum Cost Algorithm itself does not guarantee the user equilibrium of the network. This forced us to improve this solution and develop a complementary algorithm, which consists of a greedy strategy. This proved to improve the final result and also the route assignment.

INTRODUCTION

The problem addressed in this paper is how to assign an evacuation route to a large amount of vehicles if an emergency situation occurs. In order to do that we have to take into account that in case of an emergency it is probable that all citizens will be moving at the same time, and therefore a large amount of vehicles may try to circulate through the same road, so it is critical to avoid traffic congestion.

Concerning the development of the algorithms, we define a directed graph $D(V, A)$ with a set of nodes V and a set of arcs A.

The nodes can be of three types: source (o), intermediate (i), and destiny (d).

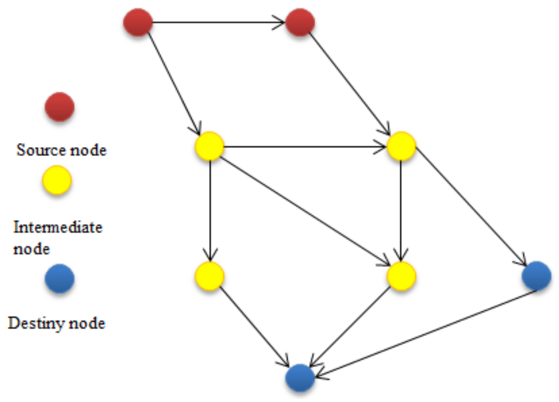

Figure 1: Network example

The main definitions are:

- i represents the index of a network node;
- V_i represents the network node with index i;
- $i \rightarrow j$ represents a directed arc from node with index i to j;
- A route R_k is a set of arcs belonging to the graph that lead a source node to a destiny node;
- N_i represents the number of vehicles that is associated to each source node with index i;
- f_{ij} represents the flow in an arc from $i \rightarrow j$ of the network. The concept of flow can be defined as the number of vehicles passing through a reference point per unit of time;
- fc_{ij} represents the maximum flow allowed in an arc $i \rightarrow j$ (arc capacity);
- f_{R_k} represents the flow of vehicles in the route R_k of the network;
- TT_{ij} is the average travel time for a vehicle along the road represented by the arc $i \rightarrow j$, according to the current flow f_{ij};
- FFT_{ij} represents the average free flow travel time along the road represented by the arc $i \rightarrow j$. It depends on the maximum speed limit allowed, road length and physical characteristics of the road.
- TT_{R_k} is the travel time along the route R_k;

- $max(TT_n)$ is the max network travel time of a network n.

In order to solve the problem presented earlier, the proposed solution is to minimize the worse travel time among the network and guarantee that all the vehicles arrive at the destiny nodes in a reasonable period of time. In this approach we seek a solution that follows the principles of user equilibrium [1]. This means that the algorithm guarantees that the network is completly explored, and therefore assigning another route to specific vehicles will not achieve better results.

Definition of the travel time based on the BPR function

In our model, the travel time needed to travel a distance between two nodes i and j is a parameter of the arc $i \rightarrow j$ between them. However, this is not a constant measure, but a function that depends on f_{ij} which will be calculated considering the volume of vehicles to be assined to that arc, using the BPR function [1].

$$TT_{ij}(f_{ij}) = FFT_{ij}\left(1 + 0.15\left(\frac{f_{ij}}{fc_{ij}}\right)^4\right) \tag{1}$$

The free flow travel time of an arc $i \rightarrow j$ (FFT_{ij}) corresponds to the time it takes a vehicle to travel along the link and reach its end, considering that there will be no need to interact with other vehicles also circulating there, stop at traffic lights or other external factors.
In the context of this travel time function, the practical flow capacity of a link $i \rightarrow j$ (fc_{ij}) corresponds to a measure of flow (vehicles per unit of time) at which the TT_{ij} is about 15% higher than the FFT_{ij}, whereas the flow attempting to use the link (f_{ij}) consists on the input variable.

Model formulation and constraints

Our optimization model can be mathematically defined as follows:

Minimize:

$$max(TT_n) \tag{2}$$

where:

$$max(TT_n) = max\left\{TT_{R_k}\right\}, \forall k \tag{3}$$

$$TT_{R_k} = \sum\nolimits_{i,j:(i\rightarrow j)\in R_k} TT_{ij} \tag{4}$$

$$f_{R_k} = min\left\{f_{ij} : (i \rightarrow j) \in R_k\right\}, \forall k \tag{5}$$

$$\begin{gathered} FFT_{ij} \geq 0 \ \wedge \ TT_{ij} \geq 0 \ \wedge \ TT_{R_k} \geq 0 \\ \wedge \ max(TT_n) \geq 0 \ , \ \forall i, \forall j, \forall k, \forall n \end{gathered} \tag{6}$$

$$N_i \geq 0 \ , \ \forall i \ : \ i = o \tag{7}$$

$$\begin{gathered} f_{ij} \geq 0 \wedge fc_{ij} \geq 0 \ \wedge \ f_{R_k} \geq 0 \ , \\ \forall i, \forall j, \forall k \end{gathered} \tag{8}$$

$$\begin{gathered} \sum\nolimits_{i:(i\rightarrow j)\in A} f_{ij} \ - \ \sum\nolimits_{i:(j\rightarrow i)\in A} f_{ji} = 0 \ , \\ \forall j : j \neq o \ \wedge \ j \neq d \end{gathered} \tag{9}$$

$$\begin{gathered} \sum\nolimits_{i:(j\rightarrow i)\in A} f_{ji} \ - \ \sum\nolimits_{i:(i\rightarrow j)\in A} f_{ij} = N_j \ , \\ \forall j : j = o \end{gathered} \tag{10}$$

The objective function represented in (2) aims to minimize the estimated total time of the evacuation. Formulation (3) defines the variable to minimize as being the maximum TT_{R_k} in the targeted network. Formulation (4) defines the travel time of a route k as being the sum of the travel times of all its arcs. Formulation (5) defines the flow of a route R_k as being the minimum flow of all arcs that belong to R_k. Constraint (6) guarantees that all travel times are non negative. Constraint (7) guarantees that all sources have a non negative number of vehicles. Constraint (8) guarantees all flow values are non negative. Constraint (9) guarantees that the sum of all incoming flow in all nodes (not being origins or destinations) is equal to the sum of all outgoing flow in the same node (flow conservation theorem). Constraint (10) guarantees that the difference between the incoming flow and the outgoing flow in all the source nodes is the number of vehicles that will be evacuated starting in that node (in order to respect the flow conservation theorem).

It is important to notice that $f_{ij} \leq fc_{ij}, \forall i, \forall j$ is not a constraint, although algorithms described in sections and respect this condition, the algorithm analyzed in section violates it if it minimizes the goal function.

Problem assumptions

- The number of vehicles in each origin will not exceed the maximum possible flow of the network.
- It is assured that all vehicles entering the network will be able to reach the exit points, even if that may delay the evacuation process.
- It is assumed that all the vehicles adopt the assigned route.

CONCEPTUAL SOLUTION

In order to minimize the network travel time, we started searching among maximum flow and minimum cost flow algorithms for a solution.
It is assumed that this problem deals with sparse graphs because in an urban transportation network, the most common scenario is to have a node with four adjacent edges (4-way intersection). Therefore, the number of edges will have the same order of magnitude as the number of vertices. This is important because a dense graph, with an higher number of edges, would imply using different algorithms.

BPR function usage

The study of the BPR function was also relevant in the research about the algorithm to use, as it is explained here.

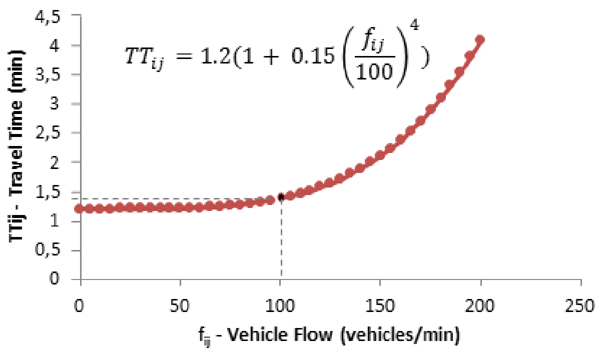

Figure 2: BPR function of an arc $i \rightarrow j$ with FFT_{ij} of 1.2 minutes and fc_{ij} of 100 vehicles/minute

Firstly, the FFT_{ij} in the BPR function is the travel time at zero flow. At this point, a travelling car would not be delayed because of an interaction with any other car. The only source of delay at this point is the time needed to travel from the origin to the destiny of that link $i \rightarrow j$. However, as more vehicles start using the same network path, the value of TT_{ij} needed to travel through the link will increase.

In the begining, when the number of vehicles is below the capacity of the link, TT_{ij} increases slowly as we add more vehicles to the network. As we can observe in the figure 2, TT_{ij} is almost constant while the f_{ij} is below the edge capacity (fc_{ij}).
However if f_{ij} exceeds fc_{ij}, TT_{ij} grows at a much higher rate. When we overcome this point (marked as bold in the chart) the travel time is constantly increasing, as the function has no assimptotic behavior. This particular aspect of the BPR function was taken into account in the algorithm choice.

In order to find the minimum time needed to take the vehicles from a point in the network to another, we use an adapted Minimum Cost Flow algorithm [4]. This algorithm works with the limit capacity from the edges and tries to put the greatest amount of flow in the links, costing the least travel time possible.
This assure that the assumptions of the algorithm are fulfilled and the flow is assigned to the edges in the best possible way and with no degradation of the edges travel time, as the Minimum Cost Flow algorithm, according to the flow conservation principles [1], assumes that the network will be using only up to its maximum flow.
However that is not totally compatible with the BPR function where the cost of an arc $i \rightarrow j$ depends on its fc_{ij}, and a major fc_{ij} can always be supported by the arc in detriment of the TT_{ij}.
In order to overcome this algorithmic limitation, and using our knowledge about the BPR function, we chose to use a second algorithm we developed ("Paliguel Algorithm"), that uses a greedy approach to improve the travel times.

Maximum flow of a network

In order to study a specific network, as well as the maximum flow [2] that can pass through each link, we implemented the Ford-Fulkerson algorithm.
The goal is to develop a generic evacuation plan that ensures that the largest number of vehicles are evacuated from the origin nodes to the destiny nodes, despite the time costs that this could take.
It can also be used to study the viability of a network towards catastrophic events and the effectiveness of the best evacuation plan possible.

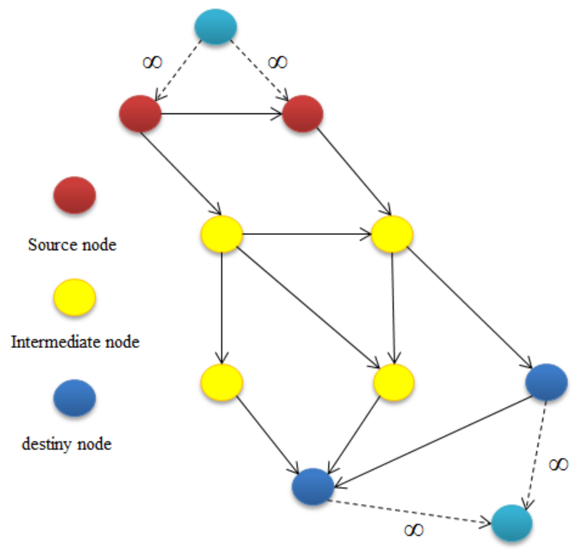

Figure 3: Multi-origin and multi-destiny network

The Ford-Fulkerson maximum flow algorithm only deals with networks with one origin and one destiny.

In a multi-origin network this restriction can be easily overcomed by creating a new node to be considered the network source, and linking this new source to all the "real" ones with arcs with $fc = \infty$. The same idea can be applied in a multi-destiny network, by creating a new node to be considered the network destiny, and linking all the "real" ones to this one with $fc = \infty$.

Pseudocode implementation and analysis

```
1: Graph $residual \leftarrow graph.clone()$

2: for all Arc $(i \rightarrow j) \in A \subset graph$ do
3:     $f_{ij} \leftarrow 0$
4:     $TT_{ij} \leftarrow FFT_{ij}$
5: end for

6: repeat
7:     for all Node $v \in V \subset residual$ do
8:         $v.visited \leftarrow false$
9:     end for

10:    do a DFS search for a source-destiny path $p$ with $fc_{ij} > 0 : (i \rightarrow j) \in A \wedge V_j.visited = false$

11:    if $p$ is valid then
12:        for all Node $v \in p$ do
13:            $v.visited \leftarrow true$
14:        end for

15:        $minf \leftarrow min\{fc_{ij} : (i \rightarrow j) \in p\}$

16:        for all Arc $(i \rightarrow j) \in p$ do
17:            $fc_{ij} \leftarrow fc_{ij} - minf$
18:            $fc_{ji} \leftarrow fc_{ji} + minf$

19:            if $\exists a, \exists b : (a \rightarrow b) \in A \subset graph \wedge V_a = V_i \wedge V_b = V_j$ then
20:                $f_{ab} \leftarrow minf$
21:            else
22:                $f_{ba} \leftarrow minf$
23:            end if
24:        end for
25:    end if
26: until $p$ is not valid

27: for all Arc $e \in A \subset graph$ do
28:     $e.updateBPR()$
29: end for
```

For each iteration of the algorithm, that depends on the edge number ($|E|$) thus using depth-first search, it is assured that the flow value increases by at least one unit in each iteration [3].

For that reason, this algorithm has a temporal complexity $O(|E|f)$ [3], being $|E|$ the number of arcs and f the maximum flow found.

Minimum cost flow of a network

To solve the problem of evacuating a certain amount of vehicles as fast as possible from the origin to the destiny nodes, we used a Minimum Cost Flow algorithm [4].
This algorithm is a generic version of the augmenting-path algorithm for the Maximum Flow problem, as previously presented. The main differences between the two is that in the Maximum Flow, the search for a path follows a DFS algorithm (depth-first search), while the Minimum Cost algorithm follows the shortest path.
This algorithm also receives as input a specific value for the flow to be used in the cost optimization (which in this is the number of vehicles to enter the network).

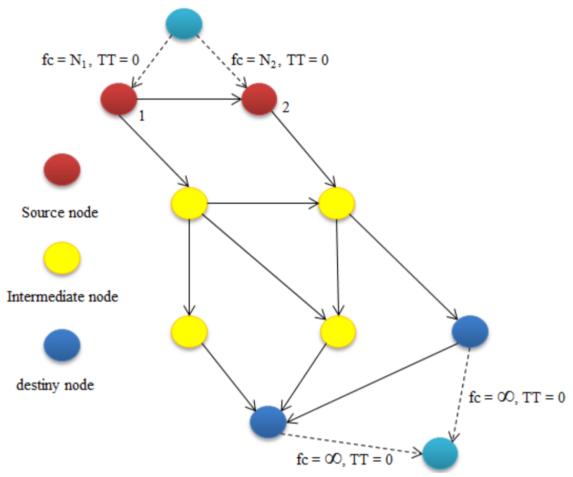

Figure 4: Multi-origin and multi-destiny network

The Minimum Cost Flow algorithm only deals with networks with one origin and one destiny.
In a multi-origin network, this restriction can be easily overcomed by creating a new node to be considered the network source, and linking this new source to all the "real" ones i with arcs with $fc = N_i$ and $TT = 0$.
In a multi-destiny network, this restriction can be easily overcomed by creating a new node to be considered the network destiny, and linking all the "real" ones to this one with arcs with $fc = \infty$ and $TT = 0$.

Pseudocode implementation and analysis

```
1: Graph $residual \leftarrow graph.clone()$
2: $min\_acum \leftarrow 0$

3: for all Arc $(i \rightarrow j) \in A \subset graph$ do
4:     $f_{ij} \leftarrow 0$
5:     $TT_{ij} \leftarrow FFT_{ij}$
6: end for

7: repeat
8:     $residual.dijkstraShortestPath(graph, source)$
```

```
 9:     for all Arc $(i \to j) \in A \subset residual$ do
10:         $TT_{ij} \leftarrow TT_{ij} + V_i.dist - V_j.dist$
11:     end for
12:     for all Node $v \in V \subset residual$ do
13:         $v.visited \leftarrow false$
14:     end for
15:     do a DFS search for a source-destiny path $p$ with $fc_{ij} > 0 \wedge TT_{ij} = 0 : (i \to j) \in A \wedge V_j.visited = false$ in $residual$
16:     if $p$ is valid then
17:         for all Node $v \in p$ do
18:             $v.visited \leftarrow true$
19:         end for
20:         $minf \leftarrow min\{fc_{ij} : (i \to j) \in p\}$
21:         $min\_acum \leftarrow min\_acum + minf$
22:         for all Arc $(i \to j) \in p$ do
23:             $fc_{ij} \leftarrow fc_{ij} - minf$
24:             $fc_{ji} \leftarrow fc_{ji} + minf$
25:             if $\exists a, \exists b : (a \to b) \in A \subset graph \wedge V_a = V_i \wedge V_b = V_j$ then
26:                 $f_{ab} \leftarrow minf$
27:             else
28:                 $f_{ba} \leftarrow minf$
29:             end if
30:         end for
31:     end if
32: until $p$ is not valid $\vee$ $min\_acum \geq flow\_wanted$
33: for all Arc $e \in A \subset graph$ do
34:     $e.updateBPR()$
35: end for
```

For each iteration of the algorithm, that depends on the edge number ($|E|$) thus using depth-first search, it is assured that the flow value increases by at least one unit in each iteration.

Also the Dijkstra Shortest Path algorithm used in the pseucode in each iteration has temporal complexity $O(|E| \log |V|)$ [2]. The interested reader is refered to Dijkstra's original work for more details [5].

The analysis of the pseudocode above and the previous conclusions suggest that this algorithm has a temporal complexity $O(f|E|^2 \log |V|)$, being $|E|$ the number of arcs, $|V|$ the number of nodes and f the maximum flow found.

Improvements to the original algorithm

The minimum cost flow algorithm implemented in this paper was improved when compared to the original one, due to some changes made dealing with the constraints of the problem.

To improve its efficiency, the Bellman-Ford algorithm was never used to calculate the shortest path. In the more general implementation of the Minimum Cost Flow algorithm, the Bellman-Ford algorithm, which has growth order of $O(|V||E|)$ [3] (being $|V|$ the number of nodes and $|E|$ the number of arcs) must be used in the first shortest path determination because it is possible that the edges have negative costs, which Dijkstra algorithm is not able to handle.

Due to our problem′s restrictions it was possible to always use the Dijkstra version because the edge costs are travel times and therefore always have non negative values.

The algorithm used in this section finds a solution to the problem, however this can possibly not be the best. Sometimes a better solution could be found if we put more flow in a route than its capacity. Before analysing that situation more deeply we must have a procedure to retrieve the routes assigned to the network′s vehicles.

Route search

In order to search for routes that evacuate a specific amount of vehicles, we developed an algorithm that finds all paths leading from the origin nodes to the destiny ones.

The search algorithm is based on the depth-first search algorithm that iteratively finds a route and reserves its resources until there is no route left available in the graph.

Pseudocode implementation and analysis

```
 1: Graph $temp \leftarrow graph.clone()$
 2: $routes \leftarrow \{\}$
 3: for all Node $v \in V \subset temp : v = o$ do
 4:     repeat
 5:         for all Node $d \in V \subset temp$ do
 6:             $d.visited \leftarrow false$
 7:         end for
 8:         do a DFS search for a path $p$ $v$-destiny with $f_{ij} > 0 : (i \to j) \in A \wedge V_j.visited = false$ in $temp$
 9:         if $p$ is valid then
10:             for all Node $v \in p$ do
11:                 $v.visited \leftarrow true$
12:             end for
13:             $minf \leftarrow min\{f_{ij} : (i \to j) \in p\}$
14:             $rtt \leftarrow \sum_{i,j:(i \to j) \in p} TT_{ij}$
15:             for all $i, j : (i \to j) \in p$ do
16:                 $f_{ij} \leftarrow f_{ij} - minf$
17:             end for
```

```
18:             $new\_route.nodes \leftarrow p.nodes$
19:             $new\_route.traveltime \leftarrow rtt$
20:             $new\_route.flow \leftarrow minf$
21:             $routes \leftarrow routes \cup new\_route$
22:         **end if**
23:     **until** $p$ *is not valid*
24: **end for**

25: sort $routes$ by $traveltime$ in descending order
26: $max(TT_n) \leftarrow routes.last()$
```

For each iteration of the algorithm, that depends on the edge number ($|E|$) thus using depth-first search, it is assured that the flow value in route found decreases by at least one unit in each iteration.
The sorting algorithm used has temporal complexity $O(R \log R)$, being R the number of routes.
The analysis of the pseudocode above and the previous conclusion suggest that this algorithm has a temporal complexity $O(|E|f + R \log R)$, being $|E|$ the number of arcs, f the maximum flow found by the algorithm and R the number of routes found.

PALIGUEL: AN IMPROVED ALGORITHM

The algorithm presented in previous section allowed us to give a solution to our problem, although in some situations this could not be the best solution, as $max(TT_n)$ could be additionally optimized by running a second algorithm.
In this section we illustrate the critical cases and propose a complementary algorithm to try to overcome these drawbacks.

The problem that the previous algorithm does not cover is when we have a fast link versus a slower link, as the next example illustrates.

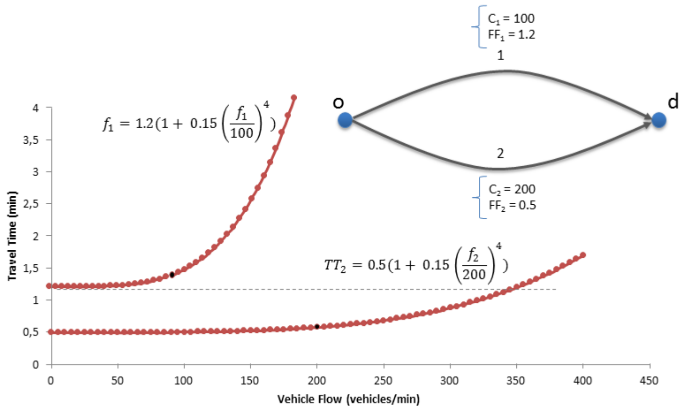

Figure 5: Concrete network and respective arcs

The figure pretends to illustrate the situation when it is better to overuse a link and exceed its fc than changing to a new one. This happens when we have a fast arc, with a better FFT. In that case, even if we exceed its fc and the TT increases, the results are still better than using an empty slower arc with a worse initial FFT. To overcome this, after running the Minimum Cost Flow algorithm and getting the routes that link the origin and destiny nodes and respective travel times, we propose running another algorithm which is now explained.

The idea behind it is that after a first initial solution is found, we should try to exchange flow from the slower routes to the fastest ones in order to improve the worse TT_{R_k}.

Pseudocode implementation and analysis

```
 1: **while** true **do**
 2:     sort $routes$ by $traveltime$ in descending order
 3:     $i \leftarrow routes.last()$                    ▷ slowest route
 4:     $f \leftarrow routes.first()$                   ▷ fastest route

 5:     **if** $routes.size() = 1$ **then**
 6:         end algorithm
 7:     **end if**

 8:     $graph.travel\_time \leftarrow i.traveltime$

 9:     **while** $f \neq routes.last()$ **do**

10:         **if** $i.nodes.first() = f.nodes.first() \wedge f_{R_f} \leftarrow f_{R_f} + 1$ *implies* $TT_{R_f} \leq max(TT_n)$ **then**
11:             $f_{R_f} \leftarrow f_{R_f} + 1$
12:             $f_{R_i} \leftarrow f_{R_i} - 1$

13:             **if** $f_{R_i} = 0$ **then**
14:                 $routes.delete(i)$
15:                 $i \leftarrow routes.last()$
16:             **end if**

17:             break from While cycle
18:         **else**
19:             $f \leftarrow routes.nextElement(f)$
20:             **if** $f = routes.last()$ **then**
21:                 end algorithm
22:             **end if**
23:         **end if**
24:     **end while**
25: **end while**
```

The analysis of the pseudocode above suggest that this algorithm has a temporal complexity $O(f|E||V| + fR \log R)$, being $|E|$ the number of arcs, $|V|$ the number of nodes, R the number of existing routes and f the network flow.

PRELIMINARY RESULTS AND ANALYSIS

To evaluate the algorithms here described, their performance and result accuracy was tested. In order to achieve that, the time before and after calling the function was measured and subtracted. The results of the elapsed time are shown in the figure 6.

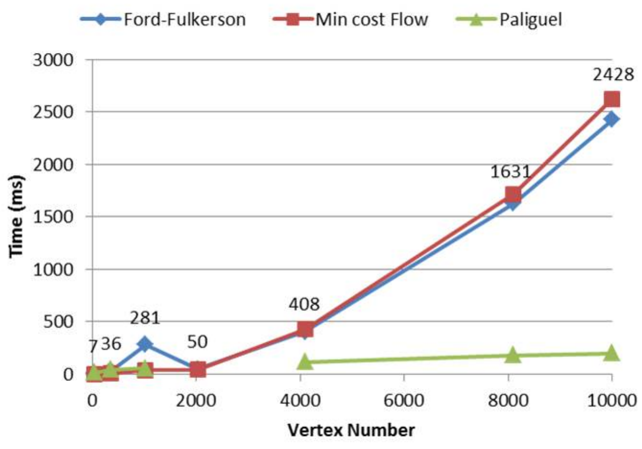

Figure 6: Algorithm time performance in function of vertex number

The results for each function are consistent with the behavior and theorethical complexity described earlier. In what concerns the Ford-Folkurson and Minimum Cost Flow algorithms, their performance is highly affected by the number of vertices of the graph.
However, this is not the case with the Paliguel algorithm. Since the algorithm already works with the final routes, its complexity depends on the number of routes obtained and also on the flow that travels it. Therefore, in the previous graph it is noticeable that the performance of the algorithm isn't affected by a higher number of vertices.

CONCLUSIONS AND FUTURE WORK

The aim of this study was to develop a strategy to address the problem of assigning routes to vehicles in an emergency situation. We conceived a model that supports both multiple sources and multiple destinies.
In order to minimize the network travel time we started searching among maximum flow and minimum cost flow algorithms, and chose to implement the Ford-Fulkerson Maximum Flow and its generalization with the Minimum Cost Flow. Both algorithms worked as an initial starting point to our solution.
However due our travel time definition using BPR function, the use of the Minimum Cost Algorithm itself does not guarantee the user equilibrium of the network. This forced us to improve this solution by using a complementary algorithm, that follows a greedy approach.

Although our algorithm improved the solution given by the previous algorithm, it does not insure user equilibrium either, as there may be better solutions not being considered. It would then be relevant to find an heuristic method to improve this algorithm and allowing transferring more than one unit of flow from a route to another in the same iteration, for example.

REFERENCES

[1] Sheffi, Y. *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods.* Prentice-Hall, Inc., Englewood Cliffs, 1985.

[2] Weiss, Mark Allen. *Data structures and algorithm analysis in C++. (2nd ed.)* Reading, Massachusetts: Addison Wesley, 1999.

[3] Cormen, Thomas H., Leiserson, C., Rivest, R. *Introduction to algorithms,* Mcgraw-Hill Science/Engineering/Math.

[4] Sedgewick, Robert, *Algorithms in C++ Part 5: Graph Algorithms*, 3/E, Addison-Wesley Professional, 2001.

[5] Dijkstra, Edsger W. A note on two problems in connexion with graphs. *Numerische Mathematik* 1: 269271, 1959.