Route Search Algorithm in Timetable Graphs and Extension for Block Agents

Ion Cozac

"Petru Maior" University of Tirgu-Mures
cozac@upm.ro

Abstract. This paper describes an algorithm that determines routes using three graphs: the railway graph, the train timetable graph and the summary timetable graph. The search in the timetable graphs is guided by a subgraph of the railway graph, which is defined by the nodes that form an ellipse around the minimum distance path from departure to arrival. We also present some performance evaluations of our proposed algorithm. Finally we describe an extension of this algorithm that can be used in conjunction with block agents to find routes in large timetable graphs, and some applications for medical domain.

Keywords: timetable graph, maximum allowed distance, bridges in graph, block agents.

1 Introduction

The problem of determining optimum path in timetable graphs has been intensively studied during the last ten years, both from theoretical and practical point of view. This problem is important especially if we want to plan trips using the public transportation system (trains, buses, airplanes etc). All the approaches are based on supplementary data structures which are used to speedup the route search. The speedup is important because the server must be able to solve numerous queries simultaneously.

One approach to speedup the search uses multi-level graphs [9]. Precomputed shortest paths are replaced by single edges whose weights are the cost of the corresponding paths. The paper introduces the concept of multi-level decomposition.

Another approach models the timetable information using some heuristics to speedup the implementation [6]. The authors exhibit important extensions of the time-dependent approach to model the earliest arrival and minimum number of transfer problems.

An overview of known models and efficient algorithms for optimal solving the timetable information problem has been given [5]. A comparison between time-dependent and time-expanded approaches has been made in order to evaluate their relative performance [7].

Some authors considered the planning route problem using timetable information, taking into account the presence of delays [4].

The SHARC approach [1] uses contraction, that is, iteratively removal of non-important nodes, plus edges addition to preserve correct costs between remaining

R. Kountchev & B. Iantovics (Eds.): *Adv. in Intell. Anal. of Med. Data & Decis.*, SCI 473, pp. 189–201.
DOI: 10.1007/978-3-319-00029-9_17

nodes. It is a fast unidirectional algorithm, which is advantageous for timetable graphs where bidirectional search is not allowed.

New reviews of all known algorithms in this field have been made [2], and the algorithms have been evaluated using large, real data sets that are publicly available.

This paper is structured as follows. Section 2 introduces some basic notions which will be used throughout the paper: railway graph, bridges and blocks in graph, timetable graph. Section 3 describes in detail our contribution to solve the route search problem in timetable graphs. Section 4 presents some experiments based on our approach. Section 5 shows how our approach can be extended to distribute the route search problem in large timetable graphs, using block agents. Section 6 presents some possible applications of our algorithms in medical domain. The last section is reserved for conclusions and discussions about future work.

Our approach is based on three graphs: the railway graph, the original timetable graph and the summary timetable graph. First, we delimit the search space in the railway graph, that is, which are the stations that can be taken into account when searching routes from departure to arrival. Second, we use the summary graph to identify which trains are valid for our planned route. The original timetable graph is used only in some few special cases, if departure and / or arrival station is not in the summary graph.

2 Preliminaries

Railway graphs. A railway graph $G = (V, E)$ is symmetric and weighted: each edge (v,w) has a cost $c(v,w)>0$. We assume the graph G is connected, that is, there is a path from any vertex v to any other vertex w. The cost of a path $P = \langle v_1,\ldots,v_k\rangle$ is the sum of the cost of all its edges:

$$c(P) = \sum_{i=2}^{k} c(v_{i-1}, v_i) \tag{1}$$

$P^* = \langle s,\ldots,t\rangle$ is a shortest path from s to t if there is no other path P' such that $c(P^*) > c(P')$.

Bridges in Graphs. A *bridge* is an edge whose deletion will disconnect the graph [10]. A *block* is a maximal subgraph that doesn't contain bridges, that is, every new edge we can add to this subgraph is a bridge.

Timetable Graphs. Two basic models for timetable information have been used throughout the above mentioned papers. The time-dependent model uses one node for each station, and every connection by train between two neighbor stations has a corresponding arc. The time-expanded model uses one node for each event (departure from / arrival in station), and an arc between two nodes depicts the train route. We used the time-expanded model due to its versatility, especially because transfers are easily managed.

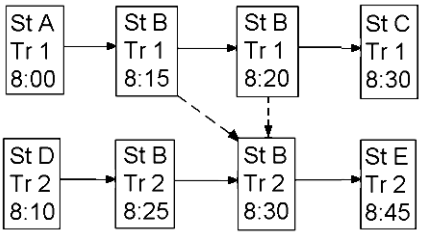

Fig. 1. Partial timetable subgraph, the time-expanded model

Let us examine Figure 1, which depicts the time-expanded model using a partial timetable subgraph. The train T1 starts from station StA at 8:00, arrives in station StB at 8:15, departs at 8:20 and arrives in station StC at 8:30. The train T2 passes through stations StD, StB and StE. If we want to travel from StA to StE, we will take the train T1 from StA to StB, then we will wait for the train T2 to continue the travel.

A *timetable graph* $G_T = (V_T, A_T)$ is directed (not symmetric). Each vertex v_T is a triplet (s,t,h) where s is a railway station, t is a train and h is a time moment. An arc (v_T,w_T) may denote :

- train passing from one station to the next station, if $s(v_T) \neq s(w_T)$ and $t(v_T) = t(w_T)$ (two distinct stations, the same train);
- train halt in a station if $s(v_T) = s(w_T)$ and $t(v_T) = t(w_T)$ (the same station, the same train);
- transfer possibility, if the traveler may gets off in a station to take another train (distinct trains).

The arcs of first two types are marked with continuous links, and the arcs of third type are marked with dashed links. In order to simplify the computations, the field t takes integer values from 0 (for 0:00) to 1439 (for 23:59).

Let us denote by *tim* the cost function which is defined as follows:

$$\text{if } (v_T,w_T) \in A_T \text{ then } tim(v_T,w_T) = (h(w_T) - h(v_T) + 1440) \text{ modulo } 1440 \qquad (2)$$

$tim(v_T,w_T)$ may be the time needed to cover the route from $s(v_T)$ to $s(w_T)$ (different stations), or the waiting time in that station.

There are many solutions to build the timetable graph, if we are talking about how to model the transfers. One solution is to connect each arrival vertex to each departure vertex (Fig 2.a). Another solution is to connect in a ring all the vertices that are related to the same station; the vertices are ordered by time. The last vertex is linked to the first vertex, to allow night transfers [2] [4] [5] [7] [9].

We used a slightly different solution (Fig. 2.b). The arrival vertex is connected to the nearest departure vertex, and the ring contains only departure vertices. This solution allows us to manage easily the transfers, especially if we want to count them, to limit their number, to impose minimum transfer time.

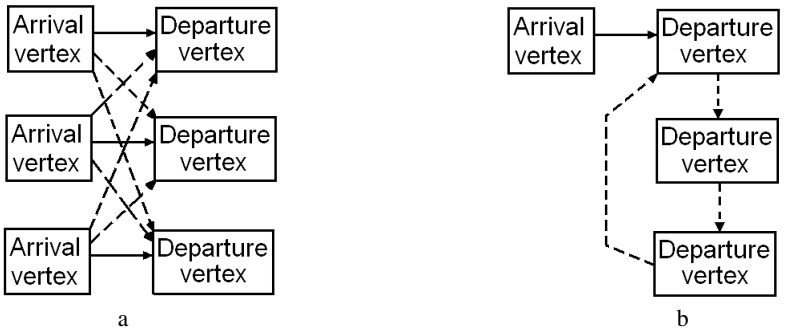

Fig. 2. How to model the transfers: a) from each arrival point to each departure point; b) building a ring with departure points

Suppose k arrival vertices and k departure vertices are related to a particular station. Using the solution depicted in Figure 2.a), the number of auxiliary arcs is k^2. Using the solution depicted in Figure 2.b), the number of auxiliary arcs is k.

Block agents [11] [7]. A *block agent* is a virtual entity that has the following properties:

- autonomy : block agents operate their subproblems without direct intervention of other agents or human;
- social ability : block agents interact with other agents by sending messages to communicate consistent partial states;
- proactivity : block agents perceive their environment and changes in it; they can extend new partial consistent states to more complete consistent states.

3 Route Search Algorithm

3.1 How to Generate the Summary Timetable Graph

Our search algorithm uses the railway graph and two timetable graphs. The first timetable graph depicts only the routes of all trains and nothing else.

If a train starts from station s, or station s has at least three neighbors, then s is set as important. The reason is that the station s may be used to take another train to continue the travel. The second timetable graph depicts a summary of the first timetable, such that every arc links vertices which are related to important stations. This graph includes also rings which are related to important stations.

The built of the summary timetable graph takes about $\boldsymbol{O}(n \log n)$ time, due to some sort and search operations, where $n = |V_T|$.

3.2 Effective Route Search Algorithm

Let dp be the departure station and ar the arrival station. We have to determine as many as possible optimum routes from dp to ar. We should define what an optimum

route is. Let us consider the station *dp* and departure time *h*; we have to determine a route in such a way that we arrive in station *ar* the earliest possible moment. In other words, we solve the earliest arrival problem [2, 4, 5, 6, 7, 9].

The summary timetable graph is used to accelerate the search process, by examining only vertices that are related to important stations. What should we do if *dp*, *ar* or both are not important?

If *dp* is not an important station, we start the search from the departure vertex using the original timetable graph. The start vertex is related to the station *dp* and its time is *h*. The search stops when the first important station is reached. If the next vertex exists, it is of departure type and is related to the same important station. If the search reaches the arrival station, the algorithm returns the path to this vertex. If the search reaches the end of the train route, the algorithm must search for a departure vertex which is related to the end station.

If *dp* is not an important station, the first phase of the algorithm may either return the found path or a departure vertex which is related to an important station.

The destination station *ar* may also be important or unimportant. No matter the type of *ar*, we assign to each train a pointer to the first arrival vertex which is related to *ar*, if such vertex exists in the train route.

Now we have a departure vertex which is related to an important station, so we can use only the summary timetable graph. A variant of Dijkstra's algorithm [3] is used to determine optimum routes in this graph. Every time the search algorithm detects a new train, it checks if this train passes through the arrival station. If this is the case we may insert the arrival vertex in the priority queue and continue the search, or we may simply return the path to it.

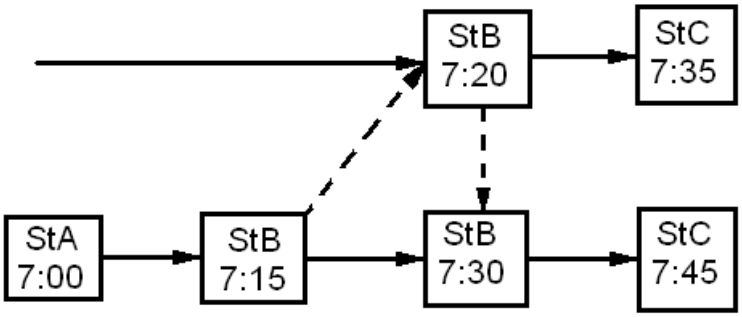

Fig. 3. Optimum versus non optimum path from StA to StC

Let us examine Figure 3 to analyze a particular case. When the algorithm starts, the arrival vertex (time 7:45) is detected on the route of the start train. If this vertex is inserted in the priority queue and the algorithm continues, another arrival vertex will be reached, which gives the best arrival time (7:35). The direct route is not optimum, but it doesn't need any transfer.

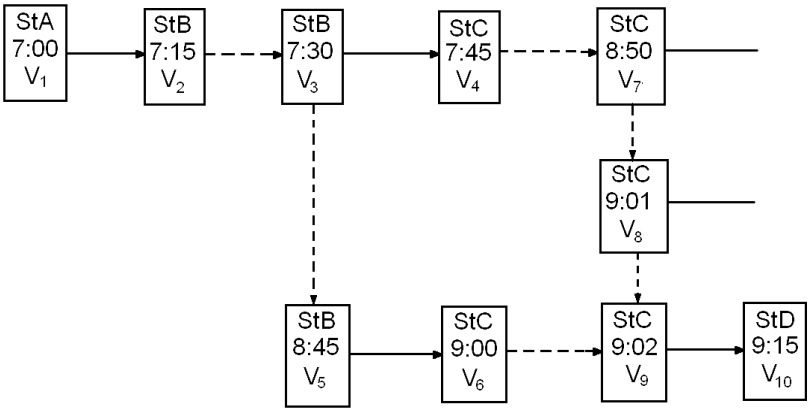

Fig. 4. Determining optimum route from StA to StD with imposed transfer time

Important Remark. Using rings to connect departure vertices may cause troubles when we are searching for routes with imposed minimum transfer time. Let us examine the figure 4, which depicts a critical case in station StC, assuming the waiting time is imposed to be at least 3 minutes. According to the algorithm of Dijkstra, the vertices are extracted from the priority queue in this order: v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, etc. When v_6 is extracted, the successor v_9 is entered in the priority queue with its settings (cost of the route from departure etc). The waiting time is 2 minutes, which is not acceptable. So we have to find another predecessor for v_9, if such predecessor exists. In this case, a better predecessor is v_4, which has been extracted before. If such predecessor doesn't exist, we have to consider the next departure vertex which will replace the vertex v_9 in the current path.

We need to use a supplementary list of vertices, which is assigned to each important station we traverse. This list contains all the arrival vertices which are detected by the search algorithm. When we traverse the ring, this list must also be scanned to link a departure vertex from the best previous arrival vertex, such that the imposed conditions are fulfilled.

This problem does not appear if we are using the method which has been depicted in Fig. 2.a). But that method has another disadvantage: it consumes too much memory and is slower.

3.3 How to Improve the Search Algorithm

This algorithm may have a large search space if stations *dp* and *ar* are far away from each other and there is no direct train to connect them, so the response time may not be acceptable. We know that, using the algorithm of Dijkstra, the search will span a disk around the departure vertex until the destination is reached.

The difficulty of search process is considered using our experimental data. A route is difficult if the search algorithm spans more than quarter the number of arcs.

In order to reduce the search space, we used the following idea (see Fig. 5).

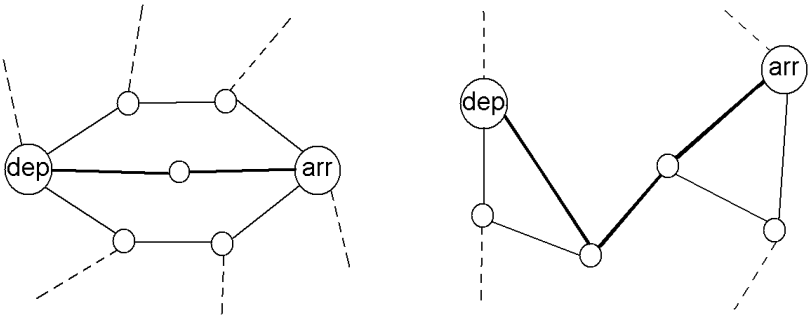

Fig. 5. Determining an ellipse around the minimum cost path

We determine a minimum cost path from *dp* to *ar* in the railway graph. We also determine a subgraph which includes the found path and each vertex z fulfills the following condition:

$$dist(dp,z) + dist(z,ar) \leq f \cdot dist(dp,ar) \quad (3)$$

where $dist(x,y)$ is the cost of the shortest path from x to y (the cost is computed over the railway graph), and $f > 1$ is a convenient chosen parameter (see Experiments). This subgraph will define an ellipse around the minimum cost path and will be used as search space. The value $f \cdot dist(dp,ar)$ is the ***maximum allowed distance*** for any route we are searching.

The search space may have some fragments that will never be used by any route from *dp* to *ar* (see dashed lines in Fig. 5). These fragments are identified using bridges and blocks in the ellipse subgraph. First, we determine the blocks that include the arcs of the shortest path; we may have one block (left) or many blocks (two blocks and one bridge, right). The arcs that don't belong to any of these blocks are dropped.

The usage of this method is motivated by the fact that the railway graph is much lower than the timetable graph (see Experiments). The search space is reduced once, before starting any route search, and every route search will span a disk slice.

4 Experiments

We implemented our algorithms in ANSI C and compiled them using GNU C Compiler version 4.1.0 on an AMD Athlon processor at 1.4 GHz with 2 GB of RAM running Linux Red-Hat 4.1.0-3.

We dealt with the whole Romanian timetable, valid for year 2011, which includes international trains that pass through Romania. The railway graph has 2250 vertices (250 vertices correspond to important stations) and 5150 arcs, the timetable graph has 58300 vertices and 56150 arcs, and the auxiliary graph has 20850 vertices and 33450 arcs. All these data structures need about 1.8 MB to be stored.

The summary timetable graph is generated in about one second. If the routes or the timetables of some particular trains are changed, the original timetable graph is

updated and the summary timetable graph is rebuilt. Whenever the timetable is changed, the railway graph is updated such that it keeps only those links which are traversed by trains.

We queried for the following groups of routes:

- from Bucuresti-Nord to Tirgu-Mures and return;
- from Bucuresti-Nord to Botosani and return;
- from Suceava to Tirgu-Mures and return;
- random departure and arrival.

We chose routes from and to Bucuresti-Nord because it is the most important Romanian station, and has assigned the maximum number of departures and arrivals. We also chose random routes to estimate the general performance of our proposed algorithm.

The parameter *f,* which is used to determine an ellipse around the minimum distance path (to limit the search space), has been set to 1.60. This value has been determined by examining some sinuous routes between Romanian stations. One such route is between Suceava and Tirgu-Mures, which gives three distinct rail paths having distances 365 km, 450 km and 532 km respectively. The ratio between the longest and the shortest path is 1.45. The value of *f* has been increased for safety reasons.

Our railway graph is denser inside Romania than outside. The reason to consider only Romanian stations (the dense part) is that, given a particular pair (*dp*,*ar*) whose shortest distance is *d*, the following condition is fulfilled:

$$(df+e)/(d+e) < f, \text{ for any supplementary distance } e>0 \tag{4}$$

So this value of *f* covers well almost all routes. It is possible, even though very unlikely, that some routes may not be optimal. That is, there may be another route whose distance is greater than the found one, but the travel time is lower. To prevent such cases, an exhaustive method could be used to check routes between any two important stations. This method will imply 62500 queries, and assuming 0.4 seconds on average for one query, this action could take about 7 hours. We may also determine different values for distinct departure stations, to obtain better space reduction. This solution should be improved to eliminate unnecessary queries, for example considering the railway topology.

The first column of Table 1 below shows: departure station, arrival station, minimum and maximum distance for found routes, number of departure vertices. The second column shows: total response time, averaged response time. The total response time is measured from the point of query reception to the point of answer completing. The averaged response time is computed as ratio between total response time and number of departure vertices.

The third column shows the total number of spanned arcs for two cases: with and without space reduction, the unproductive searches being included (see remark below). All these tests used the same triplet: railway graph, original timetable graph, summary timetable graph. The number of spanned arcs depends heavily on the topology of the railway net, which is not uniformly dense.

Table 1. Performance evaluation (samples)

Routes	Response time	Spanned arcs
Bucuresti-Nord -- Tirgu-Mures distances : 449 km / 612 km 109 departure vertices	total : 0.356 seconds per dep vtx : 0.0033 sec	48000 / 185000
Tirgu-Mures -- Bucuresti-Nord distances : 449 km / 612 km 19 departure vertices	total : 0.035 seconds per dep vtx : 0.0018 sec	3200 / 8000
Bucuresti-Nord -- Botosani distance : 477 km 109 departure vertices	total : 0.181 seconds per dep vtx : 0.0017 sec	24000 / 280000
Botosani -- Bucuresti-Nord distance : 477 km 8 departure vertices	total : 0.018 sec per dep vtx : 0.0023 sec	1500 / 2500
Suceava – Tirgu-Mures distances : 365 km / 532 km 61 departure vertices	total : 0.078 sec per dep vtx : 0.0013 sec	16500 / 96000
Tirgu-Mures -- Suceava distances : 365 km / 532 km 19 departure vertices	total : 0.032 sec per dep vtx : 0.0017 sec	2400 / 12000

In order to estimate the general performance of our proposed algorithm, we generated 10000 random queries. The estimation gives the following results: 400 seconds to answer all 10000 queries for 125000 total departure vertices. The averaged response time is 0.04 seconds per query, and 0.0032 seconds per departure vertex.

Generally speaking, the total response time depends on the number of departure vertices, the number of necessary transfers and the size of the ellipse graph which is built using the maximum allowed distance criterion.

It is important to say that the end user is interested about the total response time: he receives several variants of which he can choose. It is also important for us to know the averaged response time (per departure vertex) to evaluate the efficiency of the algorithm.

Remark. In order to find routes from departure to arrival, the application starts a new search from each departure vertex. Some of these vertices are not productive, that is, they will not give any route to destination. But the time consumed with these unproductive nodes is taken into account to measure the response time (total and averaged).

If we don't use any speedup technique (that is, the original timetable graph is extended with supplementary arcs to solve the transfers, so the summary timetable graph is not used), the total response time may be up to 15 times slower. The coefficient is lower if direct trains connect departure and arrival, and higher if three or more trains are needed to cover the route.

The auxiliary graph has been built using rings to connect departure vertices which are assigned to the same important station (see Fig. 2.b). This solution uses low

volume of memory to store this data structure. Another possible technique could be to connect each arrival vertex to each departure vertex (see Fig. 2.a), if the related station is important. Using this technique, the auxiliary graph could have 110000 arcs, and the response time may be three times slower. These estimations are valid for our timetable. If the timetable is higher, the number of arcs and the response time will be higher; the increase factor is worse than linear.

5 Extension for Block Agents

The above solution may be successfully used for medium timetable graphs, for example, if the timetable stores information about a particular country, including its international trains. This solution may not be well suited for continental timetables, due to the following reasons:

- searching for routes traversing two or more countries may take long time to be completed, if three or more trains are needed to cover the route;
- updating the timetable is a time consuming task.

To surpass these difficulties, we propose a solution based on block agents. We didn't find any paper that tackles the subject of finding routes in timetable graphs using block agents. The only paper that tackles a close related subject is, to our knowledge, the work of Salido et al [8], which discusses about how to design a new timetable using constraint satisfactions.

Now let us depict our proposed solution. Every country uses its own graphs (one railway graph and two timetable graphs), which are managed by its own block agent. The timetable graphs include the trains that traverse this country, including the international ones. If both departure and arrival are known, the above solution may be successfully used without any auxiliary tools.

To solve the general problem, we introduce two new block agents. The first agent (let us denote it by CRA - central railway agent) manages the continental railway graph. The second agent (denoted as CTA - central timetable agent) manages a subgraph of the continental timetable graph which is built as following. Each agent who is related to a particular country identifies al the trains that are needed to cover routes between any two distinct frontier stations. The timetable of these trains is sent to CTA, which will build the needed timetable subgraph. The agent CRA receives also the list of stations that are managed by CTA. The central agent CTA will manage the summary timetable graph only, which includes the important stations of the continent.

Now we are able to find routes between any two stations, which are located in different countries. The agent CRA receives a query to identify the ellipse around the minimum distance path and identifies the countries that are covered by this ellipse. It also identifies some important intermediate stations: one set of stations for the departure country and another set of stations for the arrival country (Fig. 6). Three queries are sent to three different agents.

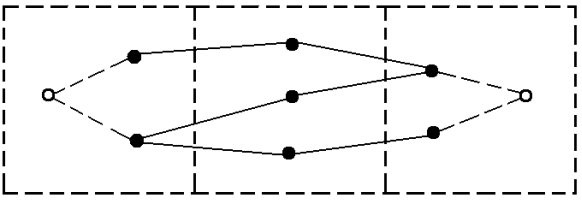

Fig. 6. Determining a route that traverses more than two countries and needs cooperation among agents

The first query is addressed to the agent who has knowledge about the departure station. This agent will determine the first list of routes: if the departure station is not in the list of the agent CTA, it determines trains from departure to the nearest stations that may be managed by CTA (see dashed links). If the departure station is managed by CTA, the first query is not needed.

The second query is addressed to the agent CTA, which will determine the list of routes between departure country and arrival country (see continuous links). This step is always necessary if the departure country is not able to determine the route.

The third query is addressed to the agent who has knowledge about the arrival station. This agent will determine the last list of routes: if the arrival station is not in the map of CTA, it determines trains from the stations that have been set by CRA to the arrival station (see dashed links). If the arrival station is managed by CTA, the third query is not needed.

If the departure and/or arrival station is not in the CTA map, we need to build routes from arrival to departure by mixing the partial routes.

6 Application for Medical Domain

The first and obvious application of the algorithms presented above is for touristic domain, if one is interested to plan its route using the public transportation system. We can also use these algorithms in the medical system, for example to optimize the traffic of the emergency vehicles. A customized application may be designed to watch over this traffic, if the vehicles are endowed with GPS devices.

Using the public transportation system may have a positive impact concerning the public health. If people don't use their own cars, the traffic will be decongested, the pollution will be reduced and the emergency traffic will be improved.

7 Conclusions and Future Work

This paper presented a simple, less memory consuming and fast solution that can be used to solve the problem of searching routes in timetable graphs. We adopted this solution in the hypothesis that short / direct routes are much more frequent than long routes with two or more transfers (difficult routes). Our approach requires the least memory consumption and the shortest time needed to generate the summary timetable graph. Other methods require about several minutes or hours to generate the auxiliary data structures.

We have to search for better solutions to solve the update problem, to eliminate the rebuilt of the summary timetable graph. Removing an existing train or inserting a new train should involve some operations to update the links between different trains in all important related stations. Updating the timetable graph of an existing train may involve a new order of vertices in the related rings. Improved solutions are necessary: if the original timetable graph will have 1000000 vertices, the built of the summary timetable graph will take about 24 seconds - the time complexity is $\boldsymbol{O}(n \log n)$.

In order to reduce the search space, we determine an ellipse around the shortest distance path, which is defined using the maximum allowed distance criterion, based on the parameter *f*. Its value has been determined using an empirical method. We may determine better (possible multiple) values for *f*, to obtain better space reduction. We may also search for other solutions to reduce the search space, maybe with the cost of increasing the preprocessing time and / or the memory consumption for supplementary data structures.

We also described a theoretical solution to solve the large timetable case, but this solution is not yet implemented. The main difficulty consists in solving some special cases, if some trains have running restrictions. Indeed, it is possible that some intermediate trains or some final trains don't run daily. The queries may be solved sequentially or simultaneously. In the sequential approach, the agent located in the departure country will solve the departure subproblem. CTA will solve the intermediate subproblem using the output of the first train. The last agent will solve the arrival problem using the output of CTA.

Another approach could be to solve simultaneously these three subproblems. We have to obtain multiple solution sets, each set being valid for a certain day. This is because we can not know in advance the arrival day in an intermediate station following a certain route. The arrival day is known only after the search algorithm reaches the destination station.

Using separate agents for different countries will be an advantageous solution, because this way every local agent will almost always manage local queries, that is, departure and arrival are in the same country. Of course, local agents are also able to solve queries for routes between neighbor countries. Anyway, we can assume that local queries are much more frequent than international ones, so this is a good reason to continue this work using block agents.

Acknowledgement. This work was supported by the Bilateral Cooperation Research Project between Romania and Bulgaria, entitled: "Electronic Health Records for the Next Generation Medical Decision Support in Romanian and Bulgarian National Healthcare System", the involved institutions are the Technical University of Sofia and Petru Maior University of Tirgu Mures.

References

1. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: Proceedings of 10th Workshop of Algorithms and Engineering Experiments (ALENEX 2008), pp. 13–26. SIAM (April 2008)

2. Bauer, R., Delling, D., Wagner, D.: Experimental Study of Speed-up Techniques for Timetable Information Systems. Journal Networks 57, 38–52 (2011)
3. Dijkstra, E.W.: A Note on Two Problems Ion Connection with Graphs. Numerische Mathematic 1, 269–271 (1959)
4. Frede, L., Müller-Hannemann, M., Schnee, M.: Efficient On-trip Timetable Information in the Presence of Delays. In: ATMOS 2008, 8th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization and Systems, pp. 1–16 (2008)
5. http://drops.dagstuhl.de/opus/volltexte/2008/1584
6. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.D.: Timetable Information: Models and Algorithms. In: Geraets, F., Kroon, L.G., Schoebel, A., Wagner, D., Zaroliagis, C.D. (eds.) Railway Optimization 2004. LNCS, vol. 4359, pp. 67–90. Springer, Heidelberg (2007)
7. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Toward Realistic Modeling of Timetable Information Through the Time-dependent Approach. In: Proc. of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2003). ENTCS, vol. 92, pp. 85–103. Elsevier (2004)
8. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. ACM Journal of Experimental Algorithms 12, Article 2.4 (2007)
9. Salido, M.A., Abril, M., Barber, F., Ingolotti, L., Tormos, P., Lova, A.: Domain-Dependent Distributed Models for Railway Scheduling. Journal Knowledge Based Systems (Elsevier) 20, 186–194 (2007)
10. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using Multi-level Graphs for Timetable Information in Railway Systems. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 43–59. Springer, Heidelberg (2002)
11. Tarjan, R.E.: A Note on Finding the Bridges of a Graph. Information Processing Letters 2(6), 160–161 (1974)
12. Wooldridge, M., Jennings, R.: Agent Theories, Architectures, and Languages: A Survey. Intelligent Agents, 1–22 (1995)