Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

THE SHORTEST PATH: COMPARISON OF DIFFERENT APPROACHES AND IMPLEMENTATIONS FOR THE AUTOMATIC ROUTING OF VEHICLES

Kai Gutenschwager

Hochschule Ulm
Prittwitzstraße 10
D-89075 Ulm, GERMANY

Axel Radtke

SimPlan Integrations GmbH
Friedrich-Ebert-Straße 87
D- 58454 Witten, GERMANY

Sven Völker

Hochschule Ulm
Prittwitzstraße 10
D-89075 Ulm, GERMANY

Georg Zeller

SimPlan AG
Benzstraße 15
D-89129 Langenau, GERMANY

ABSTRACT

The routing of vehicles or personnel in complex logistics systems is a task that needs to be solved in numerous applications, e.g., detailed models of transport networks or order picking areas. The number of relevant nodes in such networks can easily exceed 10,000 nodes. Often, a basic task is finding the shortest path from one node (start) to another (destination). Within the last years various simulation tools have been extended by respective algorithms. However, the execution time of the simulation model may significantly depend on the number of nodes in the network. We present algorithms from the literature and a comparison of three simulation tools with respect to execution time and model size for different scenarios. We further present an approach to work with so called sub-networks, i.e., the network is separated into areas, where finding the shortest path includes the task of starting at a node in one sub-network with a destination in another one.

1 INTRODUCTION

There is a variety of applications where simulation models are set up to analyze systems in which the routing of personnel, e.g., workers or order pickers, and vehicles is a relevant part of the modeled process. A basic task in this field is to find the shortest path from a current position to a given destination (within a graph of nodes). For this problem, a number of algorithms is known. Several simulation systems offer an automatic routing of vehicles, such that a vehicle takes the shortest path from the current position (node) to a destination node automatically. Due to implementation details and the used algorithm, the performance of simulation tools for finding a shortest path may differ in some most relevant aspects:

- Maximum size of transport networks (number of nodes) that can be simulated
- Average computational time to fulfill a certain number of transport orders during a simulation run
- Average computational time and resulting model size to initialize the corresponding graph and matrices where shortest paths are stored

We compare three simulation tools, namely *Automod*, *Enterprise Dynamics* and *Plant Simulation* with respect to the implementation and performance of automatically routing vehicles from a current position to a destination on the shortest path within a transport network. Furthermore, we present a new approach for finding shortest paths which combines the advantages of short computational times and an acceptable model size.

2 SHORTEST PATH ALGORITHMS

Transport networks can be modeled as graphs consisting of nodes and edges connecting the nodes. Different algorithms have been proposed for finding the shortest path between the nodes in a graph. All algorithms presented here are based on weighted graphs, i.e., the network consists of a set N of n nodes and a set E of m edges (arcs), each connecting two nodes (i, j). The weight of an edge is denoted as $d(i, j)$ for given direct connections, for all other connections (via other nodes) $d(i, j)$ denotes the shortest distance.

2.1 Basic Approaches from the Literature

Dijkstra's Algorithm This algorithm delivers the shortest path from a given node i to a single destination node or all other nodes within a graph with nonnegative edge path costs (Dijkstra 1959). The idea is to set up a set of unvisited nodes and the tentative distance from node i to all other nodes j, denoted as $Dist[j]$. Furthermore, a list $Prev[j]$ stores the previous node on the path from node i to node j. All distances are initially set to infinity. The start node i is the initial current node. For the current node all of its unvisited neighbors (connected by an edge) are considered next and the tentative distances to these neighbors are calculated. After all neighbor nodes of the current node have been considered, the current node is deleted from the set of unvisited nodes. The next node selected as the current node is the one with the shortest distance to node i (i.e., the node which has the minimum value in $Dist[j]$). The algorithm terminates once the destination node has been deleted from the set of unvisited nodes, or once all nodes have been considered and the set of unvisited nodes is empty. The calculation of the tentative distances is done by checking if the currently stored distance from node i to node j (given by $Dist[j]$) is greater than $Dist[k]$ + $d(k, j)$. In this case, $Dist[j]$ is overwritten with $Dist[k]$ + $d(k, j)$ and $Prev[j]$ is set to node k.

The simplest implementation of Dijkstra's algorithm stores nodes in a linked list or an array, and the operation to find the minimum value in list $Dist$ is a linear search through all nodes in $Dist$. In this case, the time complexity is $O(n^2)$. However, the complexity can be reduced for sparse graphs, i.e., for graphs with far fewer than n^2 edges, with a more intelligent strategy to store the graph in form of adjacency lists and using different types of heaps to implement the operation to find the minimum value in $Dist$ efficiently. With a strategy of using the Fibonacci heap the running time can be reduced to $O(m + n \times log(n))$ (Cormen et al. 2001; Fredman and Tarjan 1987).

The bi-directional approach of Diskstra's algorithm, also called *two-side Dijkstra* is based on the idea that finding the shortest path from start node i to destination node j might be carried out faster, if the search is started from both sides in parallel. As Dijkstra's algorithm can be interpretated as a breadth-first search, a lot of "unnecessary" steps might be carried out before reaching the destination node. Starting from both sides might reduce the computational effort significantly (Berge and Ghouila-Houri 1965). However, if the two-sided procedure is terminated as soon as a node has been processed from both directions, there is no guarantee that this node is actually on the shortest path from the start node i to destination node j (Vahrenkamp and Mattfeld 2007). Note, that the Bellmann-Ford algorithm also computes the shortest path to all nodes from a starting node in a weighted digraph. It is slower than Dijkstra's algorithm (Bellman 1958), but can be used for graphs with negative edge weights, which does not apply for the problems considered in this paper.

A* Algorithm The A* algorithm is a more general approach than Dijkstra's algorithm for finding the shortest path between two nodes in a graph, see (Hart et al. 1968) for a first approach with a correction in (Hart et al. 1972). As stated above, Dijkstra's algorithm performs as a breadth-first search where the next node to be looked at is the one with minimum distance to the starting node (given in $Dist$). The A* algorithm introduces a heuristic to determine the order in which nodes are selected in the search process. This heuristic is a sum of two terms. The first term is the distance to the current node k (given as $Dist[k]$), the second one is an estimation of the distance to the destination node j, usually denoted as $h(k)$. In most implementations $h(k)$ is computed as the Euclidean distance from the considered node to the destination node. If coordinates of all nodes are available, the distance can, e.g., be calculated by Pythagoras' theorem. Dijkstra's algorithm can be viewed as a special case of the A* algorithm where $h(k) = 0$ for all nodes k (Nilsson 1980).

Floyd-Warshall's Algorithm This algorithm was developed independently from each other by Floyd (1962) and Warshall (1962). Instead of computing a path from a given start node to all other nodes (or a single destination node), all shortest paths, i.e., from each node to all others, are computed within a single loop. As a result we obtain a matrix *Dist*, where $Dist[i,j]$ denotes the distance from node i to node j. Furthermore a matrix *Next* can be computed where $Next[i,j]$ represents the successor of node i on the shortest path from node i to node j (see Alg. 1).

```
for all nodes $i$ of $N$ do
    for all nodes $j$ of $N$ do
        if there is an edge from $i$ to $j$ then $Dist[i,j] := d(i,j)$ else $Dist[i,j] := \infty$
for all nodes $i$ of $N$ do
    for all nodes $j$ of $N$ do
        for all nodes $k$ of $N$ do
            if $Dist[j,i] + Dist[i,k] < Dist[j,k]$ then $Dist[j,k] := Dist[j,i] + Dist[i,k]$;   $Next[j,k] := i$;
```

Algorithm 1: Floyd-Warshall's algorithm

Floyd-Warshall's algorithm has a time complexity of $O(n^3)$, which is equivalent to performing Dijkstra's algorithm n times. However, Floyd is usually faster than executing Dijkstra's algorithm for each node. In analogy to techniques to improve the time complexity of Dijkstra's algorithm, *Johnson's algorithm* (Johnson 1977) can be regarded as a superior approach to find the shortest paths between all pairs of nodes in sparse directed graph. It allows some of the edge weights to be negative, but no negative-weight cycles must exist. It works by using the Bellmann-Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.

Highway Hierarchies Several speed-up techniques for shortest-path algorithms have been proposed (e.g., Willhalm 2005 and Sturtevant and Geisberger 2010). The approach of highway hierarchies is of special importance for path-finding in road networks. It bases on the fact, that logistic systems often contain clusters of densely connected stations while the clusters are interrelated by only a few main edges—i.e., there are cities and highways between the cities. A similar structure can be found in plant logistics systems where manufacturing lines and warehouses resemble cities and in-plant roads correlate with highways. Highway hierarchies exploit this characteristic. The original graph is extended by adding a coarsened graph. An edge in the coarsened graph represents a shortest path in the original graph. The original graph and the coarsened graph are linked by additional edges. Several levels of reduced graphs may be introduced. Traditional algorithms like Dijkstra's algorithm can be applied on highway hierarchies. Details on this approach can be found in Knopp et al. (2007) and Schultes (2008).

2.2 Comparison of the Algorithms for the Usage in Simulation Models

A comparison of different approaches for finding shortest paths in road networks can be found in (Zhan and Noon 1998). As we focus on the implementation of shortest path algorithms in simulation models, we can distinguish two cases:

- Shortest paths are calculated during the simulation run (e.g., Dijkstra's algorithm or A*).

- All shortest paths are computed beforehand and are stored in respective matrices (e.g., Floyd-Warshall's algorithm). Once computed, finding the shortest path—or finding the next node to be visited on the path to the destination node, respectively—is done by looking it up in matrix *Next*.

If each shortest path is needed once for any simulation run with the respective model, both strategies should require similar computational times. However, the same shortest paths are often needed multiple times in simulation models of order picking or transportation systems. This may be due to several vehicles within the system taking the same paths. Furthermore, one has to consider the number of replications of simulation runs within an experiment. More important, however, is the fact that paths may only differ in a few nodes at the start or the end of the complete path. Using a pre-calculation of all shortest paths will also prevent from the re-calculation of paths. In this case, the pre-calculation of all shortest paths should lead to a much better computational time in total (cf. Sec. 5 for respective findings).

In case of using Floyd-Warshall's algorithm, the short computational times in model execution have to be paid for with additional storage space consumption. The required space for storing matrix *Next* depends on the squared number of nodes. This may become a restrictive factor even for moderately large graphs (or respective road networks). As can be seen from the results in Sec. 5, common simulation tools whose shortest path algorithms include the generation of adjacency matrices, can only handle problem sizes of up to 6,500 nodes for a regular PC with 6 GB RAM. On the other hand, approaches which are based on Floyd-Warshall's algorithm are much faster in model execution than approaches that compute shortest paths during model execution.

The next section presents a combined approach, where most of the computation is done before model execution and only a rather small ratio of computation has to be done during the simulation run. This approach results in small model sizes and is especially designed for large graphs.

3 A HIERARCHICAL APPROACH USING FLOYD-WARSHALL'S ALGORITHM

In this section we present an approach to combine the advantage of Floyd-Warshall's algorithm with a reasonable model size and RAM requirement for the corresponding matrices. Unlike the more general approach of highway hierarchies (see Sec. 2.1) we add just one hierarchy level. The main idea is, that the overall graph is split into several sub-graphs of reasonable size. If we consider a graph with 150×150 nodes (22,500 nodes) and split it up into 36 sub-graphs of 25×25 nodes (each 650 nodes), we reduce the space requirement for disk space and RAM significantly to about only 4.4% —from $150^4 = 506,250,000$ to $36 \times 25^4 = 14,062,500$ data fields of the needed matrices of the sub-graphs and $2900 \times 2900 = 8,410,000$ data fields of the additional matrix *Main* which stores the connections between the sub-graphs.

Floyd-Warshall's algorithm is performed for each sub-graph. Shortest paths with start node and destination node belonging to the same sub-graph can be handled as described in Sec. 2.1. Paths which start in one sub-graph and have their destination node in another sub-graph need to be calculated during the simulation run. With the definition of sub-graphs, we also can define two sub-sets of the node set of each sub-graph, referred to as exit and entry nodes of a sub-graph: An exit node is defined as a starting node of an edge with the partner node belonging to another sub-graph, which is called an entry node, respectively.

Consider the example given in Fig. 1. The shortest path from a node in sub-graph *A* to a node in sub-graph *F* might lead through sub-graphs *B, C* or *B, E* or *D, E*. In order to find the shortest path, we need to check the distance from the start node to each exit node of the sub-graph and correspondingly the distance from each entry node of the destination sub-graph to the final destination node. We further need to know the shortest distance between all entry and exit nodes of the sub-graphs. Here, an additional graph on a higher hierarchy level is used: We define a graph called *Main*, which consists of all nodes that are either an entry or an exit node (or both) of any of the given sub-graphs. Some of these nodes are connected directly by an edge. For each sub-graph we also know the shortest distances between all entry and all exit nodes (as a result from the application of Floyd-Warshall's algorithm to the respective sub-graph). These

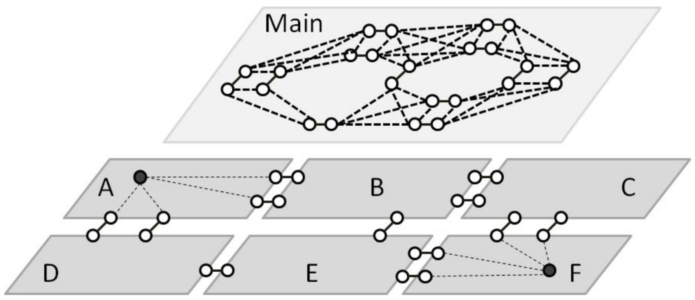

Figure 1: Hierarchy of graphs with six sub-graphs.

shortest distances now constitute further edges within graph *Main* (cf. Fig. 1). For the resulting graph all shortest distances are calculated according to Floyd-Warshall's algorithm again.

In order to obtain the shortest path from node i in sub-graph SG_{start} to a destination node k in sub-graph SG_{dest} we apply Alg. 2. The function $dist(A,i,j)$ returns the shortest distance from node i to node j of graph A. Note, that a simple improvement is to enter the inner loop only if d_{exit} is smaller than d_{min}. As can be seen, the computational time depends on the number of entry and exit nodes of the two respective sub-graphs with a complexity of $O(n \times m)$ with n being the number of exit nodes of SG_{start} and m being the number of entry nodes of SG_{dest}.

```
$d_{min} := \infty$
for all exit nodes $j$ of $SG_{start}$ do
    $d_{exit} = dist(SG_{start}, i, j)$
    for all entry nodes $l$ of $SG_{dest}$ do
        $d_{main} := dist(Main, j, l)$;   $d_{entry} := dist(SG_{dest}, l, k)$;   $d_{total} := d_{exit} + d_{main} + d_{entry}$
        if $d_{total} < d_{min}$ then $d_{min} := d_{total}$;   $j_{best} := j$;   $l_{best} := l$
```

Algorithm 2: Obtaining the shortest path for nodes within different sub-graphs.

With j_{best} and l_{best} we can obtain the sequence of all other entry and exit nodes on the shortest path directly from the successor matrix *Main*. We set up a list of these nodes from *Main*. This way, the given algorithm has to be performed only once for each transport order. The vehicle asks at each node for the next node to visit. If the next node on the list is a direct neighbor of the current node, we can delete the node from the list. Otherwise, the next node is retrieved from the corresponding successor matrix of the current sub-graph; with the next node on the list as the destination node. Once the vehicle has reached the destination sub-graph the original destination node is used to find the next node to be visited from the successor matrix of the destination sub-graph.

4 IMPLEMENTATION OF AUTOMATIC ROUTING IN SIMULATION TOOLS

There are various software tools available in the field of discrete event simulation (see Lindemann and Schmid (2007) for a market survey). We have inspected three simulation systems widely used in the domain of production and logistics: *Tecnomatix Plant Simulation* from Siemens, *AutoMod* from Applied Materials, and *Enterprise Dynamics* from INCONTROL Simulation Solutions. Although the algorithms implemented in these tools are not published by the software vendors, some conclusions can be drawn from the modeling requirements. In this section we discuss how to model a transport network and a vehicle that has to fulfill a sequence of transport orders defined by their respective destination nodes.

4.1 Plant Simulation

Plant Simulation provides uni- and bidirectional tracks to model road networks (Bangsow 2010). Each track has a starting point and an endpoint, which might be connected with starting points or endpoints of other track objects. An event handler may be called whenever a vehicle reaches the starting point or endpoint of a track. Using this modeling concept, arbitrary complex road junctions can be modeled.

Plant Simulation is obviously using Dijkstra's, A* or a derivative of these algorithms for determining the path of a vehicle from its current location to a given destination: There is no preprocessing of the road network required before performing a simulation run. It is even possible to modify the network during a simulation run: Vehicles will always use the shortest path to reach their destination.

For modeling the different graphs we used bi-directional tracks. For modeling the nodes we created an object consisting of a short bi-directional track and a method to check whether the node is the current destination of the vehicle. If so, the current transport order is fulfilled and the destination node of the next transport order is taken from the order list, i.e., the respective attribute is overwritten. The entire layout including all connections of tracks and stations is generated by respective methods.

We also used Plant Simulation as a basis for implementing the strategy presented in Sec. 3. Note however, that this approach is independent from any of the simulation tools we consider in this paper.

4.2 AutoMod

AutoMod is a compiler based 3D simulation tool, that provides a subsystem type called path mover for modeling vehicle systems. The path network distinguishes paths and control points. Paths have to be drawn using different primitives (straight sections, curves etc.) and control points for interaction have to be snapped on them. Each control point can only be assigned to one path. A network can also be imported from a text file with a little help of a spreadsheet. In this case, the uncompiled model's size exceeds the network's text file only by a few kilobyte.

In AutoMod, algorithms are used to calculate fastest routes between points, regarding user defined transfer-times and navigation-factors on specific paths. During the compiler run, AutoMod builds up a routing table with route information for every possible combination of control points. This leads to large compiled models. Building up this routing table cannot be disabled. Hence, the model size is limited.

The vehicle's behavior is simulated very accurately by regarding acceleration, curve speed, speed limits depending on load types and other physical details. Vehicle queuing, strategies for assigning jobs to vehicles and other features can be used, too. During a simulation run, vehicles can be rescheduled by user defined methods but the network itself cannot be modified.

In our simulation experiments, the vehicle reads the given list of transport orders (destination nodes) from an external text file and begins traveling on the bi-directional paths. AutoMod is able to perform simulation runs with 1,000 vehicles and more at one time in the 41×41 network (see Sec. 5).

4.3 Enterprise Dynamics

Enterprise Dynamics offers the creation of node networks, to which vehicles can be linked. Each node has a position given by its coordinates. All bi-directional edges can be created using an adjacency matrix of all nodes, where a value of one denotes a direct connection between the respective nodes. The length of the edges are computed by Enterprise Dynamics.

After generating the network, an optimization routine needs to be carried out before the first run. The shortest distances between all nodes in a network are computed by Enterprise Dynamics in advance using an optimized version of Dijkstra's algorithm.

For modeling the different graphs we used the node network. A source and a destination object for the vehicle (transporter) under consideration are used to communicate the next two destinations to visit. As vehicles need a transport order as a basis, we always create a new order with starting point being the next node on the list and the destination node being the one following that node on the list. Once the vehicle reaches its destination, the source and destination objects are dynamically attached to the corresponding next nodes (according to the order list). The entire layout is generated by respective methods generically.

5 EMPIRICAL RESULTS

5.1 Considered Problem Instances

In order to analyze the performance of different algorithms or implementations, instances of road networks (graphs) of different size and complexity are needed. We have generated road networks (graphs) where each station (node) $(u;v)$ with $u,v = 1,\dots,n$ is located at coordinates $(x;y) = (u\cdot g + rnd(g); v\cdot d + rnd(g))$. This basically describes a squared grid with grid size g where the stations are translated out of their regular position by a random distance $rnd(g)$ between 0 and g. Each station that is not located on the boundary of the grid is connected by edges with its four direct neighbors $(u-1;v)$, $(u+1;v)$, $(u,v-1)$, and $(u;v+1)$ (cf. Fig. 2). We further randomly generate a list of 10.000 stations as an order list. As a result, we obtain three tables which are used as input data for all simulation tools under consideration:

- Stations: List of all stations (nodes) in the graph with coordinates.
- Tracks: List of all tracks (edges) in the graph with the two connected stations and the length of the track.
- Transport orders: List of 10,000 stations to be visited subsequently by one single vehicle.

At first, we have defined a set of 13 different graphs ranging from 100 nodes to 22,500 nodes. The structure along with the number of edges and the average number of visited nodes per transport order are given in Tab. 1.

A second set of network instances is designed for analyzing transport networks containing sparsely connected clusters of densely connected stations. To obtain such instances we divided the original graphs into sub-graphs and eliminated connecting edges between adjacent sub-graphs. For graphs with more than 2,000 nodes, we defined sub-graphs with an average size of 610 nodes (cf. Tab. 1, where the number of sub-graphs and the average number of connecting edges between adjacent sub-graphs is given). Starting with these graph structures, we reduced the number of connecting edges between adjacent sub-graphs.

Table 1: Problem instances for the empirical study.

number of nodes	number of edges	avg. length of shortest path (num. of nodes)	number of sub-graphs	avg. number of edges between adjacent sub-graphs
$10\times 10 = 100$	180	6.67	1	-
$25\times 25 = 625$	1,200	16.67	1	-
$34\times 34 = 1,156$	2,245	22.67	2	34
$41\times 41 = 1,681$	3,282	27.33	4	20.5
$47\times 47 = 2,209$	4,327	31.33	4	23.5
$52\times 52 = 2,704$	5,308	34.67	4	26
$57\times 57 = 3,249$	6,389	38	6	23.75
$81\times 81 = 6,561$	12,966	54	12	23.63
$98\times 98 = 9,604$	19,019	65.33	16	24.5
$114\times 114 = 12,996$	25,772	76	20	25.65
$127\times 127 = 16,129$	32,013	84.67	25	25.4
$139\times 139 = 19,321$	38,374	92.67	30	25.48
$150\times 150 = 22,500$	44,711	100	36	25

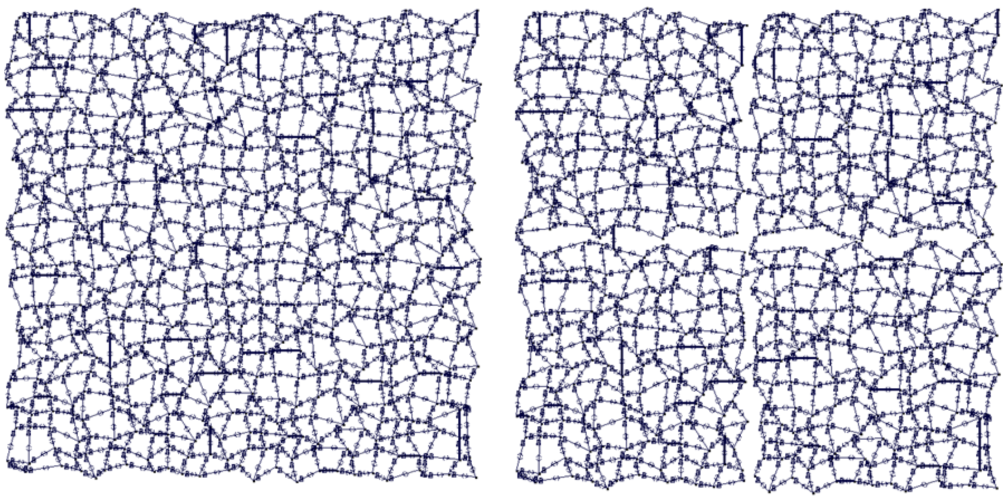

Figure 2: **Left:** Original graph with 625 nodes. **Right:** Graph with four sub-graphs and two remaining edges between adjacent sub-graphs.

For one set of problem instances we reduced the number of connecting edges to eight, for another set we reduced it to only two edges (see Fig. 2 for an example with two remaining edges).

5.2 Findings

Three different experiments were performed. All of them were executed on a workstation with a 3.07 GHz Intel Xeon CPU and 6 GB RAM operating on the 64-bit version of Windows 7.

Experiment 1 At first, the performance of the three simulation tools mentioned above was compared. Only Plant Simulation was able to cope with all problem sizes. Both Enterprise Dynamics and AutoMod were not able to generate or run a model with more than 52×52 (2,704) nodes on the hardware tested. AutoMod seems to have difficulties with fully utilizing the physical memory under the 64-bit version of Windows 7: The 'insufficient memory' message appeared when only 3 GB RAM were actually in use.

As stated above, both Enterprise Dynamics and AutoMod generate adjacency matrices for storing the shortest paths before model execution. These matrices get to big for the available RAM-size for large models. As can be seen in Tab. 2, the model size increases with the squared number of nodes. Because Plant Simulation does not perform any pre-calculation of shortest paths, the model size grows only linearly with the number of nodes and it can handle all problem sizes tested. As we have also implemented Floyd-Warshall's algorithm as an external program, but store the necessary matrices in a Plant Simulation model, we have a value for comparison of model size, when setting up only one sub-graph (see Tab. 2).

Table 2: File size of compiled simulation model [MB].

number of nodes	AutoMod	Enterprise Dynamics	Floyd-Warshall's algorithm (Plant Simulation)
$10 \times 10 = 100$	3.13	0.78	0.60
$25 \times 25 = 625$	91.60	10.61	2.74
$34 \times 34 = 1,156$	309	33.75	6.48
$41 \times 41 = 1,681$	652	70.34	13.38
$47 \times 47 = 2,209$	1,100	120.62	22.15
$52 \times 52 = 2,704$	1,650	124.95	32.44
$57 \times 57 = 3,249$	-	-	46.00
$81 \times 81 = 6,561$	-	-	178.45

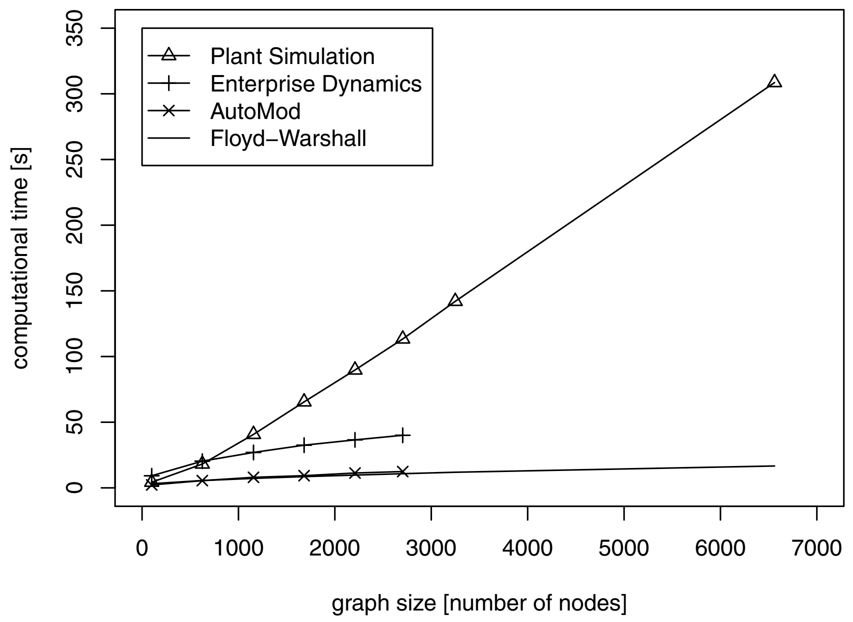

Figure 3: Computational times for model execution with different tools.

Using our implementation, Plant Simulation can cope with 6,561 nodes, which is more than twice the size that Enterprise Dynamics or AutoMod can handle.

Two aspects have to be considered when comparing the computational effort: The time for running a simulation and the time for preprocessing the adjacency matrix. Model execution takes much more time using Plant Simulation's built-in algorithm than with other approaches, as one should expect (see Fig. 3). AutoMod is about 70% faster than Enterprise Dynamics. However, our own implementation of Floyd-Warshall's algorithm is slightly faster than even AutoMod (about 10% in direct comparison).

The advantages in model execution have to be paid for with an additional effort for pre-calculating and storing shortest paths. To perform this calculation on the 52×52 instance requires AutoMod 219 s, Enterprise Dynamics 75 s, and our implementation of Floyd-Warshall's algorithm 185 s. The computational time grows with the cube of the number of nodes. Our implementation of Floyd-Warshall's algorithm required 2,510 s to pre-calculate the shortest paths on the 81×81 instance. Based on these numbers, a break-even-point can be calculated: Our implementation is to be preferred over Plant Simulation's standard functionality, if more than 18,113 transport orders have to be fulfilled in the 52×52 graph, or more than 86,210 transport orders in the 81×81 graph. Note, however, that the computational times for Floyd-Warshall's algorithm include the time for the export and import of all necessary matrices (since we use an external Java program) and a routine to copy and transform the results to internal data structures within Plant Simulation. Probably, the computational time for preprocessing could be reduced with an improved implementation.

Experiment 2 The next experiment evaluates the hierarchical approach presented in Sec. 3. Compared to Floyd-Warshall's original algorithm, our hierarchical approach is significantly slower, of course. However, it is still about 50% faster than Plant Simulation's standard algorithm and able to handle at least the 150×150 graph with 22,500 nodes, which is superior to all implementations of Floyd-Warshall's algorithm discussed in this paper. Tab. 3 shows the computational times for preprocessing and model execution. For simulating the fulfillment of random transport orders in the original graph, there is a certain critical number of orders (see column 'Break-Even' in Tab. 3): If more than this number of transport orders has to be simulated, the hierarchical version of Floyd-Warshall's algorithm is less time-consuming than Plant Simulation's internal algorithm. Since road networks are usually stable in the course of a simulation study and many replications

Table 3: Comparison of Plant Simulation with the hierarchical version of Floyd-Warshall's algorithm.

number of nodes	Plant Simulation's internal algorithm	Hierarchical version of Floyd-Warshall's algorithm		Break-even [number of transport orders]
	model execution [s]	preprocessing [s]	model execution [s]	
$10 \times 10 = 100$	4.31	0.41	3.14	3,473
$25 \times 25 = 625$	18.16	4.24	5.66	3,395
$34 \times 34 = 1,156$	40.72	7.36	17.02	3,108
$41 \times 41 = 1,681$	65.47	8.35	28.97	2,286
$47 \times 47 = 2,209$	89.73	13.71	38.28	2,665
$52 \times 52 = 2,704$	113.35	20.79	47.39	3,153
$57 \times 57 = 3,249$	142.01	20.53	64.60	2,652
$81 \times 81 = 6,561$	308.57	47.72	140.53	2,840
$98 \times 98 = 9,604$	459.95	83.80	210.54	3,360
$114 \times 114 = 12,996$	634.70	139.54	318.66	4,415
$127 \times 127 = 16,129$	816.04	200.93	396.90	4,794
$139 \times 139 = 19,321$	980.38	286.75	493.23	5,886
$150 \times 150 = 22,500$	1,171.86	396.77	571.77	6,612

have to be performed in order to achieve an acceptable confidence level, our algorithm should be preferred over Plant Simulation's algorithm in many cases.

Experiment 3 The last experiment analyzes the influence of the graph structure on the performance of the hierarchical version of Floyd-Warshall's algorithm: Connecting edges between the sub-graphs were eliminated as described in Sec. 5.1. In a first step, eight edges remained between adjacent sub-graphs. As we start with an average of about 25 connections between each two adjacent sub-graphs, we still keep about 30% of the original number of connecting edges. The computational time for preprocessing was not effected by this change. However, a significant reduction of simulation time can be observed (see Fig. 4). The computational times for model execution are reduced by 69% (for the 52×52 graph) up to 89% (for the 150×150 graph). If adjacent sub-graphs are connected by two edges only, the runtime advantage becomes even more pronounced and reaches 93% for the 150×150 instance. While the performance of the hierarchical version of Floyd-Warshall's algorithm strongly depends on the network structure, the execution time for the internal algorithm of Plant Simulation differs not significantly.

Of course, the 'natural' clusters of the underlying graph have to be reflected in the simulation model when using the hierarchical version of Floyd-Warshall's algorithm, i.e., the appropriate sub-graphs have to be identified. In general, graph partitioning is a computationally intensive and therefore time-consuming task. However, when simulating material flows and plant logistics systems (which is the major use case for the simulation tools considered in this study), the sub-graphs usually correlate with plant buildings and manufacturing lines. In this case, the partitioning can be done manually in the course of model development.

6 CONCLUSIONS AND FURTHER WORK

We have compared three simulation tools with respect to their performance concerning the automatic routing of vehicles. Different strategies are pursued by Plant Simulation on the one hand and AutoMod and Enterprise Dynamics on the other hand. While the latter two systems pre-calculate all possibly relevant shortest paths before the actual simulation run, Plant Simulation performs this search for each transport order during model execution. It depends on the simulation study in question which approach is to be preferred: If the same paths have to be used many times, the pre-calculations of AutoMod and Enterprise Dynamics will pay off. However, these tools are not able to simulate very large transportation systems. Only Plant Simulation proved to be able to handle networks with more than 3,000 nodes.

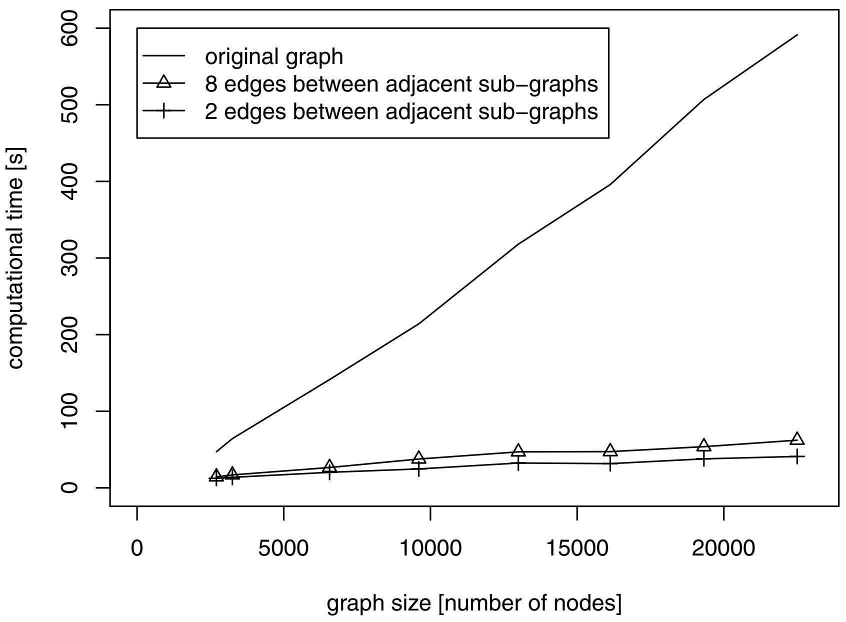

Figure 4: Computational times for all problem instances for the hierarchical Floyd-Warshall's algorithm.

When comparing different simulation systems, not only the time required for model execution has to be considered. Another—often more important—aspect is the effort for model creation, since this requires the work of a simulation engineer. According to our subjective impression, there are larger differences in usability than one could expect after decades of software development for discrete-event simulation.

In addition to the routing algorithms available in standard simulation software, we presented an extension of Floyd-Warshall's algorithm. We implemented this algorithm in Plant Simulation to use it instead of Plant Simulation's internal algorithm. Our approach is feasible for large networks and static road networks. It delivers superior results for the network structures analyzed. It is very well suited, if there are loosely coupled clusters of densely connected nodes. This is a typical characteristic of road networks that connect towns, but may also be found in plant structures.

Future research could be done on analyzing different graph structures, and further tools could be considered also. Considering the presented hierarchical approach to find shortest paths based on Floyd-Warshall's algorithm, improvements could be implemented and tested. Furthermore, research could concentrate on computing optimal partitions of given graphs such that the number of connecting edges is minimized while the number of nodes per sub-graph does not exceed a certain limit.

REFERENCES

Bangsow, S. 2010. *Manufacturing Simulation with Plant Simulation and Simtalk: Usage and Programming with Examples and Solutions*. Berlin: Springer.

Bellman, R. 1958. "On a routing problem". *Quarterly of Applied Mathematics* 16:87–90.

Berge, C., and A. Ghouila-Houri. 1965. *Programming, Games, and Transportation Networks*. New York: Wiley.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. *Introduction to Algorithms, 2nd Ed.* New York: MIT Press and McGraw-Hill.

Dijkstra, E. W. 1959. "A note on two problems in connexion with graphs". *Numerische Mathematik* 1:269–271.

Floyd, R. W. 1962. "Algorithm 97: Shortest path". *Communications of the ACM* 5:345.

Fredman, M. L., and R. E. Tarjan. 1987. "Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the Association for Computing Machinery* 34:596–615.

Hart, P. E., N. J. Nilsson, and B. Raphael. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

Hart, P. E., N. J. Nilsson, and B. Raphael. 1972. "Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths"'. *SIGART Newsletter* 37:28–29.

Johnson, D. B. 1977. "Efficient algorithms for shortest paths in sparse networks". *Journal of the ACM* 24:1–13.

Knopp, S., P. Sanders, D. Schultes, F. Schulz, and D. Wagner. 2007, January 6. "Computing Many-to-Many Shortest Paths Using Highway Hierarchies". In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 36–45. New Orleans, Louisiana.

Lindemann, M., and S. Schmid. 2007. "Marktüberblick: Simulationswerkzeuge in Produktion und Logistik". *PPS Management* 12 (2):28–35.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Palo Alto: Tioga Publishing Company.

Schultes, D. 2008. *Route Planning in Road Networks*. Ph. D. thesis, Karlsruhe University.

Sturtevant, N. R., and R. Geisberger. 2010, October 11–13. "A Comparison of High-Level Approaches for Speeding Up Pathfinding". In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*, 76–82. Palo Alto, California.

Vahrenkamp, R., and D. C. Mattfeld. 2007. *Logistiknetzwerke: Modelle für Standortwahl und Tourenplanung*. Wiesbaden: Gabler.

Warshall, S. 1962. "A theorem on boolean matrices". *Journal of the ACM* 9:11–12.

Willhalm, T. 2005. *Engineering Shortest Paths and Layout Algorithms for Large Graphs*. Ph. D. thesis, Karlsruhe University.

Zhan, F. B., and C. Noon. 1998. "Shortest path algorithms: An evaluation using real road networks". *Transportation Science* 1:65–73.

AUTHOR BIOGRAPHIES

KAI GUTENSCHWAGER studied Business Informatics and received his doctoral degree from the Technical University of Braunschweig. He worked for SimPlan as the head of the office in Braunschweig being a simulation expert in the field of logistics and supply chain management. Since 2009, he is professor for information systems in logistics at Ulm University of Applied Sciences. His research is focused on IT-based methods for the simulation and optimization of production and logistics systems.

AXEL RADTKE studied mechanical engineering at the University of Dortmund, Germany. While working at Fraunhofer IML and chair of logistics from 1993 he began focusing on simulation studies in logistics. After doing his doctorate he continued working at a planning company and is general manager of the SimPlan Integrations GmbH, Witten since 2001.

SVEN VÖLKER studied Business Informatics and received his doctoral degree from Ilmenau Technical University. He worked for Tecnomatix, UGS and Siemens PLM Software as a solution architect and project manager in the field of Digital Manufacturing and Product Lifecycle Management. Since 2010, he is Professor for Logistics Planning and Digital Manufacturing at Ulm University of Applied Sciences. His research is focused on IT-based methods for planning, simulation and optimization of production and logistics systems.

GEORG ZELLER studied mechanical engineering and business science at Munich Technical University. He worked as a design engineer (Mengele), head of R&D (Megamat) and CEO of a machine factory (Rampp). Today, Georg Zeller is CEO of InduSim GmbH, a company for numerical and logistic simulation. Furthermore, he is head of the Simplan office in Holzgerlingen.