DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Informe Técnico / Technical Report

Ref. No: DSIC-II/17/07 **Pages: 35**

Title: A Distributed Shortest Path Algorithm for Global Automated Transport System

Author (s): Frédérique C. Versteegh, Miguel A. Salido

Date: 13/07/2007

Key Words: distributed CSP, transportation

VºBº **Author (s):**

Leader of Reasearch Group

A Distributed Shortest Path Algorithm for Global Automated Transport System

F.C. Versteegh[1] and M.A. Salido[2]

[1] Twente University, Enschede, the Netherlands
f.c.versteegh@student.utwente.nl

[2] DSIC, Universidad Politecnica de Valencia, Spain
msalido@dsic.upv.es

Summary. During recent years the development of automated traffic systems has received increased attention, and substantial effort has been invested in trying to find a solution to problems associated with road transport. Among these problems are road accidents caused by human-related factors, such as tiredness, loss of control, a slow reaction time, limited field of view, etc. A further transport-related problem is that of loss of time which may be caused by slow driving speed due to weather conditions, road conditions, visibility, and traffic congestion for example. In this paper, we present a global road transportation system, which is being developed by several European Universities. The main goal of the algorithmic part is to develop algorithms capable of creating driving schemes for a vehicle from any arbitrary address to any other arbitrary address (in the address space of the system), while considering, and if necessary adapting, the driving schemes of other vehicles traveling in the system at the same time according to priorities, driving and optimization rules. To this end, distributed CP techniques are necessary to solve these problems.

Key words: route planning, distributed algorithms, shortest path problem.

1 Introduction

The Global Automated Transport System (GATS) [9] is a driver-less, integrated transport system. It has the astonishing ability to simultaneously coordinate the macro and micro needs of road transport networks. Millions of vehicles can be optimally, simultaneously and automatically "driven" over a virtually unlimited geographic region, including whole continents, while at the same time, the requirements of each individual vehicle and its passengers attended to. It is an innovative concept, based on simple, recognized principles and proven technologies.

Its application will revolutionize road travel by dramatically increasing safety, reducing congestion, and eliminating driving-associated stress and fatigue. The consequence will be an overall improvement in the quality of everyday life.

Due to its decentralized, modular architecture it can be implemented with the same ease and simplicity in small contained areas such as airports and theme parks as in larger areas such as local, national and international road systems.

The Global Automated Transport Systems has the potential to be the transport system of the future. On the other hand, its simplicity and ease of implementation can make it the transport system of the future and a creative solution to pressing problems of the present.

GATS large scale implementation involves a paradigm shift and a revolution which is impossible to achieve with one stroke. This project will be the introduction of this concept to the market. Therefore, it is aimed at developing the algorithm platform and the hierarchical control structure and proving the concept through a computer simulation and a laboratory test-bed.

GATS aims to provide an automated road-vehicle transport system which do provides an answer to known traffic problems. This driver-less automated transport system has the following highlights:

- Fully automatic driving of vehicles on all kinds of roads and all kinds of intersections

- Virtually unlimited volume of traffic, potential of system growth and geographical expansion
- Decentralized, hierarchical modular architecture
- Integrated, real-time communications and control among all system components
- Real-time personalized vehicle identification, localization and control
- Real-time, automatic sensing of driving conditions anywhere and of the functioning and safety conditions of each individual vehicle
- Real-time optimal navigation in accordance with the dynamically changing requirements of passengers, vehicles and road conditions " Management of an entire regional or continental traffic system.

The basic goal of the report is the development of an algorithm for determining the optimal path between two points in a distributed network. Due to virtually unlimited geographic regions, including whole continents, the problem is very hard to solve, and distributed techniques must be carried out to solve this problem. Furthermore, the algorithm must consider user requirements, weather conditions, etc.

2 Terminology and Integrated Functioning of GATS

Following, we summarize the internal structure of GATS. In the centre of a traffic lane, 10-15 cm under the road surface lies an "intelligent cable". This cable comprises tiny Road-Units (RUs) located at fixed distances (approximately 3 meters) from each other. While driving, the vehicle sends short radio transmissions down towards the RUs at regular time intervals. The RU receives a transmission, processes it and responds with a radio transmission back to the vehicle. As the vehicle moves along the road, it communicates with the RUs one after the other instantaneously, so the vehicle has continuous radio communication with the RUs. The RUs are connected to each other by two means: directly, by Serial Buses (red lines in Figure 1), and indirectly, by Parallel Buses (blue lines in Figure 1).

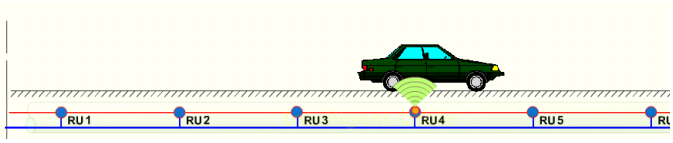

Fig. 1: RUs are connected by Serial Buses and Parallel Buses

The memory of each RU stores the specifications of the RU and individual driving instructions that it will transmit to each vehicle above it. Several hundreds of consecutive RUs constitute a Segment, whose functions are administered by a Segment Controller. The Segment Controller is connected to its RUs through the Parallel Buses and is responsible for driving the vehicles passing in its domain, performing routine maintenance check-up of the components in its Segment and monitoring and regulating their mutual performance. At level m, A group of adjacent Segment Controllers has a superior Controller: Level m-1 Controller, which coordinates and controls their individual and mutual functions. A group of adjacent Level m-1 Controllers has a superior Controller: Level m-2 Controller. This goes on hierarchically (Figure 2). The Level 0 Controller coordinates and controls the functions of the whole system. There is virtually no limit to the number of levels and to the size of the system's geographic domain.

Assume a vehicle is in a parking lot above a RU. The passengers turn the vehicle on, which begins to send short radio transmissions down towards the road. The RU detects those transmissions and responds with radio transmissions back to the vehicle. The RU initiates a communication session with the Segment Controller, in order to inform it about the new event. The passengers in the vehicle enter their requirements as destination, priority, preferred routes etc. The vehicle's processor sends a message to the Segment Controller which includes the requirements, the exact location of the vehicle relative to the RU and its own specifications. The Segment Controller processes the request while considering additional inputs from other RUs in the Segment and from its superior Controller. Finally it prepares a driving instruction message for each RU in the Segment. The RUs will send these instructions to the ve-

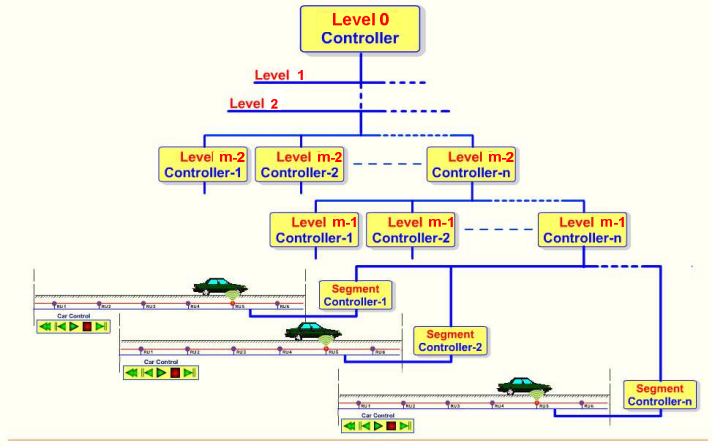

Fig. 2: Hierarchical structure of the system

hicle when it passes above them. Each message includes an addressee (RU1 etc.), a vehicle ID, the expected arrival time of the vehicle to the RU, the speed that the vehicle should travel at and the driving direction. When the vehicle is driving from one RU to another, the active RU uses the Serial Bus to inform the next and the previous RUs in the sequence about the exact timing, the ID number and other specifications of the moving vehicle. If the RUs detect intolerable deviation from the plan they can initiate a so-called Emergency Braking Procedure. The active RU uses the Parallel Bus to inform the Segment Controller with the same information of the moving vehicle.

3 Problem Requirements

In this section we present the problem requirements for the algorithm we will develop later.

The GATS controller hierarchy (Figure 2) can be viewed as a network of processors whose computational tasks are accomplished in a distributed manner. The cars are "objects" that have to be transferred from a source to a destination. In graph theoretic terms, the planar graph on which the cars travel is partitioned into a set of sub-graphs, each managed by a subset

of the controllers. The partition is refined recursively as we go down in the controllers' hierarchy tree.

The idea of the universal algorithm, which will supervise the vehicles in any segment and any level, is supposed to be the best solution. The origin and destination of a vehicle determine which controllers are required to calculate the shortest path. From the hierarchical structure as explained in section 1 follows that a border of segments or controllers that has to be crossed requires the use of a higher level controller in order to communicate with the lower level controllers. The task of every controller is the same, namely calculating the shortest path between two controllers or segments at the level below, until the lowest level of segments is reached. As the level of controllers is virtually unlimited and to reduce costs, the algorithm should be the same for every controller at every level. This requirement calls for the use of a distributed and scalable algorithm.

To be able to approach the real-time travel time as close as possible, several other factors, next to distance and maximum speed, need to be taken into account. The choice of a path and driving speed for a vehicle should be adapted to its individual characteristics, including vehicle and driver related factors such as driver experience and vehicle condition. Next to these static factors, the algorithm should be able to take into account dynamic factors such as weather and traffic density. Upon the entry of a vehicle in a segment, the Segment Controller will have to determine the vehicles probable point and time of exit. It will then have to set a plan accordingly. This plan is susceptible to changes underway since it will be influenced by other vehicles using the same segment. This means that the route has to be recalculated along the way to be sure the shortest path is continued to be chosen.

The hierarchical structure as explained in section 1 implies that every controller only has information about the segments it controls in one level below. It does not have any information about what happens in other segments or in other controllers. A level of privacy between controllers and segments has to be maintained by the algorithm.

A Distributed Model for GATS

Traditional Centralized techniques fail to model and implement problems of this type due to their complex and large nature. Due to the decentralized and modular nature of the architecture, the algorithms to calculate the scheduling of each vehicle must be distributed. Figure 3 shows the map of Europe to be distributed/divided into regions (countries); each region is divided into sub-regions, and so on.

Fig. 3: Map of Europe to be distributed.

Briefly, the system is composed by a network, where nodes are locations and arcs are roads. Depending on the granularity, nodes are points in the road or regions in a country. In the lower level of the system, each RU is represented by a variable (see figure 4). The system may be composed of millions of RUs. This problem cannot be managed by current distributed CP techniques by using a variable per agent. By using these approaches, the problem generates millions of agents and messages passing in the interaction scenarios. This makes the resulting DisCSP unmanageable.

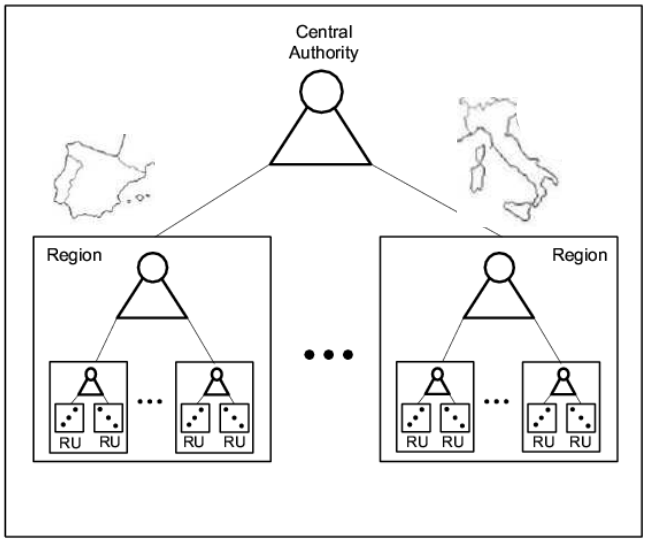

Fig. 4: Distributed model with a central authority.

To overcome this weakness, we use the distributed model presented in [6] in which the problem is partitioned into subproblems that represent regions, countries, etc (see Figure 4). Here, we use a Holonic architecture [3][5] to organize the entities (holon or agent [2]) that are responsible for solving each subproblem. A holon is an autonomous and cooperative unit that can be seen as a whole and a part [5].

The distributed model generated for this scheduling problem follows the following guidelines.

- The number of subproblems depends on the size of the system. A holon can represent a track between two traffic lights or represent a region or a country. Figure 4 shows two holons that represent two countries, Spain and Italy. Each of them is composed of a set of sub-holons that represent regions, and each sub-holon is composed by new sub-holons that represent sub-regions and so on. The base case is composed of individual variables that represent RUs.
- The execution of the subproblems is carried out in two steps. First, given the requirements of the passenger, (the destination is the most important

requirement), the central authority is the *Level i Controller* that involves both origin and destination. This *Level i Controller* is committed to solving the shortest path in a high level problem (each node is a region). This path is only a first approach that guides us to find the real shortest path. Thus, *Level i Controllers* are executed first, then all *Level i+1 Controllers* are executed concurrently and so on. Depending on the size of the journey, several hierarchical levels are necessary. Finally, the calculated route is sent to the *Segment Controllers* that are involved.

- Due to the dynamic structure of the problem, some parts of the system may change and new schedules must be calculated. The rescheduling is only calculated from the incidence to the destination. The management of backtracking is carried out in a way similar to the railway scheduling problem distributed by stations.
- The nature of the system makes the presence of a central authority necessary. However, due to the scalability of the system, the central authority has the same behaviour as a level controller. The central authority is the minimal level controller that involves both origin and destination. This level depends on the problem instance.

4 A Developed tool for GATS

The algorithm we propose will determine the shortest path from the origin to the destination the driver wants to go to. Without loss of generality we consider only two levels in the description and the implementation of the algorithm.

For example first we have a graph as in figure 5, which we have to divide to create levels as explained earlier (see figure 17 for the graph divided in levels). As an example in this report we use Spain, in which level 0 consists of the provinces and level 1 consists of the nodes in a region (see figure 7. In figure 6 both the origin and destination are defined as a region of Spain, by the number of the province, and a node, the number of the village for example. Within Spain 52 provinces exist, including the islands and overseas

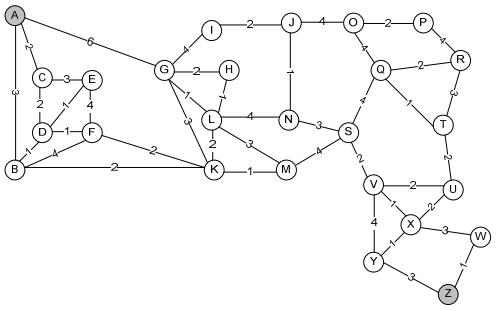

Fig. 5: Initial weighted graph that represents locations (nodes) and best time to go from a location to another one (arcs).

areas in the north of the African continent. In every province 100 nodes are determined, which are randomly distributed over the province.

Fig. 6: Spain divided in 52 regions

Once the origin and the destination have been set, the driver has to fill in the driver and car related data: the experience of the driver, type of the

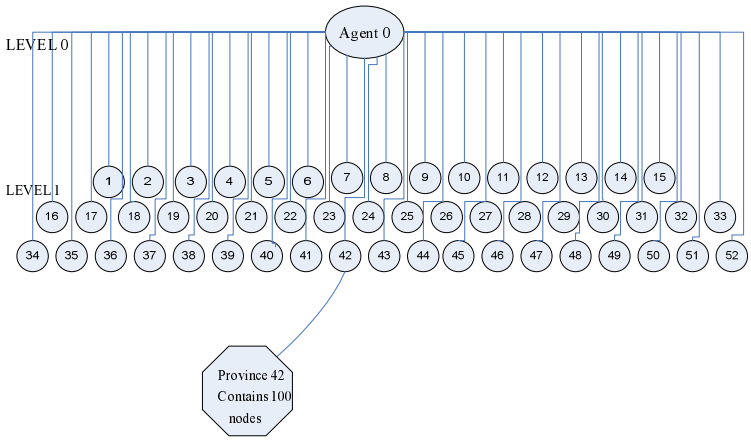

Fig. 7: Structure of levels

vehicle, the age of the vehicle, the horse power of the vehicle and the weight of the vehicle. The experience of the driver is divided in two parts, first the driver has to indicate whether he is novel or not novel. If the driver is not novel, the level of driving experience has to be indicated. All the factors to be filled in influence the time a vehicle will need to travel the path from origin to destination. For example, a relatively inexperienced driver will drive at a lower speed than an experienced driver, a truck will go slower than a motorbike and a new car will go faster than an old one.

As soon as all these data are filled in, the driver can let the system determine the shortest path from origin to destination by pushing the 'Calculate Shortest Path' button (see figure 8). The result will be the shortest path through regions and nodes (see figure 9). The regions through which the path goes will be showed in the interface, as well as the total route through all nodes and regions and the total travel time needed for this specific vehicle and this specific driver (see figure 9). The distances between provinces are real distances, so the route through provinces is in fact the shortest path from start province to destination province. On the other hand the distances between nodes in a region are randomly generated, which means that the final travel time is not made up of real data.

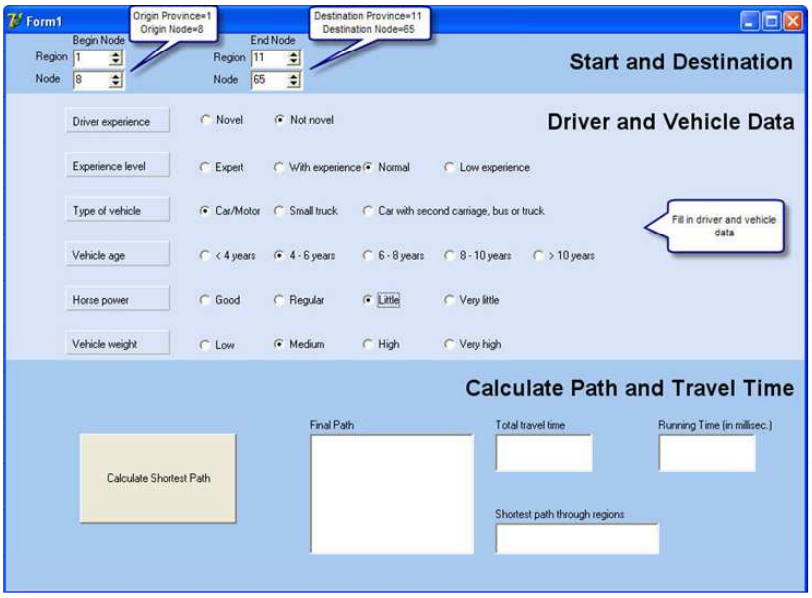

Fig. 8: Fill in data in the interface

The algorithm calculates the shortest path in a distributed way. First the shortest path from start province to end province is determined. Once the system knows through which provinces the path should go, it lets every province determine all possible shortest paths within the own province from all frontiers with the preceding province to all frontiers with the next province. The start province calculates all shortest paths from the start node to the frontiers with the next province. The last province calculates all shortest paths from all frontiers with the preceding province to the end node. Using these data the shortest path from the start node to the end node is determined. This way of calculating the shortest path avoids that provinces need to know information from each other, so privacy is guaranteed.

5 A Distributed Algorithm

The algorithm we present consists of several parts, which we will discuss first one by one, to finally come to the description of the complete algorithm.

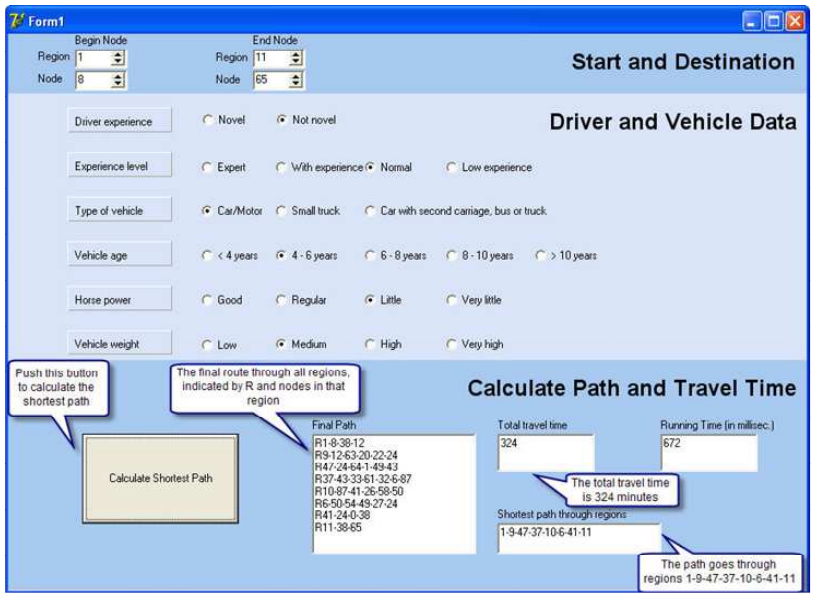

Fig. 9: Result of calculating the shortest path

The detailed pseudocode of the described procedures can be found in the Appendix.

- The procedure *ReadFile* (figure 19) captures data from a file concerning a province at level 1. For every track in a province at level 1, two base times are determined: a normal one, taking into account the distance and the maximum allowable speed, and one for inexperienced drivers, as in some countries, for example in Spain, drivers who have their license less than one year have to keep to a lower maximum allowable speed. Obviously this so-called novel base time is always equal to or larger than the base time on the same track. Every file of a province at level 1 consists of data from every node to every other node within the province, the base time, the novel base time and the inclination on that track (see figure 10).
  Also data about the weather condition and the density on a track are shown in the file. Although these data are dynamic in reality, we use static and randomly chosen data now. This can be changed easily later if in the

Origin	Destination	Base time	Novel base time	Inclination	Density	Weather
0	23	30	33	0	1	2
0	24	-1	-1	0	0	0
0	25	28	31	3	1	5

Fig. 10: Example of data at level 1

file data about weather condition and density would be captured from for example the internet every time the file is read. If there is no direct path from a node to another node, base time and novel base time are both set to -1, factors concerning inclination, weather and density are set to 0 (see figure 10). A list of arcs is created in which for every arc, origin node, destination node, base time, novel base time, inclination, density and weather are saved. While reading the data from the file, the number of nodes that take place in the file is counted. At last matrices are created and filled with the data in every arc about base time, novel base time, inclination, density and weather.

- The procedure *SetTotalTime* (figure 20) converts the base time or the novel base time for a track, which would be needed by a vehicle and driver in ideal conditions, into a real time including the specific driver and car related factors given by the driver. Whether the base time or the novel base time is converted depends on the experience of the driver. Every factor deteriorates the total time by a percentage of the base time. This percentage is calculated by means of a formula for every factor. The percentages we use are intuitively chosen and can be adapted to other situations. The base time or novel base time is accumulated with the percentages of all factors, which results in the so-called real time. If a driver is novel we assume that for being novel, the novel base time deteriorates by 10% of the difference between the novel base time and the base time. If a driver is not novel we assume that every level of experience contributes for 5% of the base time to the real time. The less experience a driver has, the higher the level is and the higher the resulting percentage will be. The condition of a vehicle is a combination of the horse power and the age of the vehicle. Every level of condition contributes for 2% of the base time

to the real time. The influence of the inclination on the real time depends next to the inclination of the track also on the weight and the condition of the vehicle. The multiplication of the level of inclination with the sum of the level of condition and the level of weight of the vehicle results in the level of inclination. Every level of inclination contributes for 5% of the base time to the real time. Every level of weather and every level of density both contribute for 10% of the base time to the real time. The type of the vehicle is divided in three categories that all represent a value as level. Every level of the type of vehicle contributes for 5% of the base time to the real time. Finally the summation of the (novel) base time and all the factor-dependent deteriorations results in the real total time for a track. Figure 17 shows an example of a weighted graph with for each arc the base time and the final weighted graph including all restrictions given by the driver.

- In the procedure *CalcRealTime* (figure 21) for every path between two nodes the real travel time is calculated by means of the procedure SetTotalTime. This real travel time is stored in a matrix.
- In the procedure *ShortestPath* (figure 22) the shortest path from a start node to a destination node is determined. This procedure will be used at every level in the hierarchy. We calculate this path using *Dijkstra's shortest path algorithm*. This algorithm works by keeping for each node the distance found so far from the start node to this node. Initially this distance is 0 for the start node and infinity for all other nodes, as we do not know a path yet that leads to these nodes. For each node we keep the status: if it is already visited or not and we keep the precedent node in the path for every node. Initially the visited status for all nodes is false and the precedent node is labeled to -1 except for the start node, which is its own predecessor. While the end node is not visited, the algorithm selects from all nodes that are not visited yet, the node closest to the start node, which means the node with the smallest distance. The status of this node is labeled to visited and for this node every other node that is not visited yet and to which a direct path from the selected node exists is examined. If

the distance from the start node to this node is larger than the sum of the distance from the start node to the selected node and the distance from the selected node to this node, then the distance from this node changes to the value of the sum and the precedent node of this node is labeled to the selected node. This process continues until the status of the end node is set to visited.

```
Procedure DetermineAgentZeroPath

    begin
    Read and store in arcs all data from file level 0
    Save data from arcs in matrix
    Define Start as begin region and Destination as end region
    ShortestPath
    Write path through regions in array
```

Fig. 11: Procedure DetermineAgentZeroPath

- In the procedure *DetermineAgentZeroPath* (figures 11, 23) the shortest path from start region to end region at level 0 is determined. This means that a path only considering provinces is made up; no nodes inside regions are taken into consideration. The result of this procedure is an array in which the provinces through which the shortest path goes are described. In this procedure, first the file with data of level 0 is loaded.

  The file concerning level 0 contains for every province to every other province with which a direct path exists, in other words for every two neighbor provinces, the distance between them. If no direct path exists the distance is set to -1 (see figure 12). Reading this file is not done using the procedure ReadFile, as in this case only the distance between two adjacent provinces has to be captured and only one matrix has to be filled with data. We use arcs to read this file. Every arc only contains the origin

Origin region	Destination region	Distance
2	13	207
2	14	-1
2	15	-1
2	16	142
2	17	-1
2	18	363

Fig. 12: Example of data at level 0

province, destination province and the distance between them. Reading the file, data are stored in arcs and the number of nodes existing in this level is counted. Once the end of the last line in the file is reached, the distances between provinces are stored in a matrix. Finally Start is defined as the begin province and Destination as the end province. Using the procedure ShortestPath, the shortest path from Start to Destination is calculated. The number of iterations in the while-loop of the algorithm is counted. If the number of iterations reaches the total number of nodes, no possible path can be found from Start to Destination, so the algorithm will stop. In an array the predecessor of every node is saved. This array can be used to find the shortest path from Start to Destination. The path through regions is found by starting at the end region and from there on working backwards to the begin region by iteratively determining the predecessor of a region.

- The procedure *DetermineLevelOnePath* (figures 13, 24) determines for every province involved in the path determined by DetermineAgentZeroPath. Agent 0 gives all involved agents at level 1 the order to calculate the shortest path from all frontiers with the precedent province to all frontiers with the next province. For the begin region the shortest path from the begin node to all frontier nodes with the next province are calculated. For the end region the shortest path from all frontier nodes with the precedent region to the end node are calculated. The sequence of provinces is given by the shortest path calculated by agent 0. This procedure is only executed if the predecessor of the end province is unequal to -1, which means that

```
Procedure DetermineLevelOnePath

begin
    Calculate in the end region the shortest path from every
        frontier node with the precedent region to the destination node
    Save found distance in cube
    For all regions between begin and end region
        calculate the shortest path from every
            frontier node with the precedent region to every frontier
            node with the next region
        Save found distances in cube
    Calculate in the begin region the shortest path from the begin node
        to every frontier node with the next region
end
```

Fig. 13: Procedure DetermineLevelOnePath

a path to this province is found. For every two adjacent provinces frontier nodes are determined. For every frontier between two provinces, a set of nodes are randomly labeled as frontier nodes. For the implementation we used the following definition of frontier nodes. For example if node 6 is a frontier node between provinces 1 and 2 then node 6 in province 1 is at the same location as node 6 in province 2. This node is in this case exactly on the frontier of the two provinces (see figure 14). In general we describe a frontier node as an artificial point that is shared by two regions. This point can be on an arc, just where the frontier line between two regions is (see figure 17).

The paths in every province are concurrently calculated by first reading the file with data of the province by ReadFile and calculating the real travel times of these arcs by CalcRealTime. For every two nodes for which the shortest path has to be calculated, this is done by the procedure ShortestPath. If the destination node can be reached from this frontier node, in

other words, if the predecessor of the destination node is unequal to -1, the distance from one frontier node to the other is saved in a cube, in which the region, origin node, destination node and distance are saved.
If the driver wants to go to a destination within the same province, in other words, if the begin province and the end province are the same, of course no frontier nodes have been determined. So if this is the case, the part of the procedure that is described above will be passed. Therefore, at the end of the procedure an if-statement is added for the case that the begin province is the same as the end province. In this case Start is defined as the begin node and Destination is defined as the end node. Using these data ShortestPath is executed and the distance is saved in the cube.

- In the first part of the procedure MakeFinalPath (figures 15, 25) the shortest path from begin node to end node via frontier nodes is determined, using the paths from frontier node to frontier node that are saved in the cube in procedure DetermineLevelOnePath. In the second part of the procedure MakeFinalPath (see figure 26) the complete path through every region will be determined. The first result of MakeFinalPath will be an array in which the begin node, a frontier node for every frontier that is in the shortest path at level 0 and the end node are saved. We calculate this path using a modified form of *Dijkstra's shortest path algorithm*. In this procedure not only the visited status, distance from start and predecessor of a node are saved, but all these data are saved for region-node combinations. This is necessary because in every region the same node numbers appear (see figure 14). A certain point in Spain therefore has to be defined by a node and a region number, whereas two region-node combinations define the same geographical point if the regions are adjacent, the node number is equal for both combinations and this node is defined as a frontier node for the frontier between these regions. In the cube we saved information about the distance from every frontier node to every other frontier node within one region. Therefore, when the algorithm selects a node, which is the frontier with the next region, from this node there is no path in the cube to the next region. On the other hand, from that same node, defined in the next

region, which is the same point in Spain, there are paths defined which lead to the frontier with a further region.

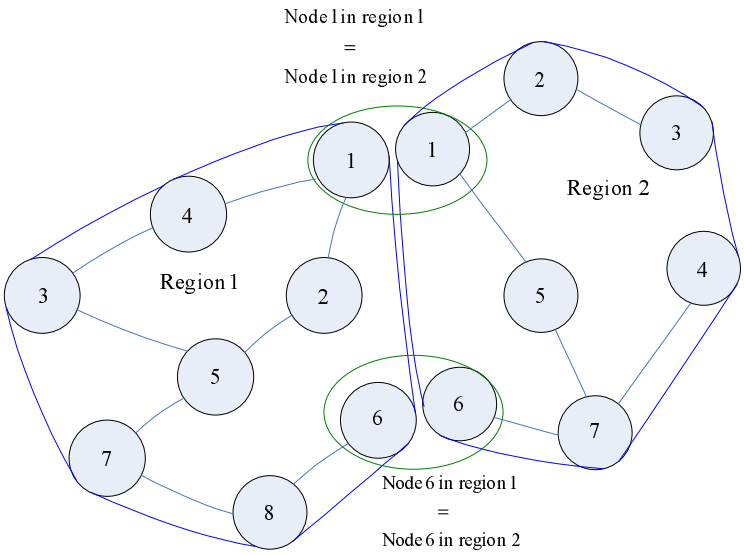

Fig. 14: The same node exists in two adjacent regions

In the procedure the node-region combination that is selected should always be at the frontier with the precedent region, as in this case paths exist to the frontier with the next province. To establish that only these combinations will be selected, also the distance of the frontier node in the next region will be changed, which characterizes the same geographical point. For example if a path leads through regions 1-2-3, first the start node in region 1 will be selected, then all nodes which are reachable from this start node, the nodes at the frontier with region 2, are examined to see if their distance from the start node can be optimized. If this distance can be optimized, not only the distance of this frontier node in region 1 will be changed to a better value, but also the distance of the same node in region 2 is optimized. The part of the algorithm that selects the smallest distance will find two nodes with the same distance, namely the frontier

```
Procedure MakeFinalPath

begin
    Calculate shortest path from begin node to end node with
        Dijkstras Algorithm using region-node combinations
    Save path through frontier nodes in array
    For every region in shortest path
        calculate shortest path between the frontier nodes saved in
            array
end
```

Fig. 15: Procedure MakeFinalPath

node in the two adjacent regions. To make sure that always the one in the next province is selected, in the example that the node in region 2 is selected, the search procedure searches the regions in the order in which they appear in the shortest path. In this way the node at the right side of the frontier will always be selected. Once this right side of the frontier is selected as node-region combination also the visited status of the node at the other side of the frontier has to be set to true, otherwise this node will be selected in the next iteration as being the node with the smallest distance. Of course the visited status of the node precedent region will not change if the selected node is in the begin region, as there is no precedent region for the begin region in the path. Only the predecessor of the node at the side of the frontier from which a path to a further region can be found will be set to the node that is selected at that moment, only this node is needed to find the complete shortest path through frontier nodes. If the end node in the end region has been visited, the array as described above will be made. This has to be done in a recursive way as from every node we only know its predecessor. Beginning at the end node will lead us back to the begin node via the shortest path. An exception has to be made when

the end node also is the frontier node of its own region with the precedent region, as its predecessor will be a node in the precedent region and the array will be one value smaller than the content we described above.
In the second part of this procedure (figure 26) the path found in the first part and the path through regions found in DetermineAgentZeroPath are used to find the complete path from begin node-region to end node-region. For every region in the path, the shortest path from the frontier node with the precedent region to the frontier node with the next region is determined. This is done by getting the first region from the list and the first two nodes from the list, ReadFile reads the data from the selected region, CalcRealTime converts these data to real travel times and ShortestPath uses the two selected nodes, which are defined as start and destination, to calculate the shortest path between these two nodes. Finally for every region the path is written in a string and shown in a memo in the interface.

- The *main algorithm* we propose (figure 16) uses the procedures described above to calculate the shortest path between the origin node and region the driver fills in and the destination node and region he wants to reach. After all arrays and the list of arcs are emptied, first the procedure DetermineAgentZeroPath is executed. If there exists a path from the begin region to the end region, in other words if the predecessor of the end region is unequal to -1 and the number of iterations made in the procedure is smaller than or equal to the number of nodes, the algorithm continues. If not, a message will be shown that there is no possible route. If the algorithm continues the procedure DetermineLevelOnePath is executed, followed by the procedure MakeFinalPath. Finally the total distance and the path through regions are shown in a memo in the interface.

6 Example

In this section we will give a small example of how the algorithm works. We use the graph as shown in figure 5.

```
Distributed Shortest Path Algorithm
```

```
begin
    DetermineAgentZeroPath
    node=predecessor(end region)
    if number of iterations<=number of nodes and node<>-1 then begin
        DetermineLevelOnePath
        MakeFinalPath
        Show total distance in interface
    end
    else
        show in interface that there exists no possible route
    end
end
```

Fig. 16: The complete distributed shortest path algorithm

At the left side of figure 17, a graph is shown, which is is divided in four regions, each region controlled by an agent. At every arc the base time for that arc is shown. Every frontier between regions consists of frontier nodes, which divide arcs in two parts. At the right side of figure 17, the base times of the graph at the left side are converted into real travel times, taking into account all factors as described in sections 4 and 5. At the left side of figure 18, the shortest path from node A to node Z is shown by bold lines. If for some reason the travel time on a track increases a lot, the agent can dynamically calculate an alternative route. This is shown at the right side of figure 18, where the travel time on arc S-F1 increases from 2 to 20.

The alternative shortest path from A to Z will then be different in agent 3 and 4, by going through nodes S-Q-T-F2-U-X-W-Z. This alternative route is shown in bold lines at the right side of figure 18. How exactly the alternative path is calculated is not yet determined by this algorithm, we will discuss this in section 8.

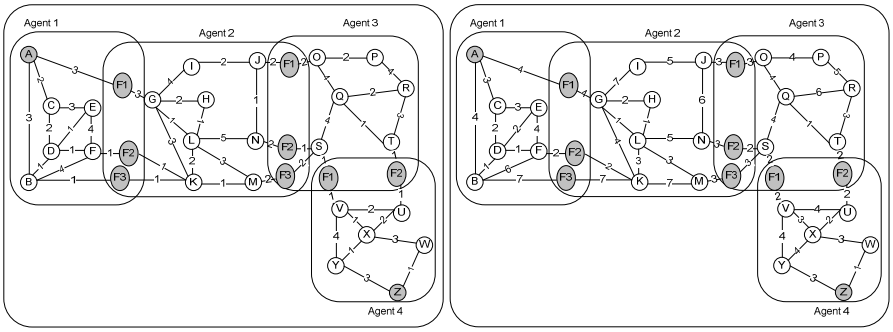

Fig. 17: (Left) Weighted graph distributed into a set of agents. Each agent is committed to a set of variables. (Right) Final weighted graph satisfying all constraints.

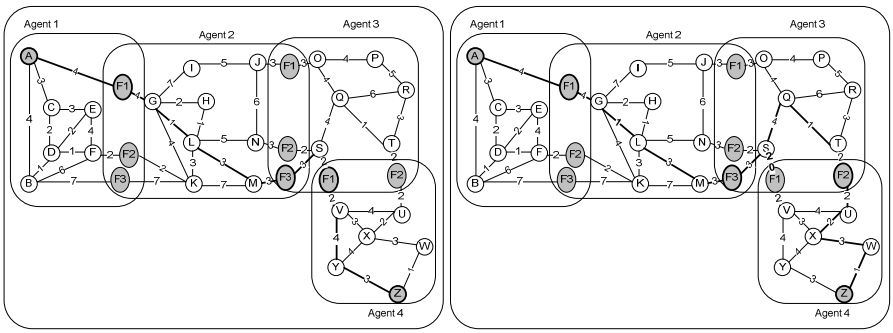

Fig. 18: (Left) Weighted graph with shortest path from node A to node Z.Weighted graph with alternative shortest path from node A to node Z if path S-F1 is congested.

7 Related works

Often in distributed shortest path algorithms the individual nodes need to be able to locally compute shortest paths. In a lot of cases this is done by a variation of *Dijkstra's Algorithm*, the most well-known and fastest basic algorithm for determining the shortest path from one location to all other possible locations in a network.[1] The use of *Dijkstra's Algorithm* alone is not sufficient for GATS as we need a distributed algorithm and Dijkstra only provides a local shortest path algorithm. In this section we discuss some of the existing distributed shortest path algorithms and their possible usefulness for GATS.

The *Link-State Algorithm*, the *Distance-Vector Algorithm* and other derived algorithms are distributed shortest path algorithms in which each node is responsible for calculating a shortest path to all other possible destinations [4]. Although these algorithms are called distributed algorithms, they are distributed in a different way than we use it it this report. No agents are used, therefore the privacy of the controllers cannot be guaranteed in these algorithms.

The well-known commercial navigation systems use, according to Sanders and Schultes [7], heuristics to compute paths. The basic idea of these heuristics is the observation that shortest paths in general use small roads only locally, that means, at the beginning and at the end of a path. This heuristic algorithm only performs a local search around start and destination and then switches to search in a network that is much smaller than the complete graph and which only consists of the most important roads. This heuristic algorithm does not take into account real-time and vehicle and driver specific data and so it not useful for GATS.

The concept of *highway hierarchies* [7] is a multi-level highway network in which iteratively every path consisting of nodes with degree two are replaced by a single edge. Although this approach has a lot in common with the system needed for GATS, this approach is not useful as privacy of regions is not guaranteed, nor involving real-time data is possible.

None of the shortest path algorithms described above suffice all the requirements for GATS as found earlier in this section. This motivates us to propose a distributed shortest path algorithm in which privacy is maintained, dynamic factors are taken into account and in which the algorithm for each controller is the same. In our approach we combine multi-agent theory [8] with *Dijkstra's Algorithm* to come to an algorithm that is useful for GATS.

8 Conclusions and further research

In this section we will discuss the conclusions of this work and we will do some recommendations for further research.

The algorithm we propose in this work is meant for use in GATS. To be useful for GATS the algorithm has to suffice some requirements as described in section 3.

The first requirement, that the algorithm must be distributed, is sufficed to. The algorithm we propose in section 5 is distributed since it contains autonomous systems, i.e. every region.

We succeeded in creating an algorithm that guarantees privacy for every agent. Every agent, who controls a region or a segment of regions, only has information about what happens in the area this agent controls. The agent above can ask him for information about distances, but will never know what happens in the region and will always need the direct controlling agent to gather information.

The algorithm we propose uses the same algorithm for each controller in every level, every controller determines the shortest path in the region below, once this path is determined the controller in the level below does the same thing, using the information the controller above gathered in his search. This means the algorithm is scalable.

The algorithm is dynamic in the sense that it can use real time data for weather and traffic density. Changing of the path when something occurs, certain tracks become longer, density increases or weather worsens still has to be implemented.

In further work, we will study the dynamic part of the algorithm. What happens when the travel time on a track becomes larger? Will the whole route be recalculated or only the region in which this track lies? Another possibility might be to calculate in the first step not only the shortest route at that moment, but as well the best alternative. The difference between the best case of the alternative and the current route could be used as slack to determine whether a delay in the current route is large enough to start recalculating the shortest path.

The algorithm as we propose it in this report does use data for weather and density, but they are randomly chosen, in other words we do not use real

data for these factors. In further research it would be convenient to find a way to gather these real data directly from for example the internet.

In the algorithm we did not use priority queues for finding the shortest distance from the start, in the part where we use *Dijkstra's Algorithm*. Using priority queues could reduce running time as we work with a sparse graph. Another recommendation to reduce running time, although complexity is not a main concern at this moment, is to let the same algorithm be implemented by someone with more experience in programming. Also the same problem should be studied by other researchers to be able to compare different approaches for this problem.

References

1. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
2. A. Giret and V. Botti. Holons and Agents. *Journal of Intelligent Manufacturing*, 15:645–659, 2004.
3. Press Release HMS. *HMS Requirements*. HMS Server, http://hms.ifw.uni-hannover.de/, 1994.
4. A.C. Hoffman. Multiple approaches for distributed routing algorithms. Technical report, 2004.
5. A. Koestler. *The Ghost in the Machine*. Arkana Books, 1971.
6. M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, and A. Lova. Domain-dependent distributed models for railway scheduling. *Knowledge Based Systems. Ed. Elsevier Science*, 20:186–194, 2007.
7. P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. *In proceeding of ESA 2005, LNCS 3669*, pages 568–579, 2005.
8. M. "Wooldridge and P." Ciancarini. "agent-oriented software engineering: The state of the art". Technical report, "Department of Computer Science, University of Liverpool".
9. R. Zelinkovsky. Global automated transport system. *http://www.global-transportation.com*.

APPENDIX

```
Procedure ReadFile
```

```
begin
    Open file to be read
    ArcList=empty set
    While NOT (End of file) do
    begin
        Create arc
        Read data of one line into arc
        Add arc to arcList
        Check if nodes added in arc have a higher number than
            those that are already added
    end
    Close File
    Create matrices for Basetime,Novelbasetime,Inclination,
        Weather,Density,Realtime
    for all arcs in arcList do
    begin
        Save data from arc in matrices Basetime,Novelbasetime,
            Inclination,Weather,Density
    end
end
```

Fig. 19: Procedure ReadFile

```
Procedure SetTotalTime
```

```
begin
    Get basetime(origin,destination) from matrix
    Get novelbasetime(origin,destination) from matrix
    Calculate deterioration to basetime for every factor
    totaltime=sum of basetime or novelbasetime and
        deterioration for every factor
end
```

Fig. 20: Procedure SetTotalTime

```
Procedure CalcRealTime
```

```
begin
    for all nodes do
    begin
        for all nodes do
        begin
            origin=i
            destination=j
            SetTotalTime(i,j)
            Realtime[i,j]=totaltime
        end
    end
end
```

Fig. 21: Procedure CalcRealTime

```
Procedure ShortestPath
```

```
begin
    for all nodes do
    begin
        if node<>beginnode then
        begin
            distance=infinitive
            predecessor=-1
            visited=false
        end
        else
        begin
            distance=0
            predecessor=-1
            visited=false
        end
    end
    while Destination is not visited and
        number of iterations<= number of nodes
    begin
        from unvisited nodes select node with smallest distance
        visited(selected node)=true
        for all unvisited neighbor nodes of selected node do
        begin
            if distance(node)>distance(selected node)+realtime(nodeselected,node) do
            begin
                distance(node)=distance(selected node)+realtime(nodeselected,node)
                predecessor(node)=selected node
            end
        end
    end
end
```

Fig. 22: Procedure ShortestPath

```
Procedure DetermineAgentZeroPath
```

```
begin
    Open file data level 0
    While not (End of file) do
    begin
        Create Arc
        Read data of one line into arc
        Add arc to arcList
        Check if nodes added in arc have a higher number than
            those that are already added
    end
    Close File
    for all arcs in arcList
    begin
        save distance from node to node in matrix agentzero
    end
    Start=begin region
    Destination= end region
    ShortestPath
    j=number of regions in the shortest path
    Set the length of array finalpathregions to j
    Set last value in array to end Region
    node=predecessor of end region
    for i=j-2 downto 0 do
    begin
        finalpathregions(i)=node
        node=predecessor of node
    end
end
```

Fig. 23: Procedure DetermineAgentZeroPath

```
Procedure DetermineLevelOnePath
```

```
begin
    if predecessor end region<>-1 then begin
        postregion=end region
        inregion=predecessor of postregion
        preregion=predecessor of inregion
        File to read=data from inregion
        ReadFile
        CalcRealTime
        for all nodes do begin
            if node is frontier node between inregion and postregion then begin
                Start=node
                Destination=end node
                ShortestPath
                if destination can be reached then save distance from start
                    to destination in cube lengthfront
            end
        end
        while inregion<>begin region do begin
            File to read=data from inregion
            ReadFile
            CalcRealTime
            for all nodes i do begin
                if node i is frontier node between preregion and inregion then begin
                    for all nodes j do begin
                        if node j is frontier node between inregion and postregion then begin
                            Start=i
                            Destination=j
                            ShortestPath
                            if a path to destination exists then save distance from start
                                to destination in cube lengthfront
                        end
                    end
                end
            end
            postregion=inregion
            inregion=preregion
            preregion=predecessor of preregion
        end
        File to read=data from inregion
        ReadFile
        CalcRealTime
        for all nodes i do begin
            if node i is frontier node between inregion and postregion then begin
                Start=begin node
                Destination= i
                ShortestPath
                if a path to destination exists then save distance from start to
                    destination in cube lengthfront
            end
        end
        if preregion=postregion then begin
            Start=begin node
            Destination=end node
            ShortestPath
            if path to destination exists then save distance from start to
                destination in cube lengthfront
        end
    end
end
```

Fig. 24: Procedure DetermineLevelOnePath

```
Procedure MakeFinalPath Part 1
```

```
begin
  for all regions do begin
    for all nodes do begin
        set for begin node and begin region distance to 0 and predecessor to begin node
        set for all other combinations distance to infinity and predecessor to -1
        set for all combinations visited to false
    end
  end
  node selected and region selected=begin node and begin region
  while visited(end region,end node)=false do begin
    select the region-node combination with smallest distance
  end
  visited(selected region,selected node)=true
  if selected region is not first of path then
    visited(predecessor(selected region),selected node)=true
  for all nodes i do begin
    if visited(region selected,i)=false and a path in region selected exists from node
      selected to i then begin
        if distance(region selected,i)>distance(region selected,node selected)
          +lengthfront(region selected,node selected,i) then begin
            distance(region selected,i)=distance(region selected,node selected)
              +lengthfront(region selected,node selected,i)
            if region selected<>end region then begin
                set distance(next region,i) to distance(region selected,i)
                set predecessor(next region,i) to node selected
            end
            else if region selected=end region then begin
                predecessor(region selected,node)=node selected
            end
        end
        else if begin node is selected and is frontier node for begin region and next
          region and i=begin node then begin
            predecessor(next region,node selected)=node selected
            distance(next region,node)=distance(region selected,node selected)
              +lengthfront(region selected,node selected,i)
        end
    end
  end
  length of array finalpathfrontiers=length finalpathregions+1
  last value in finalpathfrontiers=end node
  k=last value in finalpathregions
  node=predecessor(end region,end node)
  if end node is frontier node between end region and its predecessor and end node is
   used as frontier then begin
    penultimate value in finalpathfrontiers=end node
    for the left places i in array finalpathfrontiers from last to first do begin
        finalpathfrontiers(i)=node
        region=predecessor of region
        node=predecessor(region,node)
    end
  end
  else begin
    for the left places i in array finalpathfrontiers do begin
        finalpathfrontiers(i)=node
        node=predecessor(region,node)
        region=predecessor of region
    end
  end
end
```

Fig. 25: The first part of procedure MakeFinalPath

```
Procedure MakeFinalPath Part 2
```

```
begin
    j=0
    for all regions i in finalpathregions do begin
        empty string finalpathregion
        Start=finalpathfrontiers(j)
        Destination=finalpathfrontiers(j+1)
        j=j+1;
        file to read=data from region i
        ReadFile
        CalcRealTime
        ShortestPath
        fill the string finalpathregion with the destination
        node=predecessor(destination)
        while node<>start and node<>-1 do begin
            add node to the front of string finalpathregion
            node=predecessor(node)
        end
        if node=-1 then begin
            Show in interface that no possible route exists
        end
        add R+regionnumber and start to the front of string finalpathregion
        show finalpathregion in interface
    end
end
```

Fig. 26: The second part of procedure MakeFinalPath