A Comparison Between Dynameq and Indy

Maaike Snelder

November 2009

CIRRELT-2009-48

A Comparison Between Dynameq and Indy

Maaike Snelder[1]

« In cooperation with Michael Florian and Michael Mahut »

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT) and Delft University of Technology, Postbus 5, 2600 AA Delft, The Netherlands

Abstract. This report is the result of six weeks study carried out at the Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT). In this study a comparison has been made to show the similarities and differences between the dynamic traffic models Dynameq and Indy. More specifically the aim of the comparison was threefold:

1. Identify the differences in the specifications of both models;
2. Find the differences in computational efficiency;
3. Show the differences in the outcomes of models and show the practical implications of that: "What goes wrong if no lanes are modelled?"

This study was a cooperation between the Netherlands Organisation for Applied Scientific Research (TNO), the Delft University of Technology in the Netherlands, CIRRELT and INRO both in Montreal, Canada.

Keywords. Dynamic traffic assignment, traffic flow theory, comparison of dynamic assignment results.

Acknowledgements. Financial support for this work was provided by the Natural Sciences and Engineering Council of Canada (NSERC). This support is gratefully acknowledged. The author would especially like to thank Michael Florian, Michael Mahut and Nicolas Tremblay for being an excellent host and for answering many questions about Dynameq.

* Corresponding author: maaike.snelder@tno.nl

Dépôt légal – Bibliothèque et Archives nationales du Québec,
Bibliothèque et Archives Canada, 2009

Table of contents

Summary

Since many years dynamic models have been developed and they are being applied more and more often. Traditionally, these models are classified in microscopic, mesoscopic and macroscopic models. These models differ in the level of detail in the network (e.g. with or without explicit lane modelling) and the level on which the traffic dynamics are modelled (vehicle level, flow level or a combination of both). All models aim to reproduce the traffic situation in the best possible way within there intended scope. It is interesting to make a comparison between models, to learn how differences in model specification lead to different outcomes. This report doesn't intend to compare all dynamic models with each other, but shows the differences between Indy and Dynameq. Indy is a macroscopic dynamic traffic model that has been developed in the Netherlands and Belgium by TNO, the Delft University of Technology and the Catholic University of Leuven. Dynameq is based on a traffic simulation model that was designed to produce reasonably accurate results with a minimum number of parameters and a minimum of computational effort (Astarita et al., 2001) (Mahut, 2000). However, the underlying structure of the model has more in common with microscopic than with mesoscopic approaches, as it is designed to capture the effects of car following, lane changing and gap acceptance. Dynameq has been developed in Canada by the University of Montreal and Inro.

The aim of the comparison is three fold:

1. Identify the differences in the specifications of both models:

Dynameq and Indy are two different dynamic assignment models. One of the most important differences is that the first is lane based and the second link based. A comparison of differences in specifications will indicate what the differences exactly are.

2. Find the differences in computational efficiency:

Since both models are specified in a different way, they converge in different ways to an equilibrium. Besides that, the computation time will differ per iteration, because different network loading algorithms are used. The question is how large these differences are and what causes them.

3. Show the differences in the outcomes of models and show the practical implications of that: "What goes wrong if no lanes are modelled?"

At first a comparison is made between the model specifications to get an understanding of the similarities and differences between both models. The comparison of the specifications of Dynameq and Indy showed that there are several important similarities and differences between both models. The models are comparable in the sense that both are equilibrium models in which paths are generated, path choice plays a role and dynamic network loading takes place in an iterative process. The newest and most accurate network loading model in Indy (LTM) works according to Newell's kinematic wave theory and so does the network loading algorithm in Dynameq. The main differences are:

- Dynameq generates paths in the first iterations of the simulation, whereas Indy generates paths before the simulation starts.
- Dynameq has a deterministic path choice, whereas Indy has a stochastic path choice.
- Dynameq is lane based, whereas Indy is linked based.
- Dynameq models individual vehicles, whereas Indy models aggregated flows (per path). This enables Dynameq to model gap acceptance at intersections and lane changing behavior, which can't be done by Indy.
- Dynameq has a more detailed intersection model than Indy. It can deal with traffic signals and priority flows, whereas Indy can only approximate this by introducing a maximum link outflow capacity.
- Dynameq is event based, whereas Indy works with fixed time steps.

Both models are run on three networks with several scenarios to show how the above mentioned differences influence the model outcomes. The first test network was a network in which delays at intersections where excluded and in principle two equal paths are available for the single OD-pair. To get an idea of what the network looks like, the network of the first scenario is shown below.

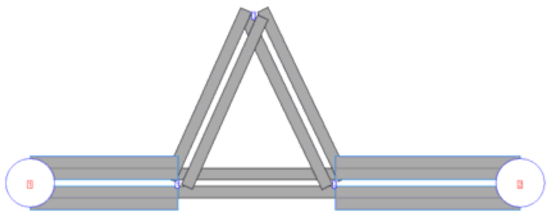


Figure S.1 : Network test case 1.

It showed that:

- That stochastic path choice of Indy in combination with generating trips can have advantages over deterministic path choice, since already in the first iteration the traffic is spread over all the pre-specified paths. In this way Indy converges faster to an equilibrium than Dynameq. This general conclusion is illustrated by the fact that in the first scenario Dynameq doesn't find a second path because there is no congestion on the first path. Therefore, in Dynameq the capacity of the network is not used fully which results in the fact that it takes Dynameq twice as long to get all the traffic over the network. It must be said that this is an extreme example, because in larger networks links will be used by multiple paths from multiple OD-pairs, which always causes some delays and therefore extra paths to be generated. The differences in path choice also became clear in the second scenario where Indy finds a perfect equilibrium in which 50% of the traffic uses each path. The spread in Dynameq after the first 30 iterations is 53%-47%.
- In the second scenario Dynameq does find two paths, which makes the results of Dynameq and Indy more comparable. In fact if Indy uses the paths of Dynameq, the results are almost identical. Which shows that both network loading models are the same if delays at intersections and lane changing behavior doesn't play a role as was to be expected based on the model specifications.
- In this test network the links are very long which allows for a high time step in Indy. The time step could even be set to 180 seconds. However a time step of 60 seconds is used. Even with this time step Indy is about 4 times as fast as Dynameq. This is probably caused by the fact that Indy is a macroscopic model and, therefore, doesn't have to keep track of individual vehicles and the way in which they behave. On the other hand, if the links, or even only one of the links, would have been shorter, a smaller time step had to be chosen which would increase the computation time of Indy. In that case Dynameq becomes faster than Indy which is illustrated better on the Bakersfield network (shown below). This shows the consequences of having event based or time step based models.

The second test network is a network with four zones and flows between all zones. There is only one path between each OD-pair. Therefore path choice doesn't play a role. The network is shown below.

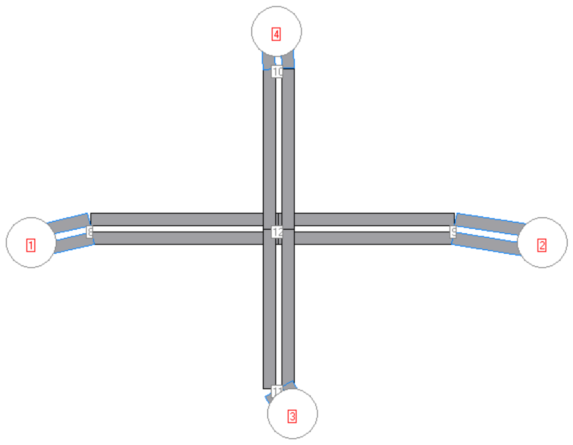


Figure S.2 : network scenario 1 and 2, test case 2.

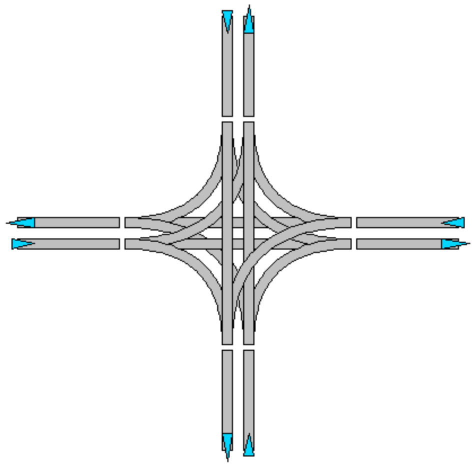


Figure S.3 : intersection, test case 2.

This example emphasizes the differences between both models at intersections:

- In the first scenario the intersection is unsignalized. In this case the differences in model outcomes are very large. The total travel in Indy is about twice as low as in Dynameq and the network is almost an hour earlier empty. The explanation for this is that Indy doesn't consider delays at intersections that are caused by waiting for gaps that are needed to cross an intersection. In Indy cars can virtually drive over each other, which is of course not realistic. The fact that the travel times are twice as low shows that the delays at intersections can add up to a substantial amount. On the other hand, the flows in this network are high. If there is less traffic in the network, the delays at the intersections also reduce.
- The two scenarios with signalized intersections showed that it is possible to model signalized intersections by outflow constraints, because the results of Indy and Dynameq are exactly the same for both scenarios. This is however only possible in the situation in which there are no conflicts in lane usage, because in those situations the outflow is a result of the arrival pattern of traffic on the intersection. Therefore, the maximum outflow cannot be computed based on the link capacity, green times and cycle lengths. For those situations an approximate outflow capacity has to be found.

Finally, both models are compared on a realistic network: the Bakersfield network. This network is shown below. It has 36 signalized intersections (green nodes).

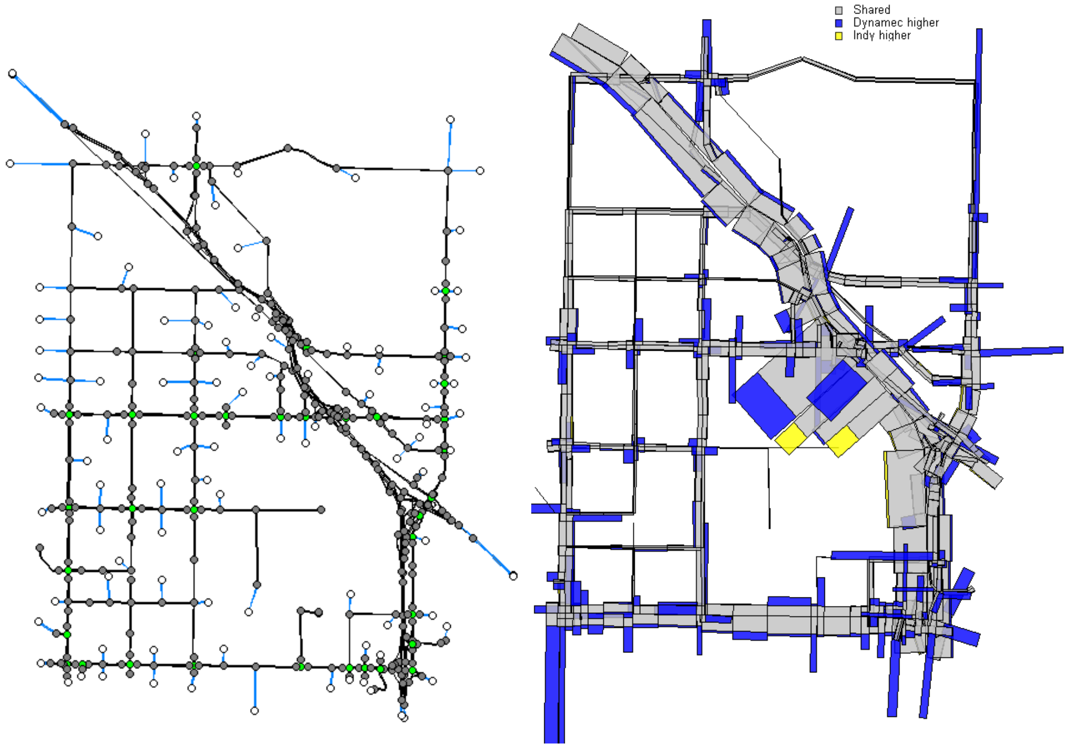


Figure S.4 : Bakersfield network (left) and differences in model outcomes (density) in the case in which Dynameq is run with signalized intersections and Indy is run with the paths of Dynameq.

All the differences mentioned above become more clear in this network. Besides that some additional differences became clear:

- In the Bakersfield network congestion occurs on the motorway due to lane changing behavior. This congestion is recognized by Dynameq. Since Indy doesn't model the described lane changing behavior, it doesn't find the congestion on the motorway. An approximate outflow restriction is imposed in Indy on the link upstream of the motorway junction where the congestion starts. However, this appears not to capture the dynamics in the traffic flow caused by lane changing behaviour completely.
- Dynameq is run with signalized and unsignalized intersections and Indy is run twice with the path and path flows of these runs and once with its own paths and path choice. The average speeds in the Dynameq outcomes of the signalized network are about 12 km/hour lower than in all the other model runs. Thus, also in the model run in which Indy uses the paths and path flows produced by the same Dynameq run. This is explained by the fact that Indy doesn't model the delays at the intersections. In

the case in which Dynameq is run with unsignalized intersections the average speeds come very close to the average speeds computed by Indy.

- The results of the three model runs with Indy are very close to each other. This suggests that the equilibrium that is found by Indy is close to the equilibrium that is found by Dynameq. It is remarkable that the equilibrium run of Indy is in fact closer to the equilibrium run of Dynameq with signalized intersections than the equilibrium run with Dynameq with unsignalized intersections, because Indy doesn't use signals. From the total vehicle kilometers driven it can be seen that the slightly shorter paths (shorter in distance) are chosen if the signals are not used. This suggests that the signals on the shorter paths cause delays which makes travelers choose longer (in distance) paths.
- The development of the average speed, the total travel time and the average lane density in the network over time for the five different model runs is more or less the same over time in all Dynameq and Indy runs. Which indicates that, despite all the before mentioned differences, the model outcomes also show a lot of similarities.
- A comparison over time per link showed that there is a very high correlation between Dynameq and Indy if the inflow and outflow are compared. In the case in which Indy uses the paths of Dynameq this is logical. Although even in that case delays can cause differences in inflow and outflow over time (but not over links). The fact that the inflow and outflow have a R^2 above 0.95 indicates that the equilibrium route choice of both models doesn't differ much. However, the R^2 of the speeds is low (0.24). For a large part this is caused by links at intersections. The R^2 of the density is even worse (0.18 or 0.19) than the speed in case of signalized intersections. However, in the case of unsignalized intersections the R^2 goes up to 0.67 and 0.76. This emphasizes the need to model delays at intersections explicitly. The right hand figure of Figure S.4 shows that the differences are indeed for a large part located near intersections.
- Dynameq converges slower and keeps higher gaps also in the later iterations. The first cause for this is that Dynameq models more delays (and therefore bigger differences between paths). This hypothesis is strengthened by the fact that the gaps in the case without signals are already much lower (below 10%) than in the case with signals in which the gaps go up to 70%. A second cause is that Dynameq hasn't found all the paths yet in the first 10 iterations, which causes the high gaps in the first iterations. Finally, the fact that Dynameq uses a deterministic assignment results in the fact that the traffic is less spread over all available paths.
- In the equilibrium run with Indy a time step of 5 seconds was used. This time step is slightly larger than the free flow link travel time of 141 links. The smallest free flow travel time is 1.3 seconds. This implies that for these links the link lengths had to be extended during the simulation in such a way that they have a free-flow travel time of 5 seconds. With this time step the computation time per iteration is still almost 10 times as high as in Dynameq, which is likely to be caused by the fact that Dynameq is event based and many links have a much higher free flow travel time than 5 seconds. A second explanation might be the number of used paths. In total there are 2988 OD- pairs with a demand larger than 0. Indy generated 8399 paths and there is flow on all these paths. Dynameq generated 19692 paths, but there is only flow on 6200 paths. Since the computation time of Indy depends on the number of paths that are used in the evaluation phase, this could also be an explanation for the longer computation times. In the comparison in which the Dynameq paths are used, a time step of 1 second is used in Indy. Therefore, in these runs the link lengths didn't have to be extended. For these runs only one iteration was needed. This iteration took 37 minutes, which is more than 5 times (9.4) higher than the case with a time step of 5 seconds. The explanation for this might as well be the number of paths. Although, only 6200 paths are used, all 19692 paths are considered in Indy.

The above mentioned differences lead to the following recommendations for Dynameq and Indy:

Dynameq:

- Generating the paths before the simulation starts or storing the paths of a previous model run can reduce the number of iterations that is needed to reach an equilibrium and therefore decrease the computation time. It might be worthwhile to investigate if this is possible.
- Stochastic route choice instead of deterministic route choice could lead to faster convergence as well. However, changing this has bigger implications, since it is a fundamental change in the assumed route choice behavior.

Indy:

- The modelling of delays at intersections can be improved. For the level on which Indy is currently mostly used (high level, with mainly motorways) this is less important than for cases in which local networks are used. However, it could still be large improvement. This requires more input data, but could make the calibration easier in the end. A first improvement which is probably relatively easy is the introduction of the option the include maximum outflow constraints per movement instead of per link. This prevents that links has to be split in three separate links to replicate the structure of a node. Other improvements that are for example needed to include the gap acceptance principle might be investigated as well which is already being done by the university of Leuven.
- Including lane changing behavior is not possible in Indy. It might be worthwhile to investigate how this behaviour can be approximated.
- A practical suggestion is not to use the adjusted link capacities from networks that are used in static assignment models and adjust those capacities a bit further in the calibration, but to reset the capacities to the level that is to be expected based on free-flow speeds, average vehicle lengths and a response time. In the calibration the maximum outflow capacities can then better be adjusted instead of the capacities themselves.
- Investigate the possibility and the gains in computation time of switching from a time step based network loading model to an event based model. This is probably relatively easy since the university of Leuven already has an event based version of LTM.
- Remove paths that are not used or barely used during the simulation to reduce the memory usage and increase the computation speed.

1 Introduction

Since many years dynamic models have been developed and they are being applied more and more often. Traditionally, these models are classified in microscopic, mesoscopic and macroscopic models. These models differ in the level of detail in the network (e.g. with or without explicit lane modelling) and the level on which the traffic dynamics are modelled (vehicle level, flow level or a combination of both). All models aim to reproduce the traffic situation in the best possible way within there intended scope. It is interesting to make a comparison between models, to learn how differences in model specification lead to different outcomes. This report doesn't intend to compare all dynamic models with each other, but shows the differences between Indy and Dynameq. Indy is a macroscopic dynamic traffic model that has been developed in the Netherlands and Belgium by TNO, the Delft University of Technology and the Catholic University of Leuven. Dynameq is based on a traffic simulation model that was designed to produce reasonably accurate results with a minimum number of parameters and a minimum of computational effort (Astarita et al., 2001) (Mahut, 2000). However, the underlying structure of the model has more in common with microscopic than with mesoscopic approaches, as it is designed to capture the effects of car following, lane changing and gap acceptance. Dynameq has been developed in Canada by the university of Montreal and Inro.

The aim of the comparison is three fold:

4. Identify the differences in the specifications of both models:

Dynameq and Indy are two different dynamic assignment models. One of the most important differences is that the first is lane based and the second link based. A comparison of differences in specifications will indicate what the differences exactly are.

5. Find the differences in computational efficiency:

Since both models are specified in a different way, they converge in different ways to an equilibrium. Besides that, the computation time will differ per iteration, because different network loading algorithms are used. The question is how large these differences are and what causes them.

6. Show the differences in the outcomes of models and show the practical implications of that: "What goes wrong if no lanes are modelled?"

In chapter 2, a comparison between the model specifications is made. It shows the specifications of both Dynameq and Indy and the similarities and differences between them. The performances of Dynameq and Indy is tested by running both models on two very small test networks and on the larger existing Bakersfield network. The cases are described in chapter 3. This chapter also describes the model outcomes of both models for the cases and gives some explanations for the differences. In chapter 4, the conclusions and recommendations are described.

2 Comparison of model specifications

This chapter describes the model specifications of Dynameq and Indy and gives an overview of the similarities and differences between both models. The description of Dynameq is copied from the help files of Dynameq. The description of Indy is mainly based on the Indy specifications (Bliemer, 2005) and the description of the link transmission model (Yperman, 2007). For practical reasons the notation that is used is kept similar to the original notation of both models. As a result different symbols are used in the description of Dynameq and Indy for similar variables.

More information about Dynameq and Indy can be found in (Bliemer, 2007), (Florian et al., 2008) (Mahut, 2000), (Mahut et al, 2004), (Mahut et al., 2008).

2.1 Description of Dynameq

2.1.1 Introduction

Dynameq captures congestion due to traffic signals, conflicting intersection movements, flow capacity, lane changing, and heavy vehicles. Dynameq then models how this congestion is propagated across lanes and links through upstream intersections. The equilibrium approach of the Dynameq DTA produces chosen paths that are consistent with drivers' desire to minimize their travel costs.

Dynameq's traffic simulator provides the necessary fidelity for a wide range of applications: Evaluate the impacts of congestion relief strategies, such as infrastructure expansions. Plan road, ramp, and lane closures for maintenance and construction projects. Model lane management strategies with multiple vehicle classes, such as truck prohibitions and reserved lanes for buses, taxis, or high occupancy vehicles. Evaluate alternative pre-timed signal control and ramp metering plans by modelling how drivers will adapt their route choices to them. Determine how traffic should be re-routed in response to incidents in critical locations, or what the impacts would be on day-to-day operations of the loss of a major bridge or other critical infrastructure.

Dynameq is a dynamic traffic assignment (DTA) model based on the principle of dynamic user equilibrium. In an equilibrium approach to DTA, the objective is to minimize each driver's travel time so that for each origin-destination (O-D) pair, vehicles leaving the origin at roughly the same time have approximately the same travel time to the destination.

Dynameq accomplishes this with an iterative method, where each iteration consists of one execution of a path-choice model and one execution of a traffic simulation. The traffic simulator receives the time-dependent path flow rates from the path-choice model, and simulates the resulting traffic patterns on the network. The simulator then provides time-dependent travel time information back to the path-choice model, which consequently modifies the path choices for the next iteration. Thus the output of each of these two models is the input to the other. The process continues cyclically until converging to an approximate state of dynamic user equilibrium. The iterations can be thought of as a sequence of consecutive days, where drivers start out on the first day with knowledge of the network but no knowledge of the traffic patterns that will result from their path choices. Each day, after experiencing the resulting traffic congestion, each driver considers the possibility of choosing a different path for the next day. After a certain number of days, drivers stop looking for new paths and restrict their choices to paths that they have already tried. The iterative process is shown in Figure 2.1.

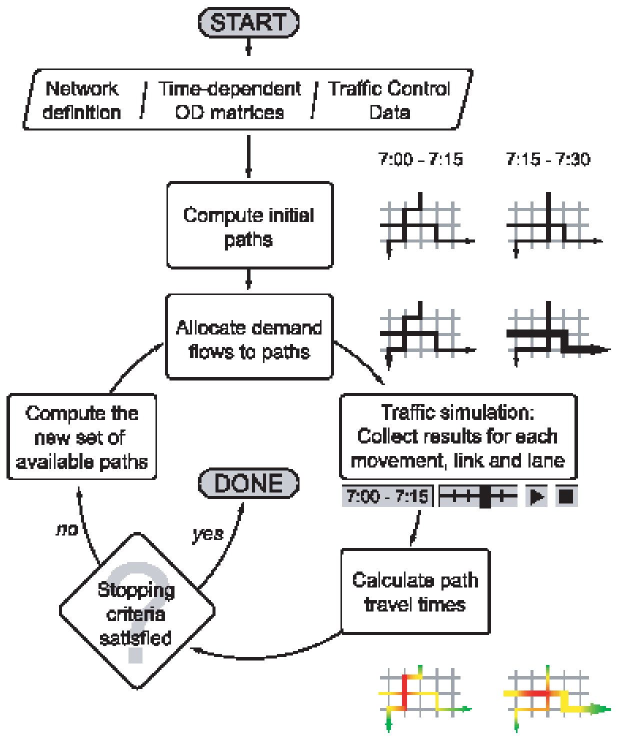

START
Network definition
Time-dependent OD matrices
Traffic Control Data
7:00 - 7:15
7:15 - 7:30
Compute initial paths
Allocate demand flows to paths
Compute the new set of available paths
Traffic simulation: Collect results for each movement, link and lane
7:00 - 7:15
DONE
no
yes
Stopping criteria satisfied
Calculate path travel times

Figure 2.1 : Flow chart Dynameq.

2.1.2 Input

Dynameq is designed to minimize data requirements and to provide useful defaults if data is unavailable. The inputs to a Dynameq DTA are the traffic demand, network definition and traffic control plans. The traffic demand is represented by one time-dependent O-D matrix for each class of vehicle being modelled. The representation of the road network is at the level of individual lanes and includes the definition of turn pockets. The permitted lanes for each intersection movement and lane prohibitions by vehicle class are specified where necessary. The network definition does not require lane width or detailed intersection geometry, such as turning radii.

2.1.3 Traffic generation

Traffic generation is the process by which an O-D flow rate in a specific table of an O-D matrix is converted into a sequence of departure times of individual vehicles. Three methods are available, which differ in the way that randomness is introduced into the number of vehicles generated and/or their departure times. The three methods (Poisson, Conditional and Constant) are described as follows:

- Poisson: the Poisson traffic generator produces vehicle departure times as a Poisson Process, in which case both the number of vehicles generated and the departure times are random. The number of vehicles generated follows the Poisson distribution, while inter-departure times (the duration of time between two sequential departures) follows the negative exponential distribution. Because the variance of the Poisson distribution is equal to the mean, the variability in the number of vehicles generated using this method may in some cases be higher than desired.
- Conditional: the Conditional traffic generator has almost no variability in the number of vehicles generated. For each O-D pair, for each table of the O-D matrix, the number of vehicles generated is obtained by multiplying the flow rate by the duration of the corresponding matrix interval. The only variability in the actual number of vehicles generated is due to the procedure by which the real-valued result of this product is rounded to obtain an integer number of vehicles. A bucket-rounding procedure is used which ensures that the total number of vehicles generated at an origin for a specific matrix interval is equal to the sum of the flow rates (row sum) multiplied by the duration of the interval (rounded to the

nearest integer). The inter-departure times follow the negative exponential distribution, as when using the Poisson traffic generator.

- Constant: the Constant traffic generator produces the same number of vehicles as the Conditional traffic generator. The inter-departure times are constant, and equal to the duration of the matrix interval divided by the number of vehicles generated. Only the first departure time for each interval and each O-D pair is random, following a uniform distribution over the duration of the inter-departure time.

2.1.4 Path generation

On the first iteration, there are no previously known traffic conditions (no previous iterations), so all drivers choose the quickest path assuming that traffic flows at the free speed of each link. The simulation component then models the movement of all vehicles through the network along these paths. At the end of the simulation, the resulting link travel times are used to find the quickest path for each O-D pair, for each assignment interval.

On the second iteration, for each assignment interval, half the drivers use the original shortest path and half use the new shortest path. This process continues, adding one new path at each iteration, until the maximum number of paths is reached. Thus, if 5 is specified as the maximum number of paths in the DTA specification, the first five iterations are used to find the five best paths for each O-D pair, for each assignment interval. In this case, on the fifth iteration, one fifth of the vehicles use each path for any given assignment interval. These iterations are the path generation stage of the DTA.

2.1.5 Path Choice and Assignment algorithm

In reality, most drivers make decisions based on first-hand knowledge of typical traffic conditions on a network. Typical traffic conditions are not part of the input of a Dynameq DTA. Since traffic conditions are a result of drivers' path choices, the traffic conditions, as well as the path choices, are outputs of the DTA model. The path choice decisions are modelled using an iterative approach, explained as follows.

The iterations of a DTA can be thought of as a sequence of days. On each iteration (or day), every driver makes a decision about which path to use (from origin to destination), for the desired departure time, based on knowledge about the traffic conditions on the previous iteration (or day). The information about traffic conditions that is used is the travel time for each path for the given O-D pair, for the time period that contains the desired departure time. These time periods are called Assignment Intervals. The percentage of drivers choosing each of the available paths, for a given O-D pair, is constant for the duration of each assignment interval. These percentages, as well as the paths themselves, can change from one assignment interval to the next. The travel times are obtained using a traffic simulator that models the traffic conditions that would occur in the network given the path choices of drivers for each iteration of the model.

During the remaining iterations, referred to as the convergence stage of the DTA, the number of vehicles using each path— referred to as the path input flow, or simply path flow for each O-D pair and assignment interval is adjusted before each iteration in order to equilibrate the travel times. Two algorithms are available for this purpose in Dynameq, both of which are based on the well-known Method of Successive Averages (MSA) scheme. The MSA-based algorithms are called regular and flow balancing. When the path choices are such that the travel times on all paths are approximately the same within each assignment interval for each O-D pair, the network is said to be in a state of Dynamic User Equilibrium (DUE).

The assignment algorithms and available options are described briefly below.

- Regular MSA

This algorithm adjusts the path flows by identifying the shortest path for each O-D pair and assignment interval after each full simulation run (execution). The path flow on the shortest (fastest) path is increased, and is decreased for all other paths. The amount of flow added to the shortest path is proportional to 1/n, where n is the iteration number. This algorithm can be used in conjunction with the path pruning, MSA reset and dynamic path search options described below.

-Flow Balancing MSA

This algorithm adjusts the path flows by evaluating the travel times on all used paths and calculating the average path travel time. The path flow is increased for all paths with travel times below the average time, and is decreased for all paths with travel times above the average travel time. The amount of path flow added to or removed from a path is proportional to the difference between the path travel time and the average path travel time. This algorithm can be used in conjunction with the path pruning, MSA reset and dynamic path search options described below.

The following options are available for use with the two assignment algorithms described above:
1. MSA Reset
2. Path Pruning
3. Dynamic Path Search

1) MSA Reset
This option can be used in conjunction with both the regular and flow balancing algorithms. This method adjusts the fraction that determines how much flow is moved to the shortest (regular) or shorter (flow balancing) path(s) as a function of the iteration number and assignment interval. The Reset parameter can take on values ranging from 1 to 5. The default value (3) is recommended.

2) Path Pruning
This option will set the path flow to zero when it drops below the indicated threshold value. This value is the fraction of the total demand for the O-D pair and assignment interval for which the path is defined. For example, a threshold value of 0.01 means that if the path flow calculated (using one of the methods above) drops below 1% of the total demand for that O-D pair and assignment interval, the path flow for this path will be automatically set to zero. The flow that was removed from the path will be distributed proportionately to the other paths. If this option is used without the dynamic path search option described below, the path may again receive a positive flow if its travel time is low enough (depending on which algorithm is being used) on a subsequent iteration.

3) Dynamic Path Search
This option will look for new shortest paths to add to the path set during the convergence stage of the DTA, if there are one or more paths with zero flow. If a new shortest path is found, for an O-D pair and assignment interval for which a zero-flow path currently exists (at a given iteration), the new shortest path will replace the zero-flow path. When using the regular assignment algorithm, the extent to which this option may change the DTA results is closely tied to the threshold value of the path pruning option: the higher the threshold, the more often path flows may be set to zero. Note that for the regular assignment algorithm, path flows can only be set to zero using the path pruning option. The flow balancing assignment algorithm may set path flows to zero without the use of the path pruning option.

Convergence measure:
As the path choices are adjusted on each iteration, the DTA is said to be converging towards equilibrium conditions. Convergence is measured by comparing the average travel time with the shortest travel time (for each O-D pair, for each assignment interval). Dividing the difference in these two values by the shortest travel time allows this difference to be expressed in relation to, or relative to, the actual travel time, so that it can be determined whether the difference is significant or not. This measure of convergence is called the relative gap, because it is obtained in the same way as the relative gap measure used in static assignment models. The average relative gap (RGap) is calculated by Dynameq for each assignment interval (a), for each iteration (n) as follows:

$$RGap^{\alpha,n} = \frac{\sum_{i \in I} \sum_{k \in K_i^{\alpha}} h_k^{\alpha,n} s_k^{\alpha,n} - \sum_{i \in I} g_i^{\alpha} u_i^{\alpha,n}}{\sum_{i \in I} g_i^{\alpha} u_i^{\alpha,n}}$$

Where: I = is the set of all O-D pairs

K_i^{α} = the set of paths for O-D pair i and assignment interval α

In this formula $h_k^{a,n}$ is path flow for path *k* in interval *a* in iteration *n*, g_i^a is the OD-demand for O-D pair *i* in interval *a*, $s_k^{a,n}$ is the travel time for path *k* in interval *a* in iteration *n* and $u_i^{a,n}$ is the shortest travel time for O-D pair *i* in interval *a* in iteration *n.*

Calculation of Path Travel Time:
In order to add a new path at the end of an iteration, it must be possible to determine the travel times of unused paths. The travel time for an unused path is estimated for each assignment interval as follows. The explanation given below is slightly simplified, for explanatory purposes, in that it refers to the path travel time as being composed of link travel times. In reality, it is the average link travel time as measured by vehicles using the specific outgoing movement from the link that is used to estimate the travel time of each path.

After each simulation (iteration), Dynameq calculates the average travel time for each link over short time intervals, such as 5 minutes (the length of this time interval can be modified on the Advanced panel of the New DTA window). For example, consider a 15-minute assignment interval starting at 8:00 and a five-minute interval for the link travel times. The procedure starts in the middle of the assignment interval (8:07:30) and adds on the travel time for the first link of the path corresponding to that time interval. This would be the link travel time for the interval 8:05-8:10. If this travel time is exactly 10 minutes, for example, the estimated arrival time to second link on the path would be 8:17:30. The travel time on the second link corresponding to this time (that is, for the interval 8:15-8:20) is then added on, and the process continues until the estimated arrival time to the destination is obtained.

If, during this process, the estimated arrival time to a link is later than the end of the simulation period, the travel time for this link and the remaining links on the path is not known. This can occur in the early iterations of a DTA if the network does not clear within the specified simulation period. In this case, the last known link travel time (corresponding to the end of the simulation period) is used for all later times, so that a travel time for the complete path can be estimated. If the simulation period is long enough that the network is clearing by the iteration at which all the paths have been added, the exact duration of the simulation period should not have a significant impact on the final path choices.

Multiclass Assignment:
Dynameq is a multiclass model, meaning that multiple vehicle classes may be specified for a DTA. Each vehicle class has its own set of paths for each assignment interval. Class permissions are defined for each intersection movement and each lane in the network, resulting in a unique set of available paths for each class. All Dynameq outputs are available by class. This applies to path-based input flows and travel times, O-D travel times and all simulation results, which are lane, link and movement-based. Convergence results are also calculated by class.

Random Seed:
Traffic simulation models are stochastic models, meaning that there is inherent randomness in the results generated by the model. This type of model makes use of a pseudo-random number generator in order to produce a single set of results for a specific DTA. A pseudo-random number generator starts with a value called the random seed, and produces a sequence of apparently random numbers from it, which are then used by Dynameq to introduce randomness into the model. If a DTA is rerun with the same random seed, the results produced are exactly the same. If a DTA is rerun with a different random seed, the results produced are not the same, but normally should not differ very much.
The traffic generation methods discussed above are a good example of a stochastic component of Dynameq. The same random seed produces the identical sequences of departure times (for the same DTA specification), different random seeds produce somewhat different sequences. Because these two sequences are simply two different ways of interpreting the same O-D matrix, the differences in the DTA results are normally not very significant.
The random seed for a DTA can be edited on the Advanced window when creating a new DTA specification.

Gridlock avoidance:
In Dynameq a deadlock prevention algorithm is implemented that identifies cycles of links that are very close to locking up (during the simulation itself), and manages the inflows to these cycles, much like an adaptive traffic control system, in order to prevent gridlock and maintain traffic flow.

2.1.6 Flow propagation

The three components of Traffic Flow are car following, gap acceptance, and lane changing. These are the interactions between vehicles that lead to traffic congestion.

The simulation is a discrete-event procedure. Unlike discrete-time microscopic simulation models, where the computational effort per link is proportional to the total vehicle-seconds of travel, the computational effort per link required by this model is strictly proportional to the number of vehicles to pass through it, regardless of their travel times.

Car-following:
A car-following model describes how vehicles move on the roadway through time. The Dynameq traffic simulation uses a simplified car-following model that is expressed as follows:

$$x_f(t) = MIN[(x_f(t-R) + V_{free} * R), (x_l(t-R) - L)]$$

Where:	$x_f(t)$	=	position of the following vehicle at time t
	$x_l(t)$	=	position of the leading vehicle at time t
	V_{free}	=	free speed of the roadway
	L	=	effective length of following vehicle
	R	=	response time of following vehicle

The effective length (L) of a vehicle and the driver response time (R) are the microscopic traffic parameters of this model, as they pertain to individual vehicles. In Dynameq, these parameters are defined separately for each vehicle type. Several vehicle types may be defined for each vehicle class. For example, the types long truck and short truck may be defined for the class truck. Car following models also describe the steady-state properties of the traffic on a roadway. Steady-state properties are those that can be observed when the traffic is moving at a constant speed, and thus at a constant flow and density as well. The relationship between the steady-state values of speed, flow, and density is called the Fundamental Diagram of traffic. This diagram can be viewed in one of three ways: flow vs. density, density vs. speed and flow vs. speed. All three versions represent the same underlying relationship between these variables, since they are also related by the following relationship: flow = density x speed. The three versions of the fundamental diagram for the car following model used in Dynameq are shown in Figure 2.2.

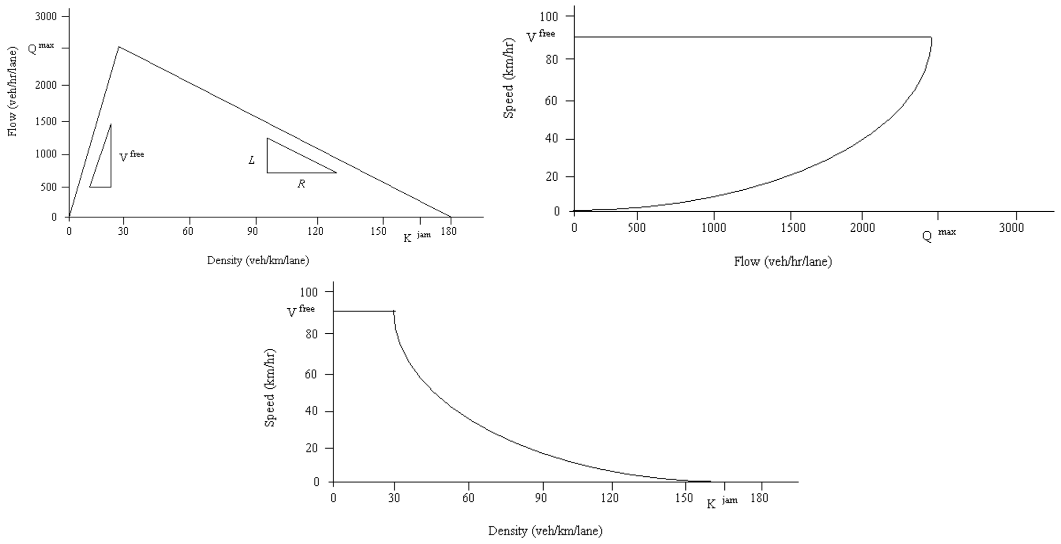


Figure 2.2 : Fundamental diagram.

The diagrams show the following three macroscopic traffic flow parameters:

- Maximum flow (Qmax) is the maximum possible flow rate (expressed in veh/hr, or veh/hr/lane) that a specific link can carry of a specific vehicle type.
- Jam density (Kjam) is the maximum number of vehicles of a specific vehicle type (expressed in veh/km or veh/km/lane) that fit on the roadway when standing still.
- Wave speed (Vwave) is the speed (km/hr) at which shock waves move through a platoon of traffic against the direction of flow, for a specific vehicle type. When a traffic signal turns green, the time between when the first and last vehicles standing still in line begin to move, divided by the distance between them, is equal to the wave speed.

The values of these three parameters for a specific vehicle type and a specific roadway can be determined from the free speed (of the road), and the effective length and response time of the vehicle type, as follows:

$$Q_{max} = \frac{1}{(R + L/V_{free})}$$

$$K_{jam} = \frac{1}{L}$$

$$V_{wave} = \frac{L}{R}$$

The flow vs. density diagram, which shows all three of these parameters, can be interpreted as follows. The level of congestion is represented by density, which increases on the horizontal axis from zero to the jam density. On the left side of the diagram, the value of flow increases with density until reaching the maximum flow. This line describes the steady-state behavior of traffic in noncongested conditions. On the right side of the diagram, the flow decreases with increasing density until reaching a value of zero at the jam density. At this point, traffic is standing still. This line describes the steady-state behavior of traffic in congested conditions. The steady-state speed of the traffic at any point on this diagram is the slope of the line from the origin of the graph to that point. Thus, speed remains constant and equal to the free speed in noncongested conditions. In congested conditions, speed decreases with increasing density, and is equal to zero at the jam density.

Link-Specific Vehicle Type Factors:

Dynameq permits the two vehicle type parameters (effective length and response time) to vary from one link to another by providing a link-specific multiplication factor for each one. These factors are applied to all vehicles while they are on the link during the simulation. This added flexibility can be useful for calibrating the traffic flow properties on each link if necessary, as both of these parameters reflect driver behavior that can change depending on the physical characteristics of a link. For example, if drivers have a tendency to drive closer together on a downhill slope than on an uphill slope, it may be desirable to adjust the response time factor on links with a significant slope in order to capture this behavior.

Gap Acceptance:
A gap-acceptance model is a stochastic model that determines whether a vehicle on a lower-priority movement precedes a vehicle on a conflicting higher-priority movement. A typical example is a permitted left-turn movement that must yield to the opposing through movement at a signal-controlled intersection (in a right-side driving network). Gap acceptance modelling concerns how drivers yield to other vehicles when this behavior is dictated explicitly by traffic signage (e.g., by a yield sign), or implicitly by the rules of the road.

Gap acceptance is modelled in Dynameq based on two quantities, the available gap and the relative wait. The available gap is the amount of time available for the lower priority vehicle to execute its movement and clear the point of conflict with the higher priority vehicle. The relative wait is the difference between the amount of delay that two vehicles have already incurred at an intersection (not including delay due to a red traffic signal), while being the next vehicles to exit by their respective lanes. This delay is the amount of time spent waiting for acceptable gaps on higher-priority movements.

These two quantities are calculated for each pair of conflicting vehicles whenever it must be determined which is to precede the other in executing its movement at a node, or whenever this decision needs to be re-evaluated. These two quantities are based on two values associated with the vehicles in question, called the demand time and the supply time. When a vehicle becomes the next vehicle to exit its lane (that is, the moment at which the vehicle ahead of it exits the lane), the time at which it arrives to the node is calculated. This value is called the demand time of the vehicle at the node, and is denoted as follows:

T_{low}^{dem}	=	the time at which the (front bumper of the) lower-priority vehicle is estimated to arrive at the node, calculated at the moment this vehicle becomes the next to exit its current lane
T_{high}^{dem}	=	the time at which the (front bumper of the) higher-priority vehicle is estimated to arrive at the node, calculated at the moment this vehicle becomes the next to exit its current lane

The supply time of a vehicle is the earliest time that the vehicle may execute its movement while maintaining a margin of safety with respect to preceding vehicles on conflicting movements at the node. Unlike the demand time, this value may be updated several times while a vehicle is the next to exit its lane. This value is denoted as follows:

$T_{low}^{sup}(t)$	=	the earliest time at which the (front bumper of the) lower-priority vehicle may safely arrive at the node, considering all vehicles to have arrived at the node prior to time t
$T_{high}^{sup}(t)$	=	the earliest time at which the (front bumper of the) higher-priority vehicle may safely arrive at the node, considering all vehicles to have arrived at the node prior to time t

The available gap at time t, denoted gap(t), is defined as:

$$gap(t) = (T^{sup}_{high}(t) - R_{high}) - (T^{sup}_{low} - R_{low})$$

Where: R_{high} and R_{low} are the response times of the vehicles on the high and low priority move respectively. The relative wait at time t, denoted $wait(t)$ is defined as:

$$wait(t) = (T^{dem}_{high}(t) - R_{high}) - (T^{dem}_{low}(t) - R_{low})$$

Each of these two quantities is used to determine a probability that the lower-priority vehicle precedes the higher priority vehicle, called the precedence probability. The maximum of the these two probabilities is used to determine which vehicle precedes which. The precedence probability distributions based on the available gap and relative wait are shown in Figure 2.3 and Figure 2.4, below. Both are linear distributions, and each one is based on a single parameter.

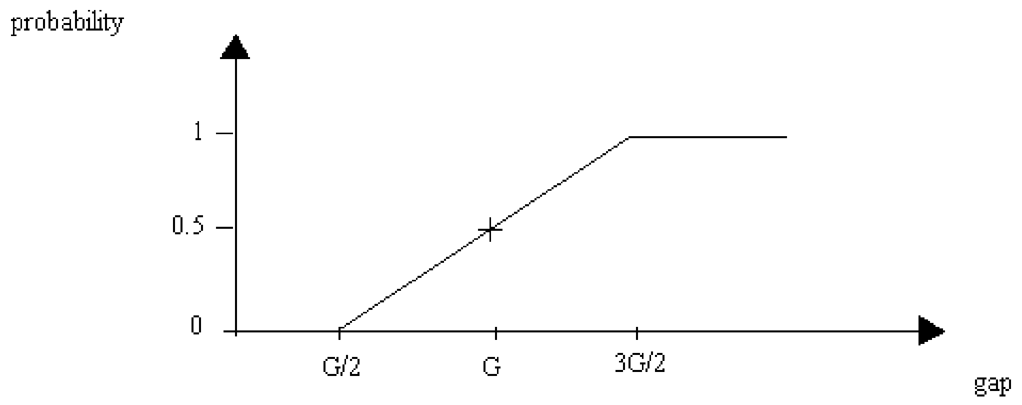


Figure 2.3 : Probability distribution based on the available gap.

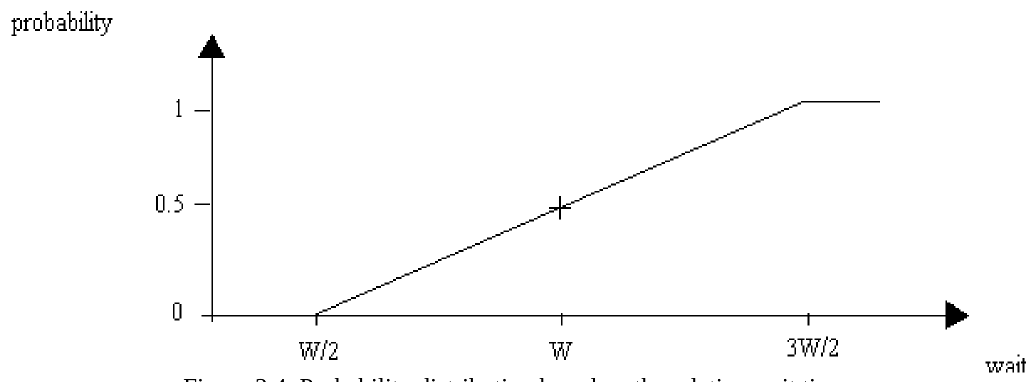


Figure 2.4: Probability distribution based on the relative wait time.

The precedence probability based on the available gap uses a parameter called the critical gap (denoted by G), which is the value of the available gap that has exactly a 50% probability of being accepted. Accepting a gap implies that the lower priority vehicle precedes the higher priority vehicle. As indicated on the graph, the precedence probability is zero for gaps less than G/2 and increases linearly, from zero at G/2 to one at 3G/2. Thus, all gaps greater than 3G/2 are accepted (with probability 1), while the probability of a gap of size G being accepted is 0.5.

The precedence probability based on the relative wait reflects the influence of driver impatience on gap-acceptance behavior. As waiting time increases, the driver may eventually accept a gap that is not normally considered acceptable, and may even oblige the higher priority vehicle to slow down in order to maintain a safe distance (or to avoid a collision).

This probability distribution uses a parameter called the critical wait (denoted by W), which is the value of the relative wait at which there is a 50% probability of the lower-priority vehicle preceding the higher-priority vehicle. The precedence probability is zero for values of relative wait that are less than W/2, and increases linearly from zero to unity over the domain (W/2, 3W/2). This relationship ensures a minimum low priority flow even when the high priority flow is at the maximum flow rate.
The critical gap and critical wait parameters have the following impacts on the relative flow rates of two conflicting movements:

Decreasing the value of critical gap results in more vehicles on the lower priority movement merging with or crossing the higher-priority traffic stream, when this stream is in under-saturated conditions. In saturated traffic conditions, there are essentially no available gaps, and the critical wait parameter determines the amount of flow on the lower-priority movement.
Decreasing the value of critical wait results in more vehicles on the lower-priority movement merging with or crossing the higher-priority traffic stream, when this stream is in saturated conditions.

Lane Selection:
Dynameq models the movement of vehicles on the individual lanes of a roadway. How drivers utilize the lanes on the road can have a significant impact on the delays experienced by drivers and how these delays propagate through the network. Since vehicle trajectories within links are modelled implicitly, each driver must choose the lane by which to exit a link just before entering it. Once on the link, the choice is not re-considered.

The rules used to model the drivers' lane-choice behavior are quite complex, combining a look-ahead procedure with local lane-choice rules. The look-ahead feature captures the behavior of drivers familiar with the roadway and the recurrent congestion patterns along their usual paths.
The look-ahead logic identifies the next critical movement on the path, which is often the next movement for which there is only one lane of flow permitted. The next step is to identify the target lanes on the next link (downstream of the current position of the vehicle), which are the lanes that are best aligned with the permitted lanes for the critical movement. This is done by considering the lane alignments specified (in the control states) for the intervening movements. The local lane choice rules consider the target lanes on the next link, the specified lanes for the movements to be used for entering and exiting the next link, as well as the presence of queuing on the next link, in choosing the lanes by which to enter and exit that link.
A good example of the importance of look-ahead rules is the case of queuing on a highway due to excessive demand for an off-ramp. In most cases, as long as the highway is at least a few lanes wide, it is expected that drivers try to stay to one side of the road and leave a clear channel for vehicles that are continuing past the exit. The Dynameq look-ahead rules identify the off-ramp as the critical movement for drivers leaving the highway there, which results in the lanes that are best aligned with the exit ramp being used by those drivers when they are still upstream of the queue. Drivers continuing beyond the exit ramp have a wider choice of lanes, and choose the more freely flowing lanes when congestion due to the off-ramp begins to set in. The lane choice rules ensure that, no matter how long the queue becomes, drivers destined for the off-ramp target the lanes already queuing, while drivers heading beyond the ramp try to by-pass the queue.

2.1.7 Outputs

Dynameq enables analysts to distill mountains of output data into visual representations of dynamic traffic conditions, from the big picture down to individual lane queues. The outputs of a Dynameq DTA are the simulation results and the path-based results.

Simulation results are presented as animated plots and time-series charts. The animated plots in Dynameq can be customized easily to display average values of traffic measurements, including flows, densities, speeds, travel times, vehicle counts and queues. Queue results show detailed animations of queues by lane. All simulation results can be broken down by class. The size of the time interval for these measurements is user-defined, but typically on the order of a few minutes. A variety of time-series charts can be generated at the link, lane, node and movement level.

Path-based results are presented as time-dependent animations, and change over departure intervals. Path sets and route-choice decisions can be inspected for use in calibration, and provide context for more detailed analyses, such as scattergram plots to compare traffic assignment results with empirical traffic measurements.

The calibration process is streamlined with effective data displays that let you do the following:

- See the big picture with animated network-scale animated plots, densities and travel time results to identify congestion patterns.
- Assess the extent of congestion with animated lane-by-lane queues and time-series plots.
- Focus on the paths used from critical origins to destinations.
- Compare the results of scenarios with side-by-side comparisons.
- Compare predictions and observations with scattergrams and histograms.
- All charts and plots can be exported for further analysis with external tools, and for inclusion in reports.

2.2 Description of Indy

2.2.1 Introduction

Indy is a macroscopic dynamic user equilibrium model which can just like Dynameq be used to evaluate the impacts of congestion relief strategies, such as infrastructure expansions. It can also be used to evaluate the effects of all kinds of dynamic traffic management strategies. Furthermore, it can be used to determine the impacts of incidents and other disruptions. This type of analysis indicates the most critical parts in road networks.

Indy shows on which locations congestion occurs and how the congestion is propagated through the network. The equilibrium approach of Indy DTA produces chosen paths that are consistent with drivers' desire to minimize their travel costs.

In Figure 2.5 the model framework of Indy is depicted. It consists of three main modules:

1. Route generation
2. Route choice
3. Dynamic network loading.

Each of these main modules can contain different kind of models. Due to the modular setup, different combinations of route generation models, route choice models and dynamic network loading models can be made. Two of the three network loading models can take different user classes (driver types and vehicle types) into account making the framework completely multiclass. The link transmission is not yet fully multiclass. Up to now, it can only deal with one type of vehicle class in the network loading phase.

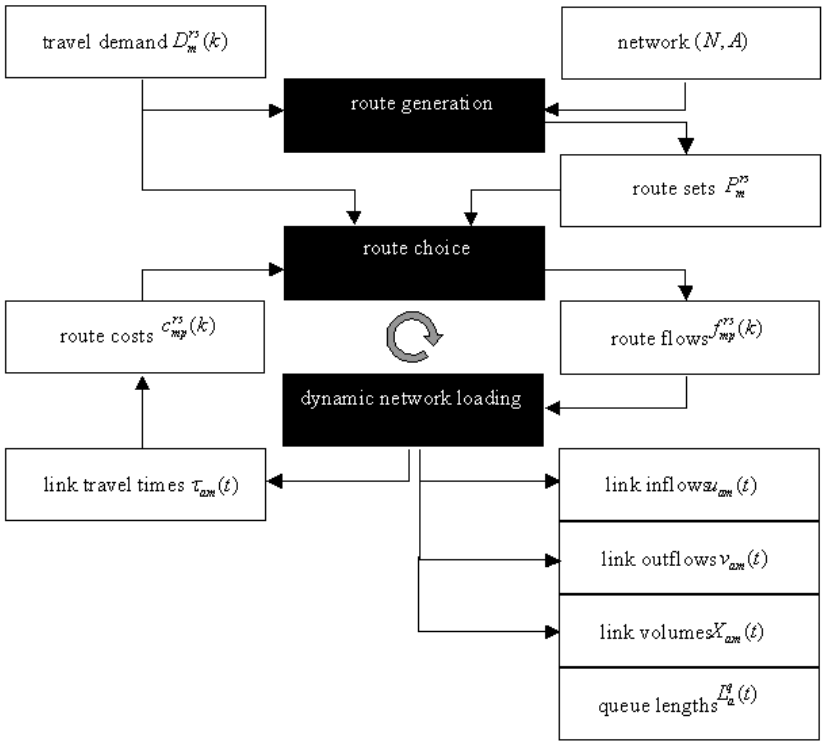


Figure 2.5 : Flow chart Indy.

Firstly, the route generation module determines the routes based on the network characteristics and the travel demand. There are three methods implemented for generating the routes: a Monte Carlo approach, an approach using a static traffic assignment and an approach in which a pre-specified route set is used. The output of the route generation module will be route sets for all vehicle classes describing the available routes between each origin-destination (OD) pair.

Second, the route choice module models the behavior of the travelers by choosing the best route for themselves from the set of available routes as determined in the route generation model. The best alternative route depends on the route costs for each of the alternatives and consists mainly of the route travel time, but can include other (non-additive) cost components such as tolls. The outputs are dynamic route flow rates between each OD pair on the network.

Third, the dynamic network loading module is the heart of the Indy model propagating the traffic along the chosen routes. Outputs are link characteristics, such as link inflows, outflows, volumes, queue lengths and travel times. These link travel times can in their turn be used to compute the route costs. Three different approaches of the dynamic network loading model are described in section 2.2.7. The first model uses link performance functions for computing the link travel times in order to propagate the flow through the network. The second model explicitly assumes hard capacity constraints on link inflows and outflows, leading to a dynamic queuing model. Finally, the third model is the so called lint transmission model.

There is a feedback from the new route costs to the route choice module, leading to new route flow rates and again performing a dynamic network loading. These two modules are performed iteratively until convergence is reached. In the following sections Indy is described in more detail.

2.2.2 Input

The inputs to Indy are the traffic demand, network definition and possibly a pre-specified set of paths with or without initial path departure volumes. The traffic demand is represented by one time-dependent O-D matrix for each class of vehicle being modelled. The representation of the road network is at the link level.

2.2.3 User classes

Indy can deal with different user classes. Most assignment models that call themselves multiclass models can only handle simple user classes. Usually, these multiclass models only distinguish user classes in the route choice level, but not in the dynamic network loading. This means that they are only able to model user classes based on different travel preferences and different route choice behavior, but all user classes behave the same when driving on the network. Therefore, they cannot model cars and trucks separately, as they assume that all vehicle classes have the same speed functions.

Indy considers a wide range of possible user classes. Different driver classes and different vehicle classes are distinguished. Different driver classes have different preferences or have different information available, impacting their route choices. Vehicle classes on the other hand have different traffic flow behavior, impacting the dynamic network loading. Driver classes are usually relatively easy to incorporate into an assignment model. However, distinguishing different vehicle classes is more difficult.

A driver class may have specific:
- travel preferences;
- value of time (depending e.g. on trip purpose and income);
- information available; etc.

These characteristics can be taken into account into specific generalized cost functions. Regarding the information available to drivers, the following driver types are distinguished:
- type I drivers, always taking the same route: drivers with very limited route information, take route choice decisions out of habit or just taking the shortest route as provided by a simple navigation system;
- type II drivers, taking the perceived cheapest/fastest route: drivers with imperfect route information, basing their route choice decisions on their (subjective) experience;
- type III drivers, taking the actual cheapest/fastest route: drivers with perfect route information, basing their route choice decisions on a smart route guidance system.

A vehicle class may have specific:
- maximum speed limits;
- vehicle length;
- dedicated infrastructure available (e.g. bus or truck lane);
- impacts on other traffic on the road; etc.

Vehicle classes can be modeled by assuming a different flow propagation for each vehicle class, e.g. assuming different speeds for each vehicle class. Note that if one would like to model different driver types such as ‘slow drivers’ and ‘fast drivers’, then this typically influences the flow propagation, hence such driver types should be modeled as different vehicle classes. If different driver types with different travel preferences or value of time are to be included, then their generalized route cost function will change, but their travel times are the same. This kind of driver types can be specified as different vehicle classes with different cost functions. The dynamic network loading, however, can be performed for all these driver types together as they drive at the same speed.

It should be pointed out that driver classes and vehicle classes can be used in combination. For example, we can have cars with type II drivers or trucks with type I drivers. The link transmission is not yet fully multiclass. Up to now, it can only deal with one type of vehicle class in the network loading phase.

2.2.4 Traffic generation

The travel demand input $D_m^{rs}(k)$ is assumed to be given for each departure time interval k representing time interval $[(k-1), k\eta),$ where η is the size of each of the departure time periods. This size η is user specified and ranges usually from 5 minutes to 1 hour. The smaller η is, the more travel demand data is needed as input.

In Indy the route choice proportions are computed for each interval k, which means that the route choice proportions are not changed within time period k. Making η smaller makes the route choice problem more dynamic in the sense that route choice decisions are made more often over time. However, very small time periods η (say, less than a minute) can make the model less stable; in each η completely different choice proportions could occur, also making it more difficult to find an equilibrium solution. Therefore, an adequate value for η has to be determined for the application at hand.

The user can choose to specify different demand matrices for each departure time interval or to specify departure fractions for each departure time interval of one single demand matrix. The number of departures within each departure time interval is spread uniformly over time.

2.2.5 Path generation

In the route generation model the route alternatives for drivers are generated. Indy is a route-based model in contrast to link-based models. Link-based models are typically easier and faster to solve than route-based models due to the fact that the number of routes in a network is much larger than the number of links. However, link-based models also assume that the route travel costs are additive, which is not always the case. Furthermore, route-based formulations yield a more intuitive way of modeling the route choice behavior,

Although being more general, a route-based approach can become computationally expensive since the number of possible routes grows exponentially with the network size. Therefore, it is usually not possible to enumerate all possible routes. In order to avoid enumerating many routes, many models search for new shortest routes to add to the route set while running the model. Since we are dealing with a dynamic model, a dynamic shortest path algorithm needs to be performed in those models, which is a complex and time consuming algorithm. In order to avoid this kind of computations while running the model, routes are generated a priori in Indy. This means that routes are generated only once at the beginning of the model run and no shortest path computations are being performed during the rest of the model run.

A priori generation of routes has three important advantages:

- Behaviorally more realistic, i.e. travelers choose from routes they know to exist;
- Speeds up the model run since no shortest path computations need to be performed;
- Faster convergence of the route choice module due to a larger initial route choice set.

A disadvantage of a priori route generation is, that one is never sure that all relevant (used) routes are included in the route set. Therefore, it is important to create a sufficiently large route choice set such that all relevant routes are included. Of course, after running the model it is possible to check whether more routes should have been included or not by running a dynamic shortest path algorithm on the dynamic link travel times.

For each INDY assignment, a set of available routes needs to be defined. For this, three different options are available:

1. Use a Monte Carlo simulation approach to generate routes
2. Use a static traffic assignment to generate routes (and initial route flow distributions)
3. Use an already existing database containing the route set (and initial route flow distributions).

Ad. 1. Monte Carlo route generation:

Considering only on network characteristics (not travel demand), route generation based on Monte Carlo simulation iteratively tries to find new fastest routes based on stochastic link travel times. Each link travel time consists of a constant free-flow travel time and a stochastic part. In each iteration the fastest route is computed for each origin-destination (OD) pair. If it is a new route, it is added to the route set. New random numbers are drawn from a stochastic distribution function (depending on the link length) and added to the free-flow link travel times. This yields new link travel times in each iteration and therefore potentially leading to new fastest routes to be added to the route set. The variance of the stochastic term is increased in each iteration (up to a certain maximum value), yielding an accelerated Monte Carlo approach. The higher the variance, the more likely it is to find a new fastest route.

Such a procedure will generally generate routes that are not much ‘longer’ than the shortest route and will exclude routes with a large detour. The stochastic component can be seen as simulating more or less congestion on each link, although congestion may not occur on all links (the travel demand is not used in this route generation approach).

Within Indy, two parameters can be used to influence the number of routes that will be found:

1. pathIterations: in each iteration the link travel times can change. The larger the number of iterations, the more routes may be generated. When only one iteration is performed, only the fastest routes based on free-flow travel times will be found.
2. pathOverlap: the overlap factor is applied as a filter on the generated routes. If the overlap of a new found route with the other routes (for the same OD pair) already in the route set, then the new found route will not be added to the route set. This avoids the problem of having many almost identical routes in the route set. A common problem is that a route using the off-ramp and the on-ramp of a freeway is added as a new route in the route set, which is usually unwanted (although it is a feasible route alternative). By applying an overlap factor these routes can be excluded from the route set.

Ad. 2. Static assignment route generation:
The static assignment route generation procedure carries out a static equilibrium assignment to define a route set. As a by-product, the distribution of traffic over alternative routes (the route flow proportions) computed by the static assignment can be used to accelerate the Indy assignment. One advantage of this route generation procedure is that by using a static equilibrium assignment, the traffic will be equally divided over the network and all available alternative routes will definitely be found. Furthermore, by using the initial flow distribution computed by the static equilibrium assignment, the traffic will be dispersed over different alternative routes quite effectively already in the first iteration of the dynamic assignment. This may prevent the occurrence of grid locks when using blocking back modeling. For the static assignment, only one user class can be used. And normally, the static assignment will use the O/D-matrix with the largest total demand from the matrices given in the ‘indy.odMatrix’ parameter.

Several options are available that influence the routes that will be found:

1. pathIterations: this defines the maximum number of iterations that may be used for the static assignment. If an equilibrium assignment is found in less iterations, the static assignment procedure will stop before the maximum number of iterations has been reached. The larger the number of iterations, the more routes may be generated. When only one iteration is performed, only the fastest routes based on free-flow travel times will be found.
2. loadingFactorStatic: this defines the factor by which the demand in the O/D-matrix used for the static assignment will be multiplied. In general, more routes will be found when the loading factor is increased.
3. odMatrixStatic: normally, the static assignment will use the O/D-matrix with the largest total demand from the matrices given in the ‘odMatrix’ parameter. With this parameter, a user can enter the PMTU combination specifying the O/D-matrix INDY will use as input for path generation with the static assignment.
4. initialPathFlows: this option defines if the route flow proportions from the static assignment will be used for the first iteration of the dynamic assignment. If set to true, the route flow proportions will be used, if false not.

Ad. 3. Route set database:
The route set can also be defined by a database that contains a predefined route set. This can either be a previously created route set, or a manually adapted route set. It is also possible to use this option in conjunction with the Monte Carlo and static assignment route generation procedures to add additional routes to the route set. Routes available in the database that have not been found by these route generation procedures will then be added to the generated route set.

It is also possible to use an additional database that contains initial route flows and costs. From this information an initial route flow distribution can be obtained that can be used for the first iteration of the dynamic assignment. It must be noted that the paths in this additional route flow/costs database, must coincide with the paths stored in the original route set database. Moreover, if a predefined route flow distribution is used, it is no longer possible to generate additional routes with the route generation procedures.

2.2.6 Path Choice and assignment algorithm

In the route choice module we model the traveler behavior regarding route choice given a route set. We assume that travelers base their route choice decisions on generalized route costs and that each traveler individually aims to minimize this route cost. In fact, the route choice module aims to find an equilibrium state, yielding an iterative process. Different driver types and vehicle classes will be considered.

We assume that drivers face a predefined set of available alternative routes from which they choose the best option. We assume heterogeneity among travelers in our model, in which we distinguish between different driver classes and different vehicle classes. It should be noted that route choice is directly influenced by the driver type and indirectly influenced by different vehicle types due to different route travel times. Indy is capable of mixing vehicle types and mixing different driver types, such as “fixed route drivers”, “drivers taking the perceived cheapest/fastest route”, and “drivers taking the actual cheapest/fastest route”. The route costs consist of route travel times and possible other cost components. Route choice calculation is part of an iterative loop.

An iterative procedure using the method of successive averages is used to solve the route choice problem for a deterministic or stochastic dynamic user equilibrium, or a combination of assignment types according to distinct driver types. The adopted procedure is based on the method of successive averages (MSA). This method computes intermediate route flow rates based on the current actual route travel costs. Then these intermediate route flow rates are averaged with the route flow rates from the previous iteration and uses this as the new route flow rates for the current iteration. These new route flow rates determine new route travel costs (by performing a new dynamic network loading) and is repeated until it converges. As MSA is a heuristic method, convergence is not guaranteed, although it has proven to be a very useful method in many models and usually provides sufficient convergence. Since Indy starts with a predefined route choice set from the first iteration on, it already has a good initial solution such that convergence is fast.

Computation of the actual route travel times:

Computing the actual route travel times from link travel times is easy to formulate in continuous time, but in discrete time we have to make some assumptions that could introduce rounding off errors. The way we propose the discretization here is such that rounding off errors are reduced to a minimum.

Note that time interval t is of size ω, which is the link aggregation period for storing data in the dynamic network loading model. We would like to know the average route travel times for each interval k (which is of size $\eta \geq \omega$). The route travel time when departing during time interval k is computed as an average of the route travel times within time period η by considering the trajectories of the trips leaving each period ω within the time period η, see also Figure 2.6. In this figure, there are four departure times for which a route travel time is computed and then the mean travel time is calculated.

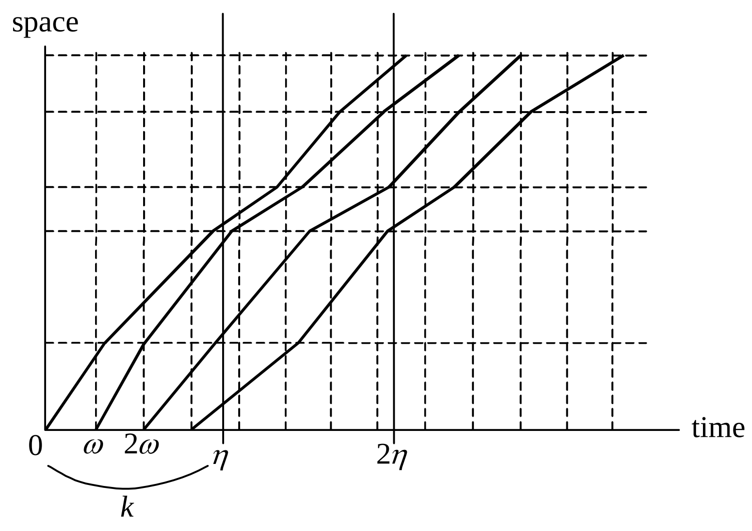


Figure 2.6 : Computing average route travel times.

In order to get the actual route travel time, the computation of each of the route travel times uses a time-discretized version of the following equation:

$$\tau_{mp}^{rs}(k) = \sum_{a \in p} \tau_{am}(\theta_{amp}^{rs}(k)),$$

This equation states how we can add the appropriate link travel times $\tau_{am}(t)$ denoting the travel time on link *a* for vehicles of class *m* that enter the link at time instant *t* with $\theta_{amp}^{rs}(k)$ denoting the time instant at which class *m* vehicles enter link *a* when traveling along route *p* from origin *r* to destination *s* and departing at the origin at time instant *k*. It can be computed as a series in time by

$$\theta_{amp}^{rs}(k) = \begin{cases} k, & \text{if } a \text{ is the first link on path } p, \\ \theta_{a-1,mp}^{rs}(k) + \tau_{a-1,m}\left(\theta_{a-1,mp}^{rs}(k)\right), & \text{otherwise.} \end{cases}$$

The problem is, at which time interval *t* do we evaluate $\tau_{am}(t)$ for the consecutive links on the path? In Indy two time intervals are used to compute a weighted average link travel time $\tau_{am}(t),$ namely the current time interval and the next time interval. For example, suppose that dynamic link travel times are known for each (aggregation) time period of $\omega = 5$ minutes. Furthermore, suppose that we are trying to compute the route travel time when departing at *k* = 0 minutes and that the time entering a certain link a on route p is $\theta_{amp}^{rs}(k) = 7$ minutes, i.e. arriving in the second time period. Then the link travel time that is added to the route travel time is an average of the link travel times of the current and the next time period, weighted according to the proportions in the time period. In other words, the average link travel time will be 60% of the link travel time in the 2nd time period plus 40% of the link travel time in the 3rd time period. By computing and keeping track of the actual time instant $\theta_{amp}^{rs}(k)$ and computing weighted average link travel times, we ensure that rounding off errors are very limited. The time periods t for which the link travel times $\tau_{am}(t)$ are known are usually (much) smaller than the time intervals *k*, such that the route travel times can be computed with a rather good accuracy.

Convergence criterion:

We can use different criteria for terminating the route choice algorithm. In Indy we simply stop after a given number of iterations of the algorithm, I. In order to be able to assess the convergence of the problem, we define two measures:

- The relative dynamic duality gap, $G_1^i(k)$;
- The relative changes in route flow rates, $G_2^i(k)$.

The first measure gives a good indication for convergence when we are looking at drivers that take the actual cheapest route, whereas the second measure may give more insight into convergence when we are dealing with drivers that take their perceived cheapest route. If we are combining both driver types, we can use both convergence measures in combination. It is important to compute both measures for each departure time separately, as otherwise the route choice model may not have converged during certain time periods while an overall measure would indicate convergence. Especially the convergence during the peak periods is important, as here the most changes in route choices will happen. Convergence for off-peak periods is usually easy to reach (perhaps even in one iteration if there is no congestion at all).

For each iteration *i* of the algorithm and for each departure time interval *k* we determine the relative dynamic duality gap as follows:

$$G_1^i(k) = \frac{\sum_{(r,s)}\sum_{m}\sum_{p\in P_m^{rs}} \left(c_{mp}^{rs,(i)}(k) - \pi_m^{rs,(i)}(k)\right) f_{mp}^{rs,(i)}(k)}{\sum_{(r,s)}\sum_{m} \pi_m^{rs,(i)}(k) D_m^{rs}(k)}, \quad i = 1, \ldots, I_c,$$

where $\pi_m^{rs}(k)$ and $C_{mp}^{rs}(k)$ are the minimum and actual costs from origin *r* to destination *s* for mode *m* in departure time interval *k* (for path *p*). $f_{mp}^{rs}(k)$ is the departure flow and $D_m^{rs}(k)$ is the demand. This measure goes to zero when the system converges to a deterministic DUE, since either a route is used ($f_{mp}^{rs}(k) > 0$) and has minimum costs ($c_{mp}^{rs}(k) - \pi_m^{rs}(k) = 0$) or it is unused ($f_{mp}^{rs}(k) = 0$) and can have greater than minimum costs ($c_{mp}^{rs}(k) - \pi_m^{rs}(k) \geq 0$), exactly following Wardrop's equilibrium first principle. In order to be network and travel demand independent, we use a relative measure by dividing the duality gap by total travel costs of the network, normalizing the measure. Note that in case we are looking for a stochastic DUE, i.e. we are modeling drivers that are taking the perceived cheapest route, then the duality gap need not go to zero. It would decrease, but it would stabilize at a certain positive value (depending on the dispersion parameter), because in the stochastic DUE not all travelers take the actually cheapest route.

The relative changes in route flow rates can be captured by the following measure:

$$G_2^i(k) = \frac{\sum_{(r,s)}\sum_{m}\sum_{p\in P_m^{rs}} \left\| f_{mp}^{rs,(i)}(k) - f_{mp}^{rs,(i-1)}(k) \right\|}{\sum_{(r,s)}\sum_{m} D_m^{rs}(k)}, \quad i = 2, \ldots, I_c.$$

This measure indicates how much flow per departure time interval is transferred to other routes in each iteration. Many route choice changes indicate that the model has not converged yet. This measure will be zero if a deterministic or a stochastic DUE is attained. Therefore, it may give additional information on convergence for the stochastic DUE problem. Again, this measure is normalized by dividing it by the total travel demand on the network for the corresponding departure time interval.

2.2.7 Flow propagation

The dynamic network loading (DNL) model is at the heart of the dynamic traffic assignment model. It is basically a simulation/propagation of route flows over the links of the network. Instead of using a micropscopic simulator in which each vehicle is represented individually, the DNL model is macroscopic in which vehicle flow rates are considered.

Three different types are implemented in Indy:

1. DNL model based on link performance functions;
2. DNL model based on dynamic queuing
3. DNL model based on link transmission.

1) link performance functions:
The DNL model based on link performance functions is an extension of the single class DNL model proposed in Chabini (2001) using multiclass dynamic link travel time functions. These link travel time functions predict at the time of link entrance the time it will take vehicles to exit the link. The capacity of the links is only implicitly taken into account in the sense that when there are more vehicles on the link, the link travel time will increase. It does not prevent vehicles from entering the link, hence queues will not be formed explicitly. This first model proposed here is different from other DNL models as proposed by Astarita (1996), Wu et al. (1998), Xu et al. (1999), and Chabini (2001) in the sense that it can handle multiple vehicle classes having different travel characteristics such as different speeds.

The formulation of the analytical DNL model using link performance functions is in fact based on the procedure used in static assignment in which link performance functions are used to compute the travel times. The main difference is now that the link performance functions are dynamic and that the actual position of the flows is calculated at each moment in time in order to base the travel times on the correct amount of vehicles on the links.

The DNL model proposed here is a combination of the model proposed by Chabini (2001) and the model proposed by Bliemer and Bovy (2003) in which the multiclass ideas of the latter are used to extend the single class model of the former. Classes here represent different vehicle types as they may have different flow characteristics. Driver types driving at the same speed can be combined into one vehicle class.

2) dynamic queuing:
The dynamic network loading (DNL) model using link performance functions takes capacity constraints only implicitly into account by increasing travel time functions. However, hard capacity constraints yielding queues and spillback cannot be included.

The DNL model based on dynamic queuing takes capacity constraints of links explicitly into account and determines (horizontal) queues. Instead of predicting the link travel time at the time of link entrance, only the flows are determined on the links based on true (not predicted) traffic conditions. At the end, the link travel times can be derived from the link flows. This model type will predict queues and will be able to deal with spillback. The model has some relationships with the cell-transmission model of Daganzo (1994, 1995), although links do not have to be splitted into cells. It is also capable of dealing with more complex network structures.

Dynamic network loading models based on link performance functions do not include so-called hard capacity constraints. As such, they are not capable of restricting inflow into links, therefore it is difficult to define queues and include important effects such as spillback. The capacity in link performance functions is only included in the link travel time functions. The link travel time will simply increase as the flow increases. However, flows greater than the capacity cannot be ruled out, only the link travel time will go up. The location of the congestion is also not predicted correctly using link performance functions. It will predict queues inside the bottleneck, instead of on the previous links.

Another important aspect that cannot be captured by link performance functions is the fact that capacities can change over time, e.g. due to spillback effects or dynamic traffic management (DTM) measures. Since link performance functions determine the link travel time at the time of link entrance and this link travel time is not adjusted anymore when traversing the link, any changes in capacity cannot be taken into account. This means that the link travel time ‘prediction’ at the time of link entrance may be wrong. Even if the outflow capacity is not sufficient, the flow will leave the link anyway after the link travel time elapses.

Queuing models have been proposed in the literature. Many of them adopt the principle of a moving part and a queuing part on a link. This means that a link is split into two parts, where the queuing part is growing from the head of the link, and the moving part is formed by the remainder (first part) of the link. In our model, we will also use this principle, as it enables us to capture all the important queuing dynamics. Although our approach shares this same principle with other models, it is important to note that the model proposed here differs significantly from other queuing models proposed in the literature. The main difference is the time instant at which the link travel time is determined. This turns out to be the key assumption for dealing with dynamic changing capacities. The moment of determining the link travel times can be:

(a) at the time of link entrance, e.g. He (1997), Ran and Boyce (1996);
(b) at the time of entering the queue, e.g. Roels and Perakis (2004);
(c) at the time of exiting the link (Indy queuing model).

The later this moment of determining the link travel time, the more is known from the past about the true traffic states over time, and the more accurate the link travel time can be determined. In Figure 2.7 the different cases are sketched. The dynamic queue length is visualized by the gray area, and trajectories are plotted for each case. Assuming an inflow of a certain user class m at time instant t, first the vehicles will drive on the moving part, and then they hit the queue and travel at the queuing speed, which depends on the outflow capacity. Further, we assume that at time instant t_2 the outflow capacity drops (e.g. due to spillback or DTM measures). In the queuing model of Indy the link travel times are modeled according to *c*.

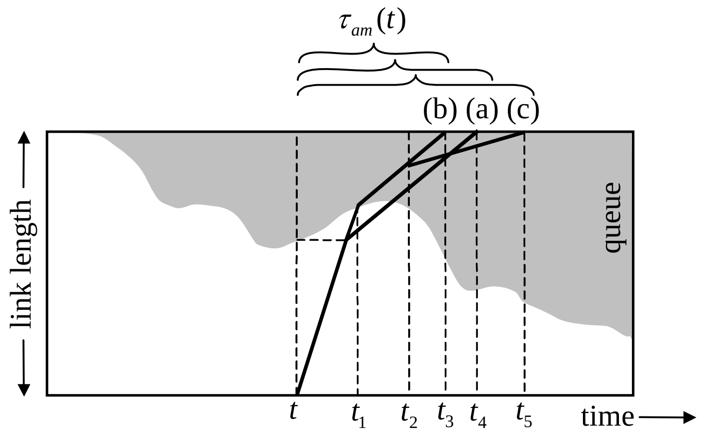


Figure 2.7 : Different queuing models.

3) Link transmission model

The Link Transmission Model (LTM) is the third and newest Dynamic Network Loading (DNL) model that is implemented in Indy. There are two versions of this model. The university of Leuven has an event based model. The version that is implemented in Indy works with fixed time steps. The maximum time step that can in principal be used is equal to the shortest free flow time on all links.

LTM determines time-dependent link volumes, link travel times τ_a and route travel times τ_p in traffic networks, given the time-dependent route flow rates $f_p\ (t)$ for a fixed time period.

Traffic networks consist of homogeneous unidirectional links a, which start at place x_a^0 and end at place xa^L. The links can have any length L_a and they are connected to each other via nodes.

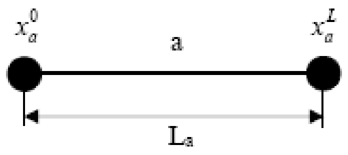


Figure 2.8 : Length and boundaries of link *a*.

A route *p* is a series of links *a* and nodes *n* between an origin node *r* and a destination node *s*. P is the set of all routes *p* on the network. Nodes have no physical length. They act merely as a flow exchange medium.

Figure 2.9 shows some possible node configurations: inhomogeneous node, origin node, destination node, diverge node, merge node and cross node.

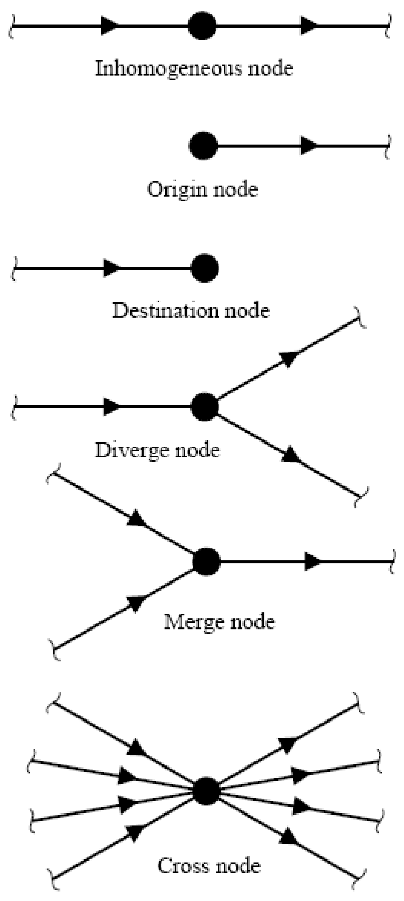


Figure 2.9 : Different node configurations.

A general traffic network can be represented by a combination of links and these basic nodes. Traffic is loaded on to the network in an origin node and it leaves the network in a destination node. An inhomogeneous node can be used to model a change in capacity or in any other characteristic on a road. Diverge nodes and merge nodes are respectively used to model diverging lanes/off-ramps and merging lanes/on-ramps in motorway networks. While the maximum number of links entering and/or leaving a merge or a diverge node is 3, cross nodes connect an arbitrary number of incoming links *i* to an arbitrary number of outgoing links *j*. Cross nodes are used to represent urban intersections.

Cumulative vehicle numbers and link travel times

The cumulative number of vehicles that pass location *x* by time *t* is indicated as *N(x,t)*. Suppose that an observer at location x numbers the vehicles consecutively as they pass him, and he attaches the numbers to the vehicles, then *N(x,t)* represents the number of the last vehicle to pass the observer before time t. LTM primarily determines the cumulative number of vehicles *N(x,t)* that pass locations x_a^0 and x_a^L of each link a by time *t*. Only afterwards, when vehicles have left the link, link volumes and link travel times are derived from these cumulative vehicle numbers, as shown in Figure 2.10.

In this figure, the vertical distance between the curves $N(x_a^0,t_1)$ and $N(x_a^L,t_1)$ represents the number of vehicles on link a at time t_1 (traffic volume). The link travel time τ_a of the hth vehicle on link *a* is represented by the horizontal distance between these curves at height h, if vehicles do not pass each other. The determination of link travel times thus requires first-in-first-out (FIFO) behavior on each network link. LTM ensures this FIFOcondition.

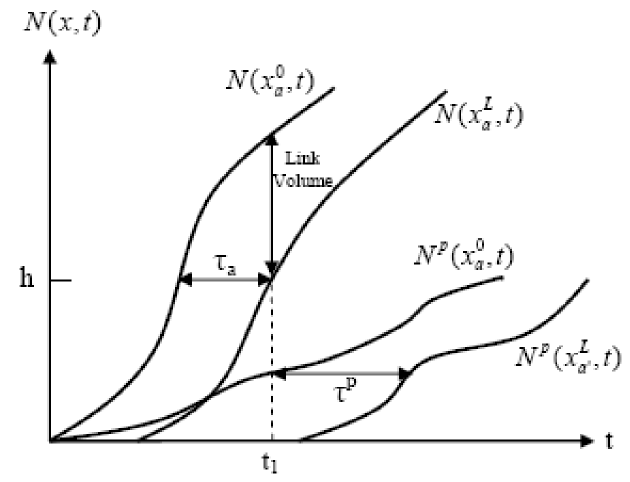


Figure 2.10 : Cumulative vehicle numbers as a function of time.

Multi-commodity traffic and route travel times:
LTM is a multi-commodity (MC) model, where each commodity corresponds to a specific (pre-defined) route. Vehicles are disaggregated by route. We keep track of the routes of the vehicles at all times, when describing the collective motion of the traffic stream. This disaggregation by routes is necessary to use route choice information within the model.
$N^p(x_a^0,t)$ represents the cumulative number of vehicles on route *p*, that pass location x_a^0 by time *t*. The representation in terms of disaggregated cumulative vehicle numbers allows for a simple derivation of route travel times. If origin node *r* and destination node *s* of route *p* are respectively connected to links *a* and *a'*, i.e. if link boundary x_a^0 (x_a^L) borders on node *r (s)*, then the route travel time τ_p of route *p* is represented by the horizontal distance between the curves $N_p(x_a^0,t)$ and $Np(x_a{}'^L,t)$.
Figure 2.10 indicates the travel time of route p for a vehicle departing at time t_1. Since nodes have no physical length, route travel times only consist of link travel times. Times spent on nodes are not taken into account. For all locations *x* and times *t*, the cumulative vehicle number *N(x,t)* is the sum of the cumulative vehicle numbers disaggregated by route:

$$N(x,t) = \sum_{p \in P} N^p(x,t) \qquad \textit{for all } x \in X,\ t \in T$$

Inverse cumulative vehicle function:
The inverse function of the cumulative vehicle number $N_{x-1}(N)$ determines the time $t_x(N)$ at which vehicle number *N* passed location *x*. Since the LTM solution algorithm only calculates cumulative vehicle numbers on discrete time steps $t + m\Delta t$ (where m is an integer), an interpolation procedure might be necessary to calculate $t_x(N)$. As shown in, we propose a linear interpolation procedure.

$$N^p(x,t_x(N)) = N^p(x,t) + \alpha(N^p(x,t+\Delta t) - N^p(x,t)) \qquad \textit{for all } p \in P$$

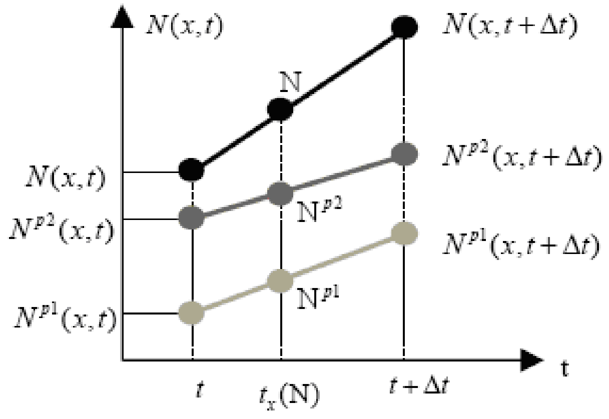


Figure 2.11: Linear interpolation of cumulative vehicle numbers.

Sending flows, Receiving flows and transition flows:
The Sending flow $S_i(t)$ of link *i* at time *t* is defined as the maximum amount of vehicles that could leave the downstream end of this link during time interval $[t, t+\Delta t]$, if this link end would be connected to a traffic reservoir with an infinite capacity. The Receiving flow $R_j(t)$ of link j at time t is defined as the maximum amount of vehicles that could enter the upstream end of this link during time interval $[t, t+\Delta t]$, if a traffic reservoir with an infinite traffic demand would be connected to this link end. Transition flow $G_{ij}(t)$ is defined as the amount of vehicles that are actually transferred from link *i* to link *j* during time interval $[t , t+\Delta t]$. A detailed description of how the sending flows, the receiving flows and the transitions flows per node configuration are computed can be found in Yperman (2007).

2.2.8 Outputs

Once Indy has been run, the output can be visualized by using OmniTRANS. OmniTRANS is the software package in which Indy is run. Standard outputs produced by Indy on a link level are: load, speed, density, inflow, outflow, speed ratio.

Indy is a route-based traffic assignment model which models traffic following predefined routes through the network. As a result of the assignment, Indy is able to produce path results on travel times and toll costs. The path results are produced per O/D pair, path and demand period.

Based on the path results, skim matrices can be automatically created by INDY. A skim matrix describes the impedances between each OD pair and can be defined in terms of distance [km], travel time [minutes], or toll costs. As multiple routes may exist between an origin and a destination, a flow-weighted average is computed.

2.3 Overview of similarities and differences in model setup

In the previous two sections the specifications of Dynameq and Indy are presented. In this section the main characteristics of both models are summarized in Table 2.1. The main differences are in the path choice and network loading. The path choice in Dynameq is deterministic whereas the path choice in Indy is stochastic. This also leads to difference in convergence. The newest and most accurate network loading model in Indy (LTM) works according to Newell's kinematic wave theory and so does the network loading algorithm in Dynameq. However Dynameq models individual vehicles, whereas Indy models aggregate flows. Beside that Dynameq is lane based and Indy is link based. This fact combined with the fact that Dynameq can deal with different types of signalized and unsignalized intersections, allows Dynameq to model lane changing and traffic behavior at intersections in more detail than Indy.

The other, less significant, differences can be found in Table 2.1

Table 2.1 : Overview of characteristics of Dynameq and Indy.

	Dynameq	Indy
Type of model	Microscopic	Macroscopic
Input	- time-dependent O-D matrix for each vehicle class - link: ID, start node, end node, reverse link, type, facility type, length, free speed, jam density (optional), saturation flow (optional), lanes, roundabout, vehicle class permissions - nodes: ID, x-coordinate, y-coordinate, control type, priority template, node type - centroids: ID, x-coordinate, y-coordinate - movements: Node ID, incoming link, outgoing link, free speed, vehicle class permissions, lanes, inlane, outlane, follow-up time - parameters: effective vehicle length and reaction time for each vehicle class, number of paths, number of demand intervals, start time, end time number of iterations, maximum duality gap, aggregation time step	- time-dependent O-D matrix for each vehicle class - link: ID, start node, end node, direction, type, length, free speed, critical speed, saturation flow, lanes, - nodes: ID, x-coordinate, y-coordinate, - centroids: ID, x-coordinate, y-coordinate - movements: (not available) - parameters: jam density, simulation time step, aggregation time step, type of path generation, type of network loading, duration of simulation, number of iterations,
Traffic generation	3 types: - Poisson -Conditional - Constant	2 types - Constant per demand interval matrix - User specified departure fractions
Path generation	Shortest path during the first prespecified iterations of the assignment	3 types: - paths used in static assignments - Monte Carlo simulation - User defined path table (optional: with departure flows)
Path Choice and assignment algorithm	Fastest path combined with regular MSA or flow Balancing MSA (deterministic)	Logit combined with MSA (stochastic)
Flow propagation	Lane based Per vehicle - car following	Link based 3 types: - point queue (multi vehicle class)

	- gap acceptance - lane changing	- dynamic (multi vehicle class ?) - link transmission (single vehicle class)
Node model	Detailed lane/movement based intersection model for different types of signalized and unsignalized intersections. User can define green times or priority flows.	Node model for divergence nodes, merge nodes and urban cross nodes. Priorities based on transition flows between links - No explicit model for signalized (can be approximated by applying traffic controls, model can be extended with average node capacity for turning movements based on effective green time and cycle length) and prioritized unsignalized intersections - No intersection delays (model can be extended with point queues to implicitly realize average intersection delays)
Outputs	Per time step (and per link, node, movement and lane): demand, inflow, outflow, number of cars waiting, number of cars travelling, density, travel time, vehicle kilometres travelled, vehicle hours of delay, speed, occupancy, lane changes.	Per time step (and per link and path): demand, inflow, outflow, load, density, total travel time, total vehicle kilometres travelled, total vehicle hours of delay, speed.
Gridlock avoidance	a deadlock prevention algorithm is implemented that identifies cycles of links that are very close to locking up (during the simulation itself), and manages the inflows to these cycles, much like an adaptive traffic control system, in order to prevent gridlock and maintain traffic flow.	Allow higher outflow
Simulation time step	Event based	User selected time step (version KU Leuven: event based)

3 Case studies

In the previous chapter the main similarities and differences between Dynameq and Indy are shown. In this chapter three cases are presented that show what the influence is of these differences on the model outcomes.

3.1 Case setup

This section describes the setup of three test cases that were used for comparing Dynameq with Indy. In order to overcome the most important difference the cases were setup in such a way that the model outcomes can be compared without having to consider difference in parameter settings. This implies the following:

- Since Indy uses one fixed jam density for each link, this jam density is applied to all the links in the network. This jam density together with the capacities of the links (saturation flow) and the free flow speed determines the basic diagram that is used both in Dynameq and Indy on each link. By using these macroscopic parameters the microscopic parameters (the effective vehicle length and the response time) are implicitly specified because they relate directly to the parameters of the basic diagram.
- The path generation and path choice differs between the two models. In order to overcome this difference the model Dynameq is run and the path results (including the departure flows) are exported to Indy. Indy can directly use these results in the network loading phase. Besides this Indy is also run with its own path generation and path choice algorithm to be able to compare the equilibriums to which both models converge and the speed (computation time and number of iterations needed) at which this occurs.
- In Dynameq the intersections are modelled in much more detail than in Indy. To overcome this difference all intersections are modelled as unsignalized intersections with a general priority scheme. Besides this Dynameq is also run with signals to show the impact of this.
- The models are run with a single user class.
- Both models are run on the same PC: Dell Precision PWS370 Intel Pentium 4 CPU 3.4 GHz, 2 GB Ram, 232 GB hard disk

Test version 1.4.5 of Dynameq is run with the following DTA-settings:

- Traffic Generator: constant
- MSA method: flow balancing
- MSA reset: 3
- Dynamic path search: not selected
- Path pruning: 0.001
- Paths: 10
- Path iterations: 30

Version 1.02.03 of Indy is run with the following DTA-settings:

- Traffic Generator: constant
- Path generation: MONTECARLO with 10 path iterations or Dynameq paths
- Network loading/Blocking back: PHYSICAL = LTM

(For all other parameters the default settings are used)

Several convergence algorithms were written to convert the Dynameq input and output to Indy input and output and the other way around:

1. Export2Dynameq: a syntax to export a network and the matrices of Indy to an ascii-file format that can be imported in Dynameq.
2. ImportDynameqNetwork: a syntax that imports the centroids, nodes, link (and connector) capacities, lanes and the jam density from Dynameq into Indy.
3. ImportDynameqMatrix: a syntax that imports the link based results of Dynameq into Indy.
4. ImportDynameqPaths: a syntax that imports the paths and path flows from Dynameq into Indy.
5. ImportDynameqResults: a syntax that imports Dynameq results into OmniTRANS such that they can be compared with Indy results.

6. ImportLength: a syntax that imports the link lengths of a Dynameq network into Indy for the links for which the length is not included in the original network file of Dynameq.

In appendix A the conversion algorithms can be found. The algorithms could probably be made a more efficient, but efficiency of these algorithms is not very important since they computation time of most of them (expect importing the path results) is very short. The algorithms are included in the appendix in order to make sure that new conversions can easily be carried out the future if necessary (not for reading purposes).

3.2 Test network with two paths (test case 1)

3.2.1 Description scenarios, test case 1

The first test case aims to compare the network loading algorithms on a network without delays on intersections. The table below shows the network characteristics.

Table 3.1 : network characteristics of scenario 1 and 2, test case 1

	Characteristics
Centroids	2
Nodes	3
Links	10
Demand	4400 pcu/h 7.00-8.00 h

In total two scenarios were used. The networks of both scenarios are shown in Figure 3.1 and Figure 3.2 respectively. In these networks there are two paths from origin 1 to destination 2. The demand is 4400 which equals the total capacity of both paths.

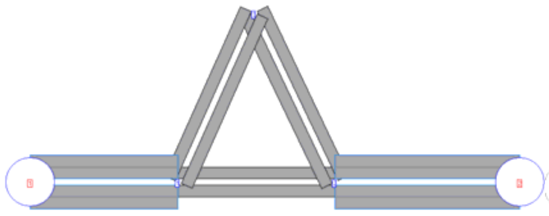


Figure 3.1 : network of scenario 1, test case 1.

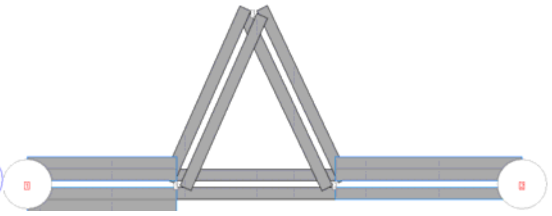


Figure 3.2 : network of scenario 2, test case 1.

In Table 3.2 the link characteristics are shown. In the second scenario a bottleneck is created at the last link before centroïde 2. In this way congestion will occur which makes a comparison of spill back effects possible.

Table 3.2 : link characteristics of scenario 1 and 2, test case 1 (- = centroid)

Start Node	End Node	Free speed (km/h)	Capacity (pcu/lane/h)	Lanes	Length	jam density (pcu/lane/km)
1	2	100	2200	1	10	150
2	1	100	2200	1	10	150
2	3	100	2200	1	5	150
3	2	100	2200	1	5	150
1	3	100	2200	1	5	150
3	1	100	2200	1	5	150
-2	2	100	2200	2	10	150
(scenario 1) 2	-2	100	2200	2	10	150
(scenario 2) 2	-2	100	2000	1	10	150
-1	1	100	2200	2	10	150
1	-1	100	2200	2	10	150

3.2.2 Results scenarios, test case1

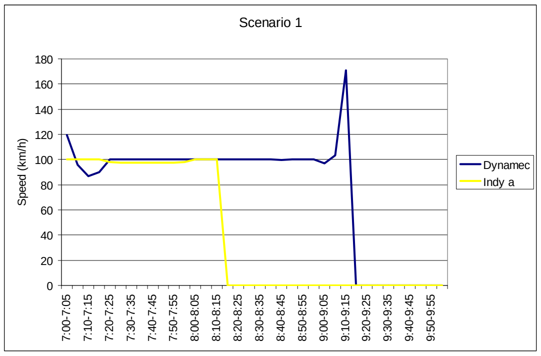
Scenario 1

Table 3.3 describes the total vehicle hours travelled, the total vehicle kilometres travelled and the average speed in the network. From this table it can be concluded that the aggregate numbers are exactly the same. In both models all traffic flow under free-flow conditions.

Table 3.3 : Network results, scenario1, test case 1

	Scenario1	
	Dynameq	Indy a
Vehicle hours travelled	1320	1320
Vehicle km travelled	132000	132000
Average speed	100.0	100.0

In Figure 3.3 the average speed and density are shown over time. These two figures show clear differences between the models. Furthermore, the absolute differences in density are visualized on the network at 7.30h and 8.30h in Figure 3.4 (the with of the links is the absolute difference). Gray means the same density, yellow means that Indy has a higher density and blue means that Dynameq has a higher density. The demand period last from 7.00-8.00 h. Under free-flow conditions the network is expected to be empty shortly thereafter as happens in Indy. However, in Dyanamec it takes much longer. The explanation for this is that in this particular case setup Dynameq is not able to find a second path. In the first iteration one of the two paths is chosen. On this path, no congestion occurs. Due to the look a head feature only one lane of the first link is used. Virtually congestion occurs before the traffic enters the network. That implies that half of the traffic volume has to wait outside the network. Because of the fact that no congestion occurs on the first path, the second path is never found to be faster. This explains why it takes twice as long to let all the traffic flow over the network. In larger networks, this phenomena is not likely to occur because links are shared by multiple paths of multiple OD-pairs which results in changes in travel times on the links. Nonetheless, it shows that stochastic path choice can have advantages over deterministic path choice, since in the case of stochastic path choice the traffic is spread over all the pre-specified paths.



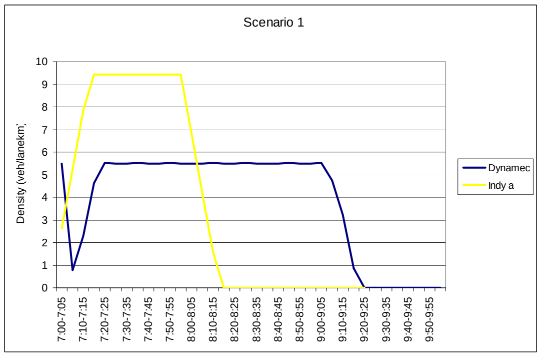


Figure 3.3 : Average speed (left) and density per lane (right) in the network in scenario 1, test case 1.

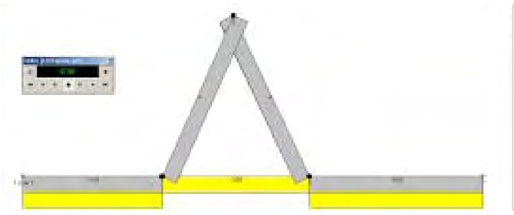
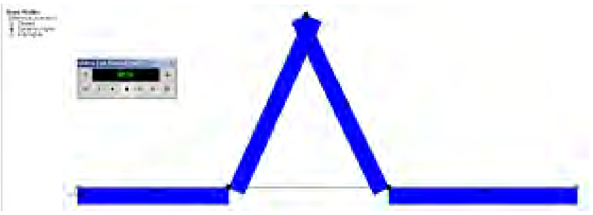



Figure 3.4 : Difference in density in scenario 1 at 7.30h (left) and 8.30h(right), test case 1.

Scenario 2

Scenario 2 is setup in such a way that Dynameq does find two paths. This is done by decreasing the capacity of the last link to one lane with a capacity that is slightly lower (2000 veh/hour) then the lane capacity of the other links (2200 veh/h). In Table 3.4 the aggregate network indicators are shown. 'Indy a' is the case in which Indy is run with its own path generation and path choice and 'Indy b is the case in which Indy is run with the path proportions of Dynameq.

The Total travel times in Indy are for the comparison with Dynameq computed based on the link results (inflow * link length/speed) in order to be able to get an approximation of the total travel time of all the vehicles in the network in a certain time interval. This deviates slightly from the actual travel times which can be computed based on the path flow results. This would result in a total travel of 4007 hours in 'Indy a' and 3938 hours in 'Indy b'. Normally the travel times in Indy are only computed on path results.

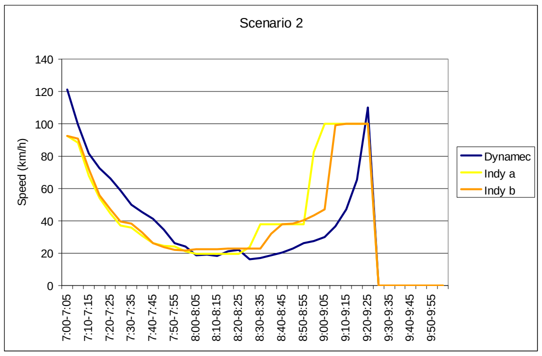
From this table it can be concluded that Dynameq and Indy produce very similar results on the aggregate level. The explanation for the fact that the total travel times for 'Indy a' and 'Indy b' are different is that a different equilibrium is found. In the case where Indy is run with its own paths and path choice an equilibrium occurs in which exactly 50% of the travelers take the first route and also exactly 50% take the second route. In Dynameq (and therefore in 'Indy b' as well) the traffic is split over both path in a 53%-47% proportion (caused by the deterministic instead of stochastic route choice).

The 47% travelers experience a gain in travel time of 5.4 minutes because of the fact that there are less cars on their path compared to the 50%-50% spread of traffic. The 53% experience an extra delay of only 3.0 minutes due to the extra traffic on their path. In total this results in a lower total travel time[1]. In a perfect equilibrium some cars on the '53%-paht' would shift to the '47%-path' because that reduces their travel time.

Table 3.4 : Network results, scenario1, test case 1

	Scenario2		
	Dynameq	Indy a	Indy b
Vehicle hours travelled	3937	3962 (4007)	3903(3938)
Vehicle km travelled	132000	132000	132000
Average speed	33.5	33.3	33.8

In Figure 3.5 the average speed and the density are shown over time. From these figures it can be concluded that the speed and density patterns are more or less the same in Dynameq and Indy. The average speed goes down a bit faster in Indy due to the congestion and goes up a bit faster after the congestion as well. The differences in average speed are explained by the way in which they are calculated (total travel time per time interval/total vehicle kilometers per time interval). As is explained above, the travel time in a time interval are approximated in Indy. However, these differences have no significant meaning, because it is only a difference in post-processing of the model results.



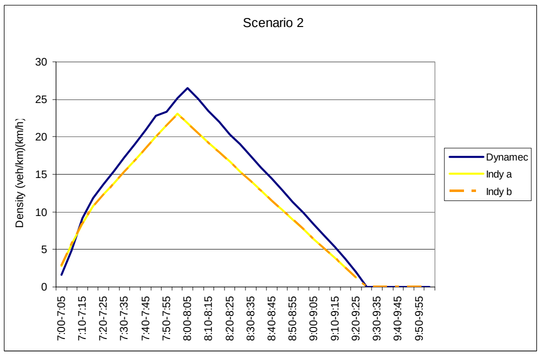


Figure 3.5 : Average speed (left) and density per lane (right) in the network in scenario 2, test case 1.

Furthermore, Figure 3.5 shows that the density is slightly lower in Indy. The reason for this difference is not completely clear. In Figure 3.6 the difference in density on the network is shown at 7.30 h. However, this pattern changes over time.

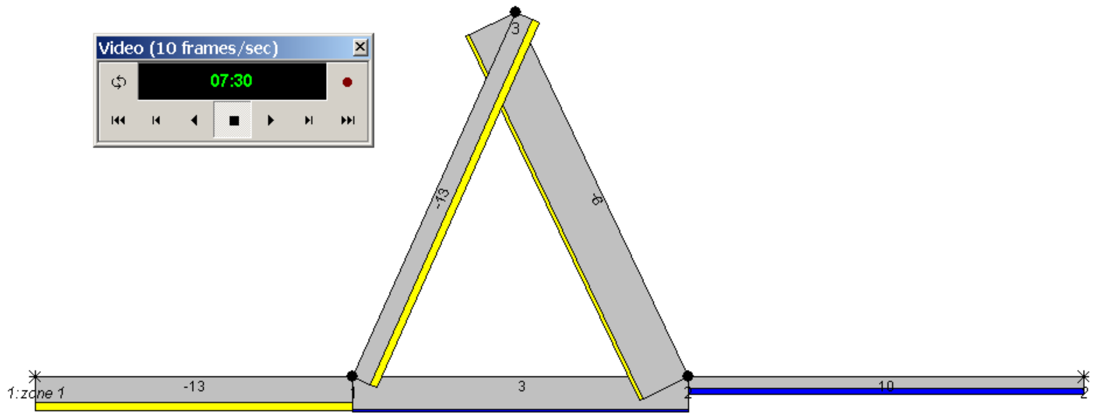


Figure 3.6 : Difference in density in scenario 2 at 7.30 h, test case 1.

Finally, the duality gaps and computation times are shown in Figure 3.7. The results of Indy are the results in which the path generation and path choice of Indy are used. The figure on the left shows that that Indy converges to a true equilibrium in the particular case whereas Dynameq doesn't find that equilibrium in the first 30 iterations. Furthermore, Indy converges in less iterations than Dynameq. There are two reasons for this. The first is that in the first iterations not all the paths are generated and the traffic is not spread over all paths. In this case there are only two paths, so this only concerns the first two iterations. The second reason is that Indy spreads the flow over all available paths because of the logit path choice, whereas Dynameq uses a deterministic path choice.

The computation times in Indy are exported in whole seconds. Since in this network the computation time per iteration is less than one second, the exact computation times per iteration are unknown and therefore computed by dividing the complete computation time by the number of iterations. Dynameq is an event based model, whereas the current implemented version of the link transmission model works with time steps. The University of Leuven has a version of LTM that is event based as well. This implies that, in the version of Indy that is used for the comparison, the time step has to be chosen. This time step has to be chosen smaller than the shortest link free flow travel time. Since the links in this network are very large the time step can be chosen large as well. In this case we used a time step of 60 seconds whereas the shortest link travel time is 180 seconds. This implies that the computation time could have been more reduced by choosing a larger time step. On the other hand, if the links, or even only one of the links, would have been shorter, a smaller time step had to be chosen which would increase the computation time. In Indy there is an option to virtually extend the link lengths in such a way that the time step can be chosen larger. Of course, this does effect the outcomes to some extend. The figure on the right shows that with the time step of 60 seconds Indy is faster then Dynameq. This is probably caused by the fact that Indy is a Macroscopic model and, therefore, doesn't have to keep track of individual vehicles and the way in which they behave.

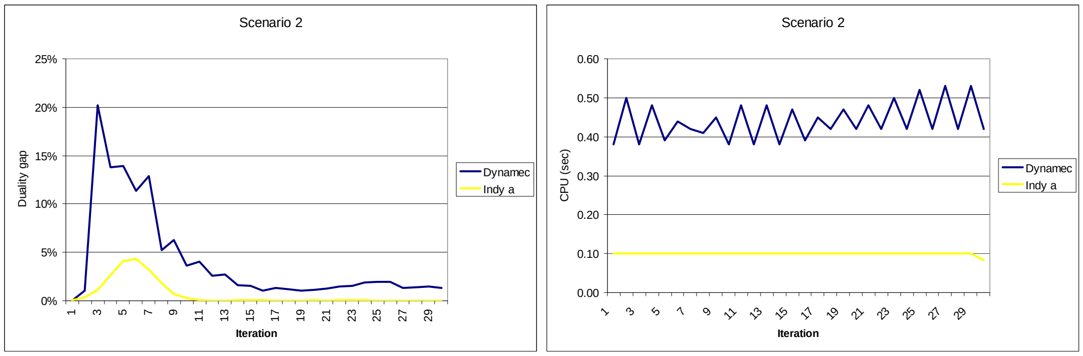


Figure 3.7 : Duality gap (left) and computation time in scenario 2 (right), test case 1.

3.3 Test network for an intersection (test case 2)

3.3.1 Description scenarios, test case 2

The second test case aims to compare the network loading algorithms on a network with an intersection. The case shows how delays at intersections (caused by minimum gaps that are needed to cross an intersection, priority rules or traffic signals) influence the model outcomes. In Dynameq these delays are explicitly modelled and in Indy they are not.

The network that is used for the first two scenarios in this case is shown in Figure 3.8. The figure next to it shows the intersection in more detail. Table 3.5 and Table 3.6 show the characteristics of the network and the links. The demand pattern is shown in Table 3.7.

Table 3.5 : network characteristics of scenario 1 and 2, test case 2.

	Characteristics
Centroids	4
Nodes	5
Links	16
Demand	9000 pcu/h 7.00-8.00 h

Table 3.6 : Link characteristics of scenario 1 and 2, test case 2.

Start Node	End Node	Free speed (km/h)	Capacity (pcu/lane/h)	Lanes	jam density (pcu/lane/km)
8	12	50	2118	1	160
12	8	50	2118	1	160
9	12	50	2118	1	160
12	9	50	2118	1	160
10	12	50	2118	1	160
12	10	50	2118	1	160
11	12	50	2118	1	160
12	11	50	2118	1	160
-1	8	50	2118	1	160
8	-1	50	2118	1	160
-2	9	50	2118	1	160
9	-2	50	2118	1	160
-4	10	50	2118	1	160
10	-4	50	2118	1	160
-3	11	50	2118	1	160
11	-3	50	2118	1	160

(- = centroid)

Table 3.7 : network characteristics of scenario 1 and 2, test case 1.

	1	2	3	4	total
1	0	2000	250	250	2500
2	2000	0	250	250	2500
3	250	250	0	1500	2000
4	250	250	1500	0	2000
Total	2500	2500	2000	2000	9000

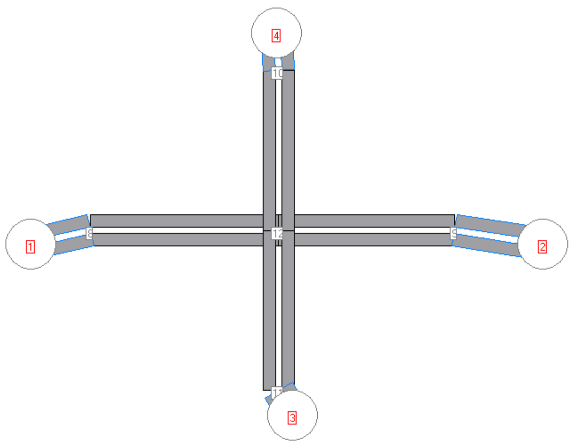


Figure 3.8 : network scenario 1 and 2, test case 2.

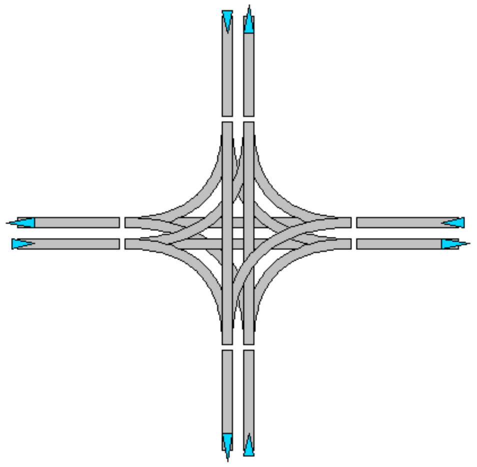


Figure 3.9 : intersection, test case 2.

In total three scenarios are used. In the first an unsignalized intersection is used. In the second a signalized intersection is used in which all directions (left, through and right) have green at the same time. The green phases and green times are shown in Figure 3.10. In Table 3.8 shows the green, yellow and red times.

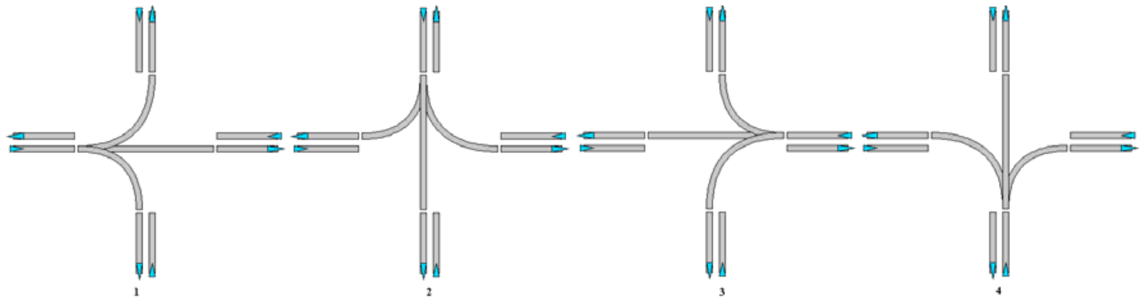


Figure 3.10 : signal phases scenario 2, test case 2.

Table 3.8 : Green, yellow and red times scenario 2, test case 2

	phase (total cycle time : 150 sec)			
	1	2	3	4
green (s)	35	30	35	30
yellow (s)	3.5	3.5	3.5	3.5
red (s)	1.5	1.5	1.5	1.5

As is explained before, Indy doesn’t yet explicitly model intersections. However, there are ways to approach the average behavior at intersections. In Indy an option is included to use outflow capacity events. For each link the outflow capacity can be restricted for a specified time. This means that we can do a simulation by using the actual green times. That is set the outflow capacity to 0 in case of an orange or red signal and set the maximum outflow capacity to the maximum capacity in case of a green signal. However, we can just as well adjust the average maximum outflow capacity to: capacity*(green time)/(cycle length). This is what we chose to do. In Dynameq an extra delay of one time the response time (1.25 seconds) is included at each green phase. Therefore this delay is also used in the computation of the average outflow capacity. By using this method the following maximum outflow capacities are used for the link from centroïd 1 to node 12 and from centroïd 2 to node 12: 477 veh/hour. This is computed as 33.75 (35 sec green time – 1.25 sec response time) divided by the cycle length of 150 seconds times the capacity of 2118 veh/hour. A similar computation resulted in a maximum average outflow capacity of 406 veh/hour for the link from centroïd 3 to node 12 and from centroïd 4 to node 12.

In the third scenario the phases have been adjusted as is shown in Figure 3.11. In this scenario the through traffic has a relatively longer green time and the traffic that makes a left turn doesn't have green at the same time as the through traffic and the right turning traffic. The green times are shown in Table 3.9. In this scenario we introduced separate lanes for the different directions just before the traffic signals. This required an adjustment of the network. The network that is used is shown in Figure 3.11.

Table 3.9 : Green, yellow and red times scenario 3, test case 2

	phase (total cycle time : 75 sec)			
	1	2	3	4
green (s)	25	20	5	5
yellow (s)	3.5	3.5	3.5	3.5
red (s)	1.5	1.5	1.5	1.5

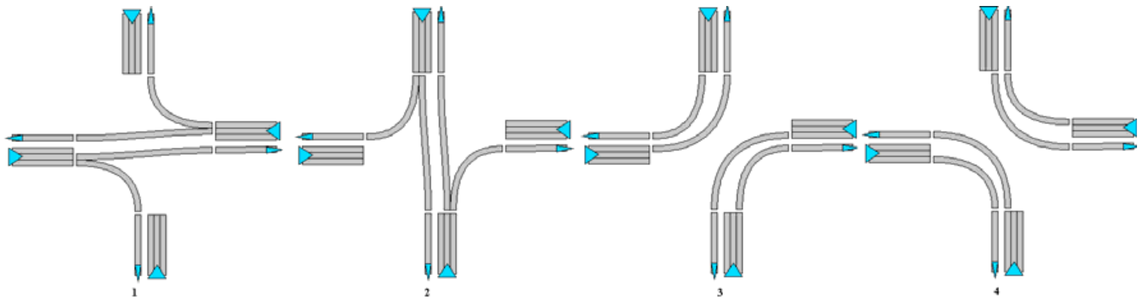


Figure 3.11 : signal phases scenario 3, test case 2.

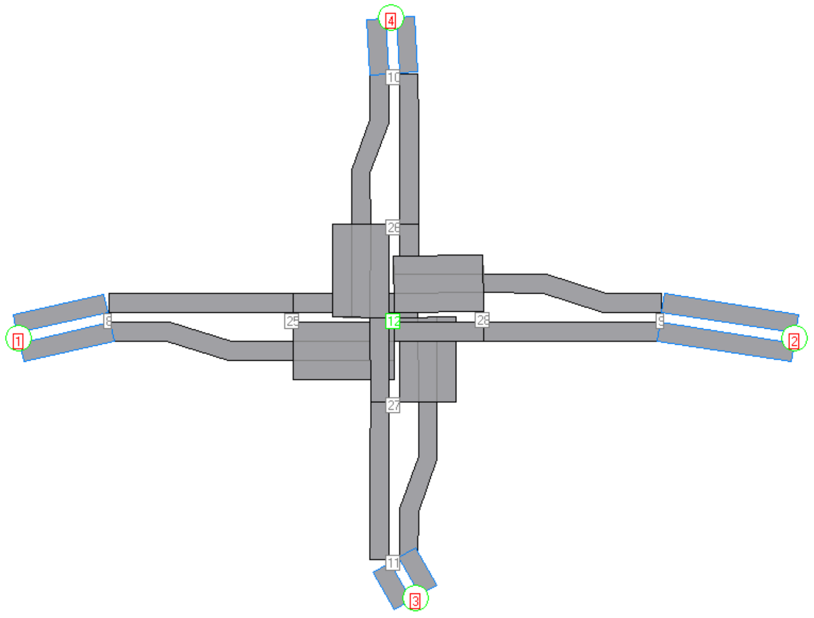


Figure 3.12 : network scenario 3 Dynameq, test case 2.

As in scenario 2, the signalized intersection has to be simulated in Indy by setting the outflow capacities. However, these maximum outflow capacities can currently only be set for a complete link and not for each turning movement. This is why the network Indy had to be adjusted to the network as is shown in Figure 3.13. Each movement has its own link with a matching average outflow capacity. The maximum average outflow capacities are shown in Table 3.10. They are computed in a similar way as in scenario 2.

Table 3.10 : average maximum outflow capacity Indy, test case 2

orig	dest	total green time	Avg. Outflowcap
1	2	23.75	671
1	3	3.75	106
1	4	27.5	777
2	1	23.75	671
2	3	27.5	777
2	4	3.75	106
3	1	22.5	635
3	2	3.75	106
3	4	18.75	530
4	1	3.75	106
4	2	22.5	635
4	3	18.75	530

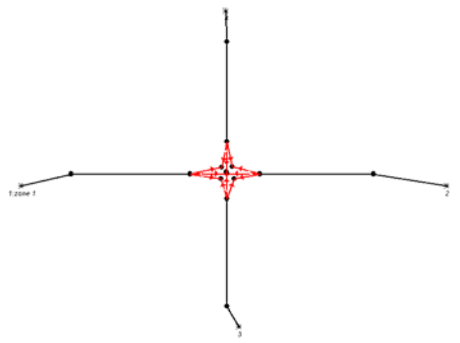


Figure 3.13 : network scenario 3 Indy, test case 2.

3.3.2 Results scenarios, test case2

The average speed, vehicle hours travelled, vehicle hours travelled for scenario 1, 2 and 3 are shown Table 3.11. There is only one path for each OD-pair. Therefore, path generation and choice don't play role. The results in this are quite misleading because of the fact the vehicles that can't enter the network wait outside the network in Dynameq. This waiting time is not included in the total travel time. In Indy the vehicles wait on the connector is a vertical queue. This waiting time is included in the total travel time. Nonetheless, the table does show that the total travel time in Indy is about twice as low as Dynameq in scenario 1. This is the scenario without traffic signals. The explanation for this is that in Indy doesn't consider delays at intersections that are caused by waiting that gaps are needed to cross an intersection. In Indy cars can virtually drive over each other, which is of course not realistic. The fact that the travel times are twice as low shows that the delays at intersections can add up to a substantial amount. On the other hand, the flows in this network are high. If there is less traffic in the network, the delays at the intersections also reduce.

Table 3.11 : Network results, scenario1,2 and 3, test case 2.

	Scenario1		Scenario2		Scenario3	
	Dynameq	Indy	Dynameq	Indy	Dynameq	Indy
Vehicle hours travelled	3065	1715	7911	19176	4047	9539
Vehicle km travelled	54000	54000	54000	54000	54000	54000
Average speed	17.6	31.5	6.8	2.8	13.3	5.7

The average speed and density over time in scenario 1 is shown in Figure 3.14. It can be seen that in scenario 1 the network is one hour earlier empty in Indy than in Dynameq. The difference between Dynameq and Indy are emphasized in Figure 3.15. This figure clearly shows that the outflow in Indy is much higher in the unsignalized scenario than in Dynameq and the density is much lower. The explanation for this is given above.

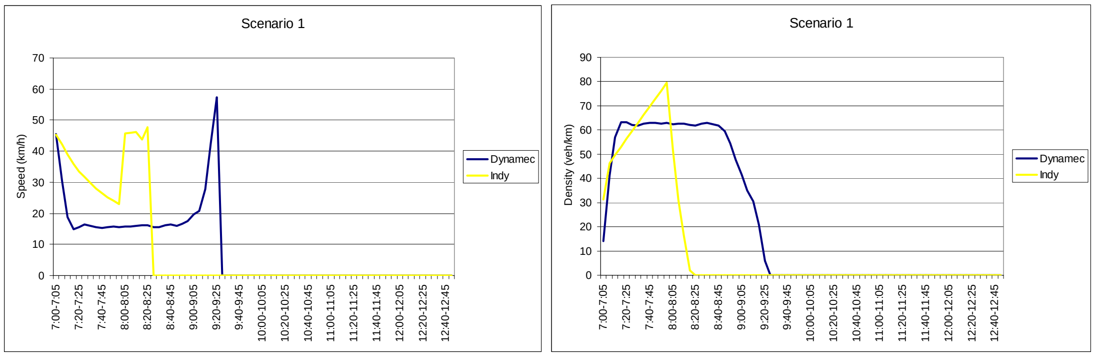


Figure 3.14 : Average speed (left) and density per lane (right) in the network in scenario 1, test case 2.

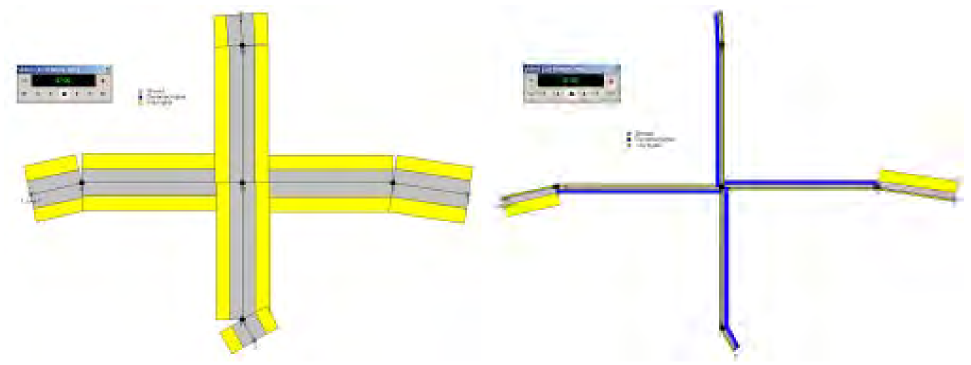
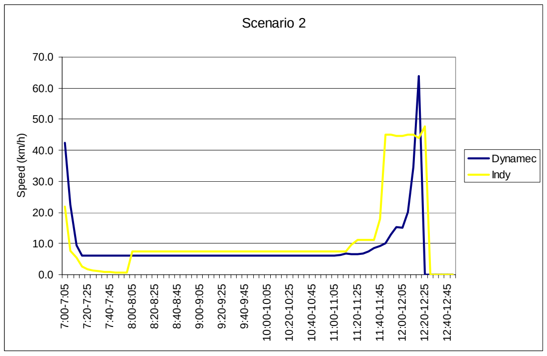


Figure 3.15 : Absolute differences in outflow (left) and density (right) in scenario 1 at 7.30 h, test case 2.

The figures of scenario 2 and 3 show that the network is empty at the same time in both models. Furthermore, the average speeds in the network per time step are quite identical as well. In the densities there are large differences, but as explained before this is caused by the waiting times to enter network. Therefore, these figures suggest that in the case of signalized intersections (scenario 2 and 3) both models produce the same results. Figure 3.18 makes this even more clear for scenario 2. On all ‘normal links’ the density and the outflow are very close to equal[2] and only on the connectors the density differs. A similar plot could not be made for scenario 3, because the network that is used in both models differs. However, the table with link results of both models indicates that both models produce exactly the same results. From this, it can be concluded that it is possible to model the delays at signalized intersections with the macroscopic model Indy by using outflow capacity restraints in such a way that the results of Dynameq are reproduced exactly. This is however only possible in the situation in which there are no conflicts in lane usage. For instance, in the situation where there are two lanes for through traffic and the right lane is also used for right turning traffic, the outflow capacity depends on arrival rate of traffic that turns right and goes through. For those situations an approximate outflow capacity has to be found.

[2] The outflow differs three vehicles per hour. This is caused by rounding off errors in the imposed outflow restriction.



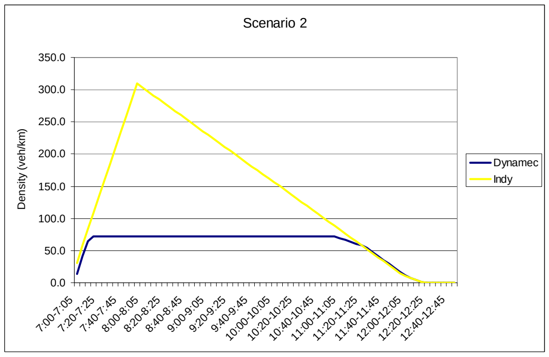
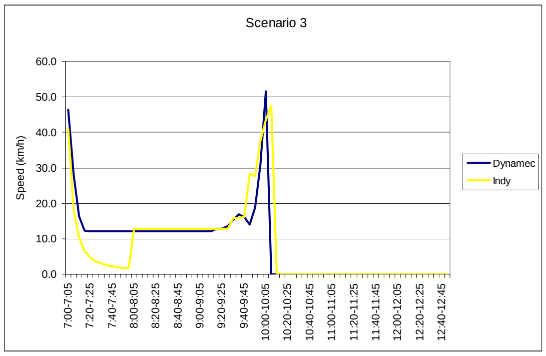


Figure 3.16 : Average speed (left) and density per lane (right) in the network in scenario 2, test case 2.



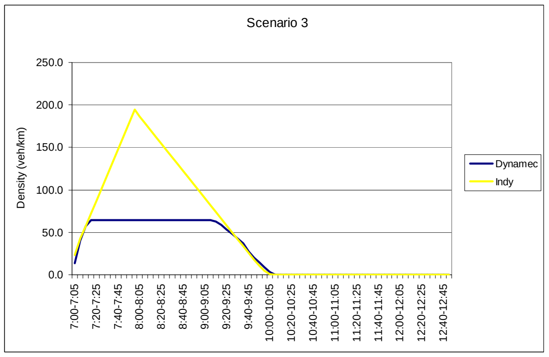


Figure 3.17 : Average speed (left) and density per lane (right) in the network in scenario 3, test case 2.

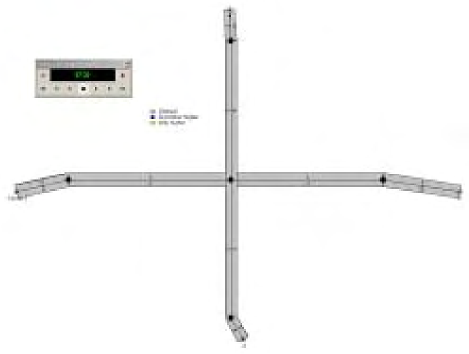
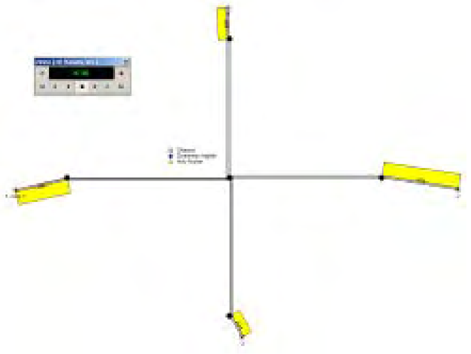



Figure 3.18 : Absolute differences in outflow (left) and density (right) in scenario 2 at 7.30 h, test case 2.

3.4 Network of Bakersfield

3.4.1 Description scenarios, network Bakersfield

The network of Bakersfield is an existing network. For this network several model runs where done with both Dynameq and Indy in the case with signalized and unsignalized crossings. The network is shown in Figure 3.19. The table below shows the network characteristics.

Table 3.12 : network characteristics of scenario 1 and 2, test case 1

	Characteristics
Centroids	65
Nodes	355 (36 signalized intersections)
Links	826
Demand	12091 pcu/h 7.00-7.30 h 12091 pcu/h 7.30-8.00 h 12091 pcu/h 8.00-8.30 h 12091 pcu/h 8.30-9.00 h

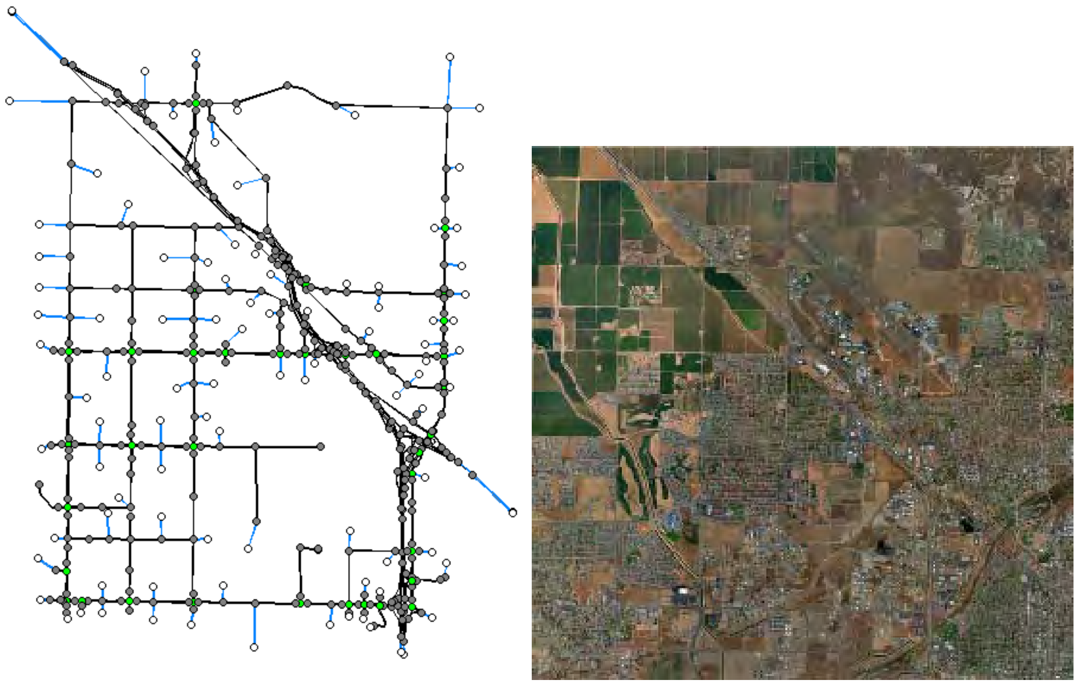


Figure 3.19 : Bakersfield network (green nodes are signalized intersections).

In total two Dynameq runs are carried out: one run with signalized intersections and one run with unsignalized intersections. The model Indy is run three times. Once with the path generation and path choice of Indy, once with the paths and path flows of Dynameq with the signalized intersections and once with the paths and path flows of Dynamic with unsignalized intersections.

It would be nice to see, if it is possible to add the signalized intersections to Indy by adding outflow restrictions and, if necessary, by changing the network configuration. However, this requires quite some work, because a lot of the signalized have different green phases for left, right and through traffic and shared lanes for right and through traffic. The first argument would require changing the network as is done in scenario 3 of test case 2. This could be done automatically, but would make a comparison on the link level difficult, because it would change the link numbers. The second argument (shared lanes) would require node, or actually movement, specific assumptions on the outflow capacity. Although very interesting, this goes beyond the aim of this comparison between Dynameq and Indy.

3.4.2 Results scenarios, network Bakersfield

The model results of Dynameq show that there is hardly any congestion in the network. Only at the signalized intersections there are some delays and at a small part of the motorway. The figures below indicate where this congestion is located and what causes this congestion (to Figure 3.25)

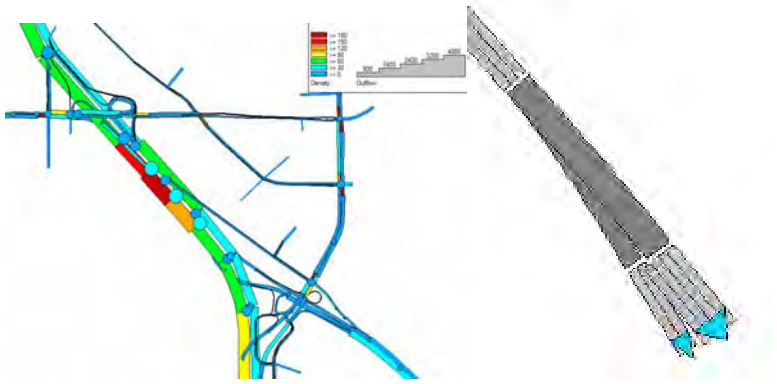


Figure 3.20 : Congested motorway intersection

The figures above show that the congestion is located before a motorway junction where the traffic splits in through traffic and traffic that takes the off-ramp. The road itself upstream of the junction has three lanes. These three lanes can both be used by through traffic and the traffic that takes the off-ramp. Down stream of the junction the motorway has four lanes and therefore isn’t a bottleneck. The off-ramp has two lanes and could therefore be a bottleneck if the traffic volume that takes the off-ramp is larger than the capacity of two lanes. However this is very unlikely and is not the case as is shown in the figure below from which it can be seen that the outflow of the link upstream of the junction to the off-ramp is less than the capacity of one lane.

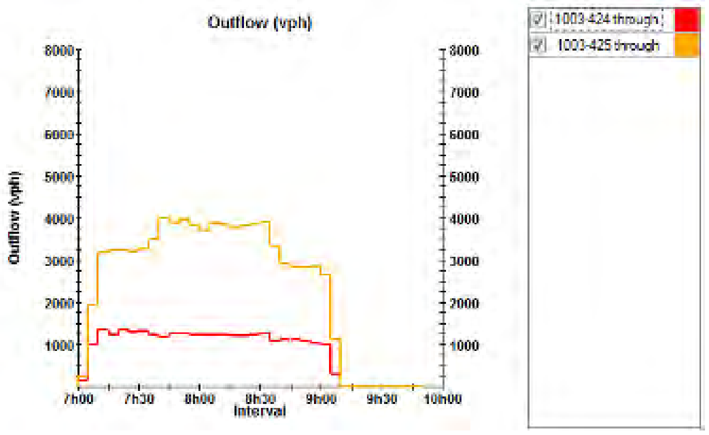


Figure 3.21 : Flows per movement (425 is through and 424 = off ramp)

A view on the outflow and density in the node/junction also doesn’t indicate that the node itself is the bottleneck. From the figure below it can be seen that the outflow and density stay below the capacity of the node.

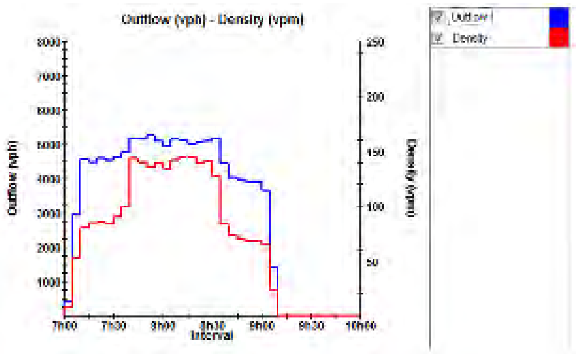


Figure 3.22 : Node outflow and density (node capacity : 3 lanes * 2071 pcu/hour)

So far, there doesn’t seem to be a bottleneck. This is the point where Indy stops, which implies that Indy doesn’t find a bottleneck on the motorway network. However, in Dynameq there is congestion anyway. The congestion appears to be caused by lane changing behavior of drivers. This is something that is not modeled in Indy, but is modeled in Dynameq. The figure below shows the flows per lane upstream of the intersection. Lane 3 is the left most lane. This is the lane that is fully occupied and causes the congestion. Apparently, the drivers that stay on the motorway know (they look ahead) that the off-ramp is coming and therefore they prefer to choose the left-most lane. This happens despite the fact that there is still unused capacity on the right and on the middle lane, which according to the network structure, they could also use to go through.

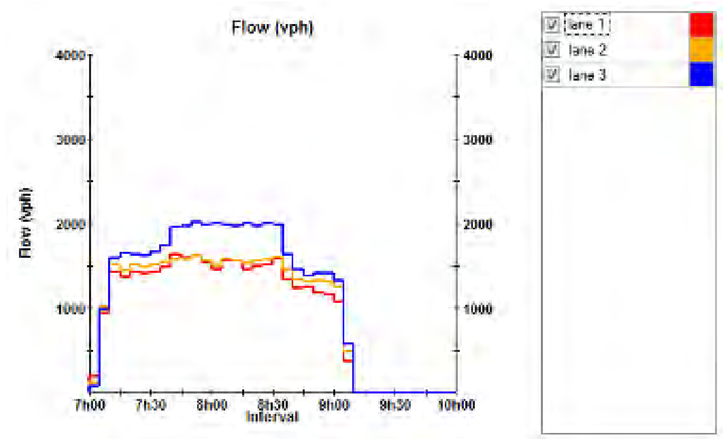


Figure 3.23 : Flows per lane upstream of intersection

The diagram (Figure 3.24) of the forced (blue line) and voluntary lane changes (red line) shows that at certain moments in time almost 20% of the traffic on the upstream link changes lanes. This is what causes the congestion.

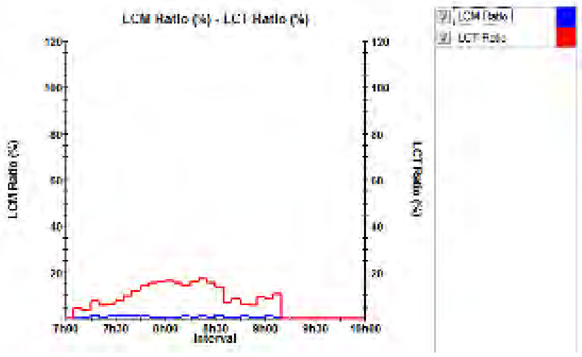


Figure 3.24 : Lane changes on the link upstream of the intersection.

Finally, the lane queues are shown in two moments in time in the figure below. The definition of a queue in Dynameq is very broad: it encompasses all vehicles that are travelling below the free speed of the link. As a result, traffic that is saturated, but still moving at a good speed, will be defined as being in a queue. These figures show that at one moment in time the queue on the left most lane of the link upstream of the junction is the longest. This cause trough traffic to prefer the other lanes and therefore the queue on the left most lane is one time step later shorter than the queue on the other links.

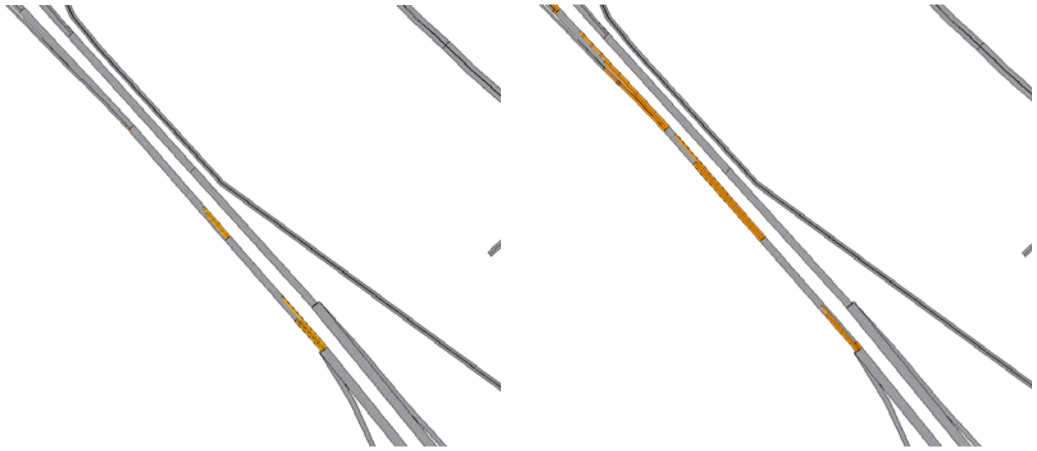


Figure 3.25 : Lane queues (left) 7.35 – 7.40h, (left) 7.40 – 7.45h (right)

Since Indy doesn't model the described lane changing behavior, it doesn't find the congestion on the motorway. In order to minimize this difference in the remainder of comparison an outflow restriction has been imposed upstream of the junction. The outflow restriction is set equal to the maximum outflow that is found in Dynameq. This is of course an approximation, because the maximum outflow capacity is a model result. Actually, it is a result of driver behavior that could in existing networks also be found in the data. In that case the outflow restriction would be imposed on the specific link as a result of the calibration of Indy. The results presented below are the results with an maximum outflow restriction on the specific link.

In Table 3.13 the aggregated vehicle hours travelled, vehicle kilometers travelled and the average speeds are shown. The average speeds in the Dynameq outcomes of the signalized network are about 12 km/hour lower than in all the other model runs. Thus, also in the model run in which Indy uses the paths and path flows produced by the same Dynameq run. This is explained by the fact that Indy doesn't model the delays at the intersections. In the case in which Dynameq is run with unsignalized intersections the average speeds come very close to the average speeds computed by Indy.

Besides that, it can be seen that the results of the three model runs with Indy are very close to each other. This suggests that the equilibrium that is found by Indy is close to the equilibrium that is found by Dynameq. It is remarkable that the equilibrium run of Indy is in fact closer to the equilibrium run of Dynameq with signalized intersections than the equilibrium run with Dynameq with unsignalized intersections, because Indy doesn't use signals. From to total vehicle kilometers driven it can be seen that the slighter shorter paths (shorter in distance) are chosen if the signals are not used. This suggests that the signals on the shorter paths cause delays which makes travelers choose longer (in distance) paths. Finally, it can be seen that the travelled kilometers of the Indy runs with Dynameq paths are not exactly the same as the vehicle kilometers computed by Dynameq. This strange, because they should be exactly the same. However, the differences are so small that they are probably caused by rounding errors.

Table 3.13 : Network results Bakersfield network

	Dynameq signalized	Dynameq unsignalized	Indy paths Indy (Indy a)	Indy paths Dynameq signalized (Indy b signalized)	Indy paths Dynameq unsignalized (Indy b unsignalized)
Vehicle hours travelled	5614	4632	4746	4727	4478
Vehicle km travelled	348402	344114	348306	348503	344311
Average speed	62.1	74.3	73.4	73.7	76.9

Figure 3.26, Figure 3.27 and Figure 3.28 show respectively the average speed, the total travel time and the average lane density in the network over time for the five different model runs. These figures show that the development over time of all three indicators is more or less the same in all Dynameq and Indy runs. The high peeks in the speeds of Dynameq at the end of the simulation are probably caused by interpolation errors. At least, they can't be realistic speeds, because they are higher than the maximum speed.

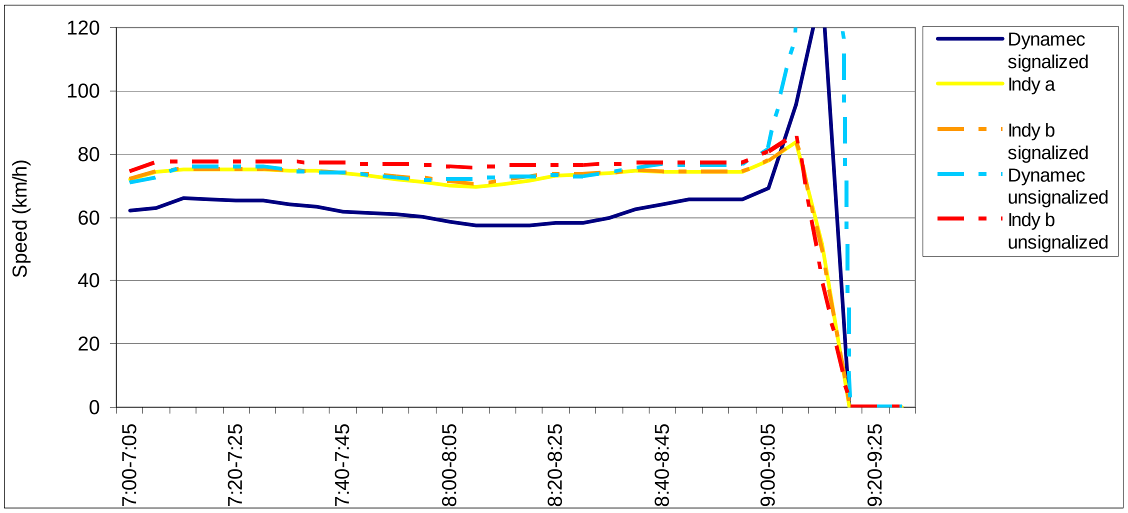


Figure 3.26 : Average speed in the network over time in the Bakersfield network.

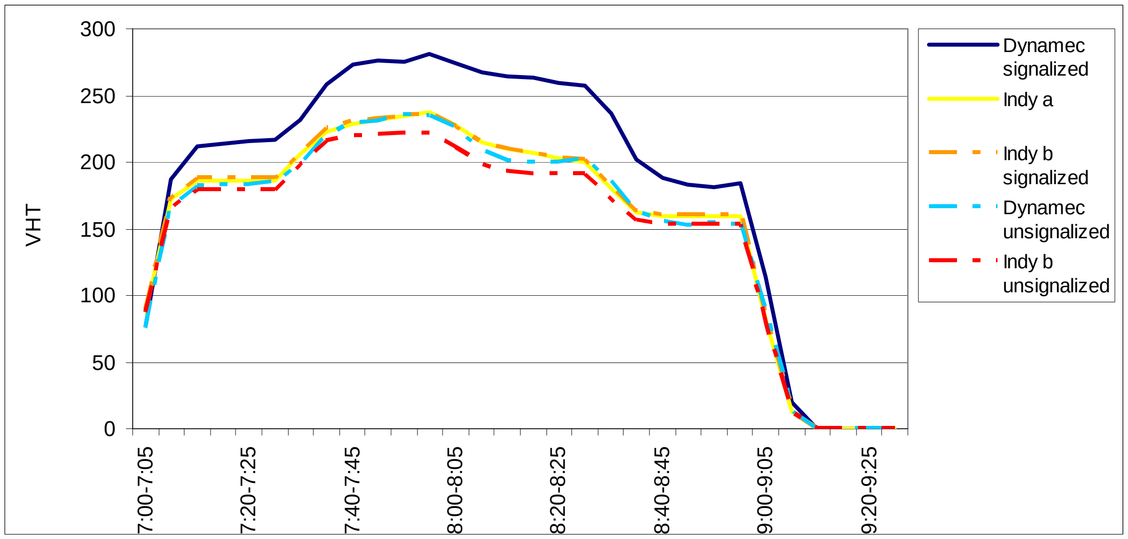


Figure 3.27 : Total travel time of the vehicles in the network over time in the Bakersfield network.

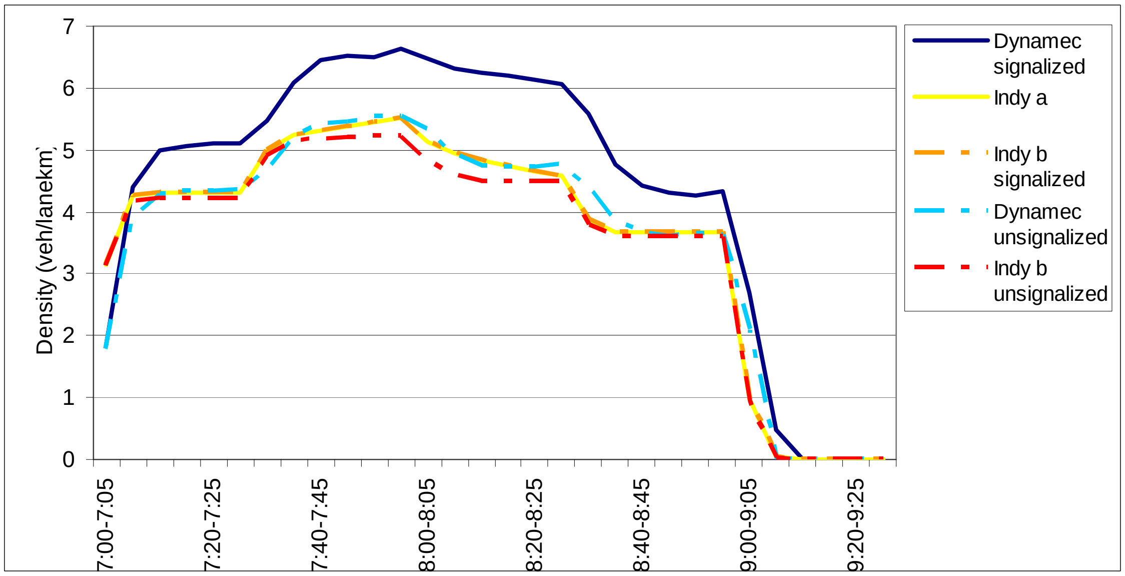


Figure 3.28 : Average lane density over time in de Bakersfield network.

A comparison of time per link could give even more insights in the model differences. In Figure 3.29 to Figure 3.32 a scatter plot of the inflow, outflow, speed and density is shown for four situations:

1. Dynameq signalized versus Indy with path generation and path choice by Indy.
2. Dynameq signalized versus Indy with Dynameq paths with signals.
3. Dynameq unsignalized versus Indy with path generation and path choice by Indy.
4. Dynameq unsignalized versus Indy with Dynameq paths without signals.

A regression line is added in all scatter plots. The R^2 of the regressions show the extent to which both models correlate. The R^2 is summarized in Table 3.14. From this table and from the figures it can be seen that both the inflow and outflow show an excellent fit. That is, the outflow and inflow of Dynameq and Indy are more or less the same. In the case in which Indy uses the paths of Dynameq this is logical. Although even in that case delays can cause differences in inflow and outflow over time (but not over links). The fact that the inflow and outflow have a R^2 above 0.95 indicates that the equilibrium route choice of both models doesn't differ much.

The R^2 of the speeds is low (0.24) in all four cases. A more detailed analysis shows that this is for a large part caused by links at intersections. If these links are left out of the analysis the R^2 goes up to 0.53 in case of signalized intersections and 0.70 in case of unsignalized intersections and an Indy run with Dynameq paths. The R^2 would probably go up further if al the links with delays caused by intersections (instead of only the links directly upstream of the intersections) are left out of the analysis. This also illustrates, that signals are at intersections for a reason. Without the signals, Dynameq still comes up with significant delays at the intersections which are not recognized by Indy.

The R^2 of the density is even worse (0.18 or 0.19) than the speed in case of signalized intersections. However, in the case of unsignalized intersections the R^2 goes up to 0.67 and 0.76.

Table 3.14 : R^2 inflow, outflow, speed and density of Dynameq versus Indy.

	inflow	outflow	speed	density
1: signalized-indy paths	0.96	0.96	0.24	0.18
2: signalized-Dynameq paths	0.99	0.99	0.24	0.19
3: unsignalized-indy paths	0.98	0.98	0.24	0.67
4: unsignalized-Dynameq paths	1.00	1.00	0.24	0.76

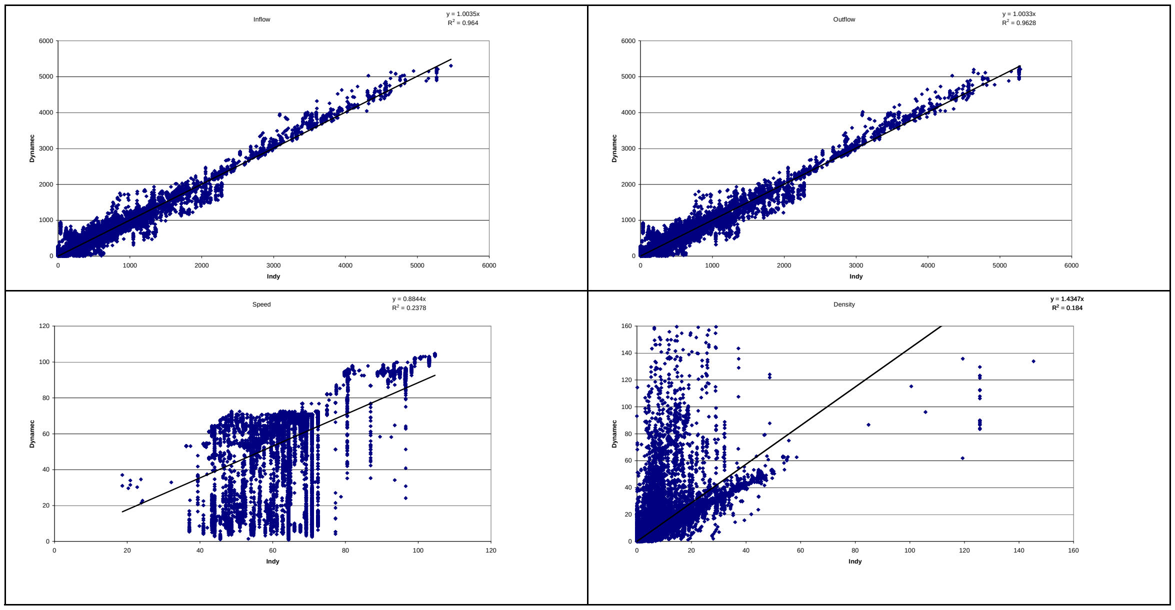


Figure 3.29: Inflow (top left), Outflow (top right), Speed (bottom left) and density (bottom right) of Dynameq with signalized intersections(y-axis) versus Indy with paths generated by Indy(x-axis) per link and time slice.

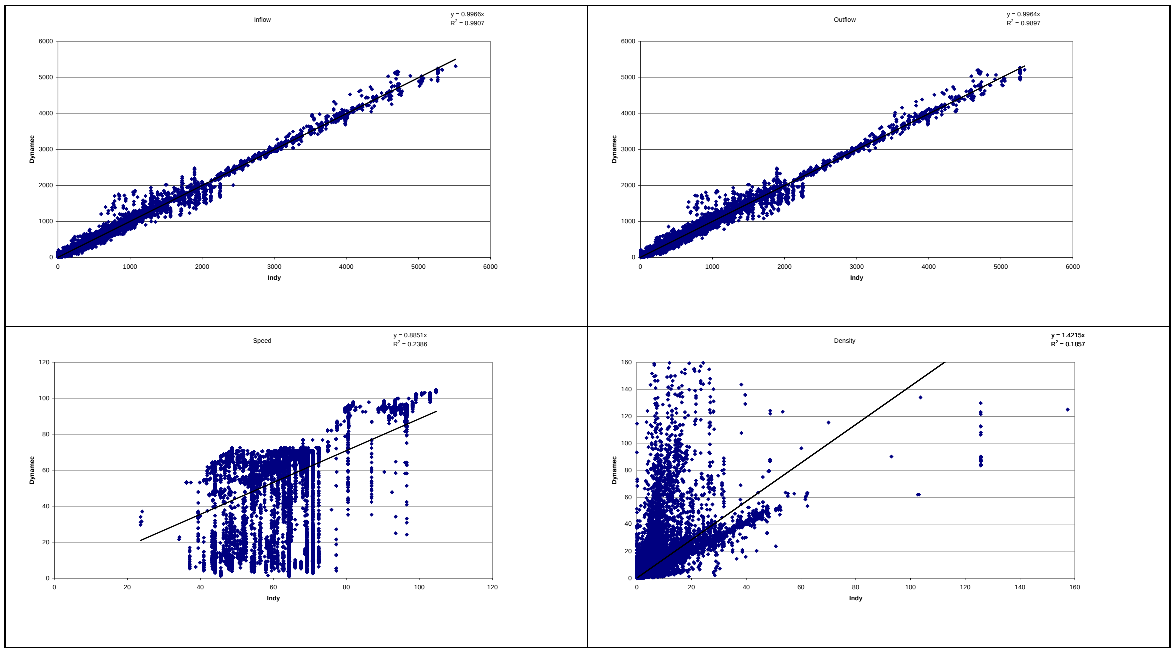


Figure 3.30: Inflow (top left), Outflow (top right), Speed (bottom left) and density (bottom right) of Dynameq with signalized intersections(y-axis) versus Indy with Dynameq paths(x-axis) per link and time slice.

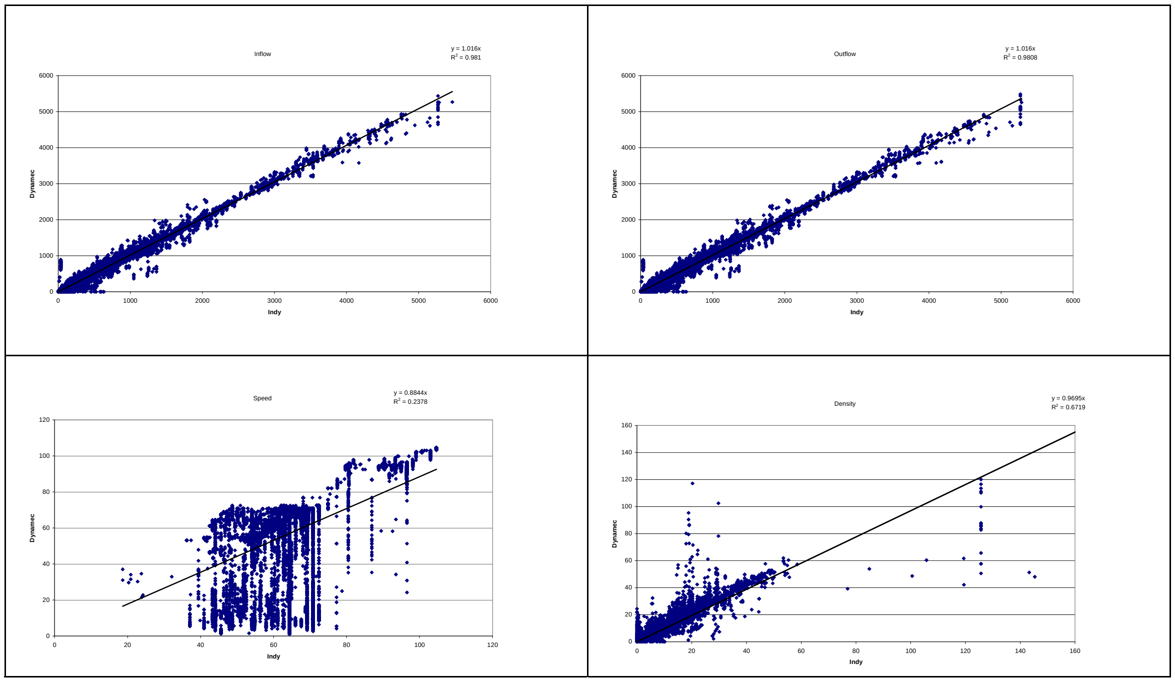


Figure 3.31: Inflow (top left), Outflow (top right), Speed (bottom left) and density (bottom right) of Dynameq with unsignalized intersections(y-axis) versus Indy with paths generated by Indy(x-axis) per link and time slice.

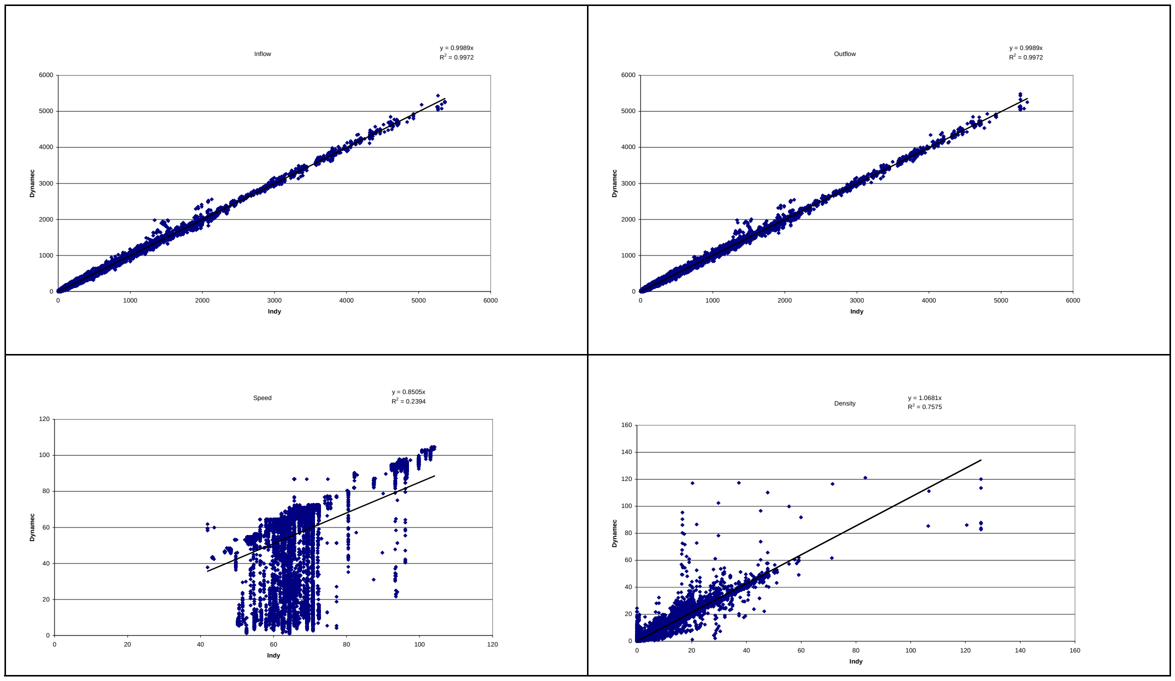


Figure 3.32: Inflow (top left), Outflow (top right), Speed (bottom left) and density (bottom right) of Dynameq with unsignalized intersections(y-axis) versus Indy with Dynameq paths (x-axis) per link and time slice.

The figures below show the difference in the density between Dynameq and Indy at 8.00 h (with Dynameq paths). In the figure on the left traffic signals are used in Dynameq and in the figure on the right traffic signals are not used. These figure on the left shows that the links before the traffic signals are, as was suggested above, indeed an important difference. On the right, the differences on these links are almost not visible any more. Furthermore, on the motorway there is a relatively big difference, despite the fact the outflow restriction was imposed in Indy. The fact that Indy has a higher density on the two links before the junction which causes the congestion seems to be strange at first sight, because the maximum outflow was set to the maximum outflow modeled by Dynameq. A possible explanation for this can be found in the look a head lane changing behavior of Dynameq. This explanation is in line with the fact, that the densities of Dynameq are higher than the densities computed by Indy more upstream. A maximum outflow constraint is therefore, not a very good approximation of delays caused by lane changing behavior.

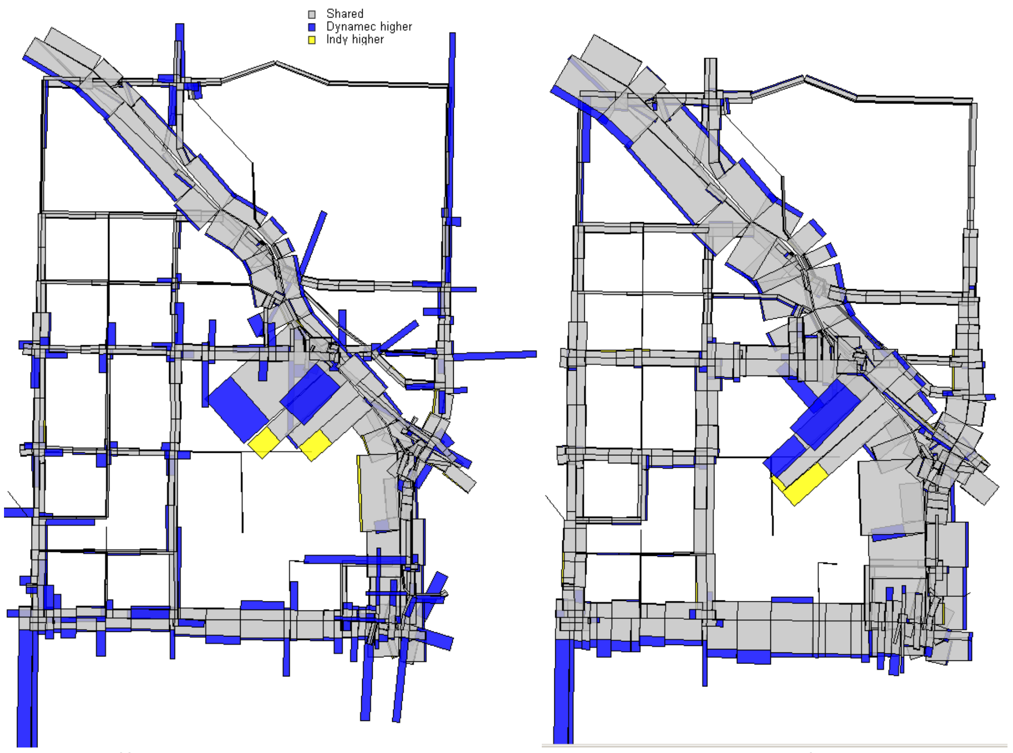


Figure 3.33 : Difference in density at 8.00h in the case with unsignalized (left) and signalized (right) intersections and Dynameq paths.

In Figure 3.34 to Figure 3.36 the relative duality gaps are shown. Only 5 iterations with Indy have been carried out in the case when Indy uses its own paths. There was no point in doing more than 5 iterations since already in the first iteration the gaps in all four demand intervals is below 1.25%. They stay more or less the same in the other four iterations. The reason for these low gaps is that there isn’t much congestion on the network. The delays at intersections are not found by Indy and therefore not included in the calculation of the gaps. Besides that, the initial spread over all generated paths is already good in the first iteration.

Dynameq converges slower and keeps higher gaps also in the later iterations. The first cause for this is that Dynameq models more delays (and therefore bigger differences between paths). This hypothesis is strengthened by the fact that the gaps in the case without signals are already much lower (below 10%) than in the case with signals in which the gaps go up to 70%. A second cause is that Dynameq hasn’t found all the paths yet in the first 10 iterations, which causes the high gaps in the first iterations. Finally, the fact that Dynameq uses a deterministic assignment results in the fact that the traffic is less spread over all available paths.

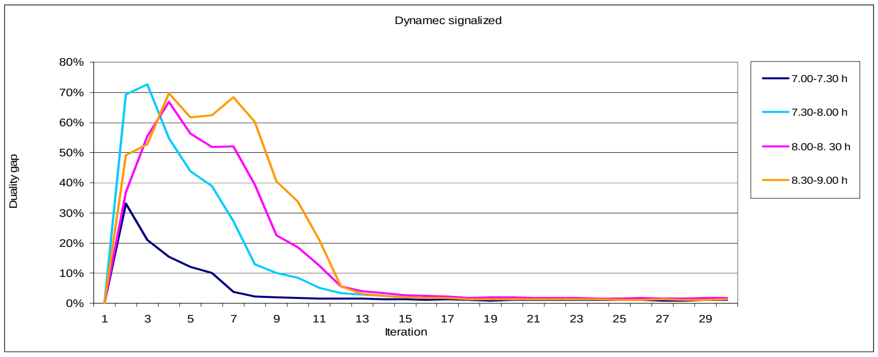


Figure 3.34: Duality gaps of Dynameq in the case with signalized intersections.

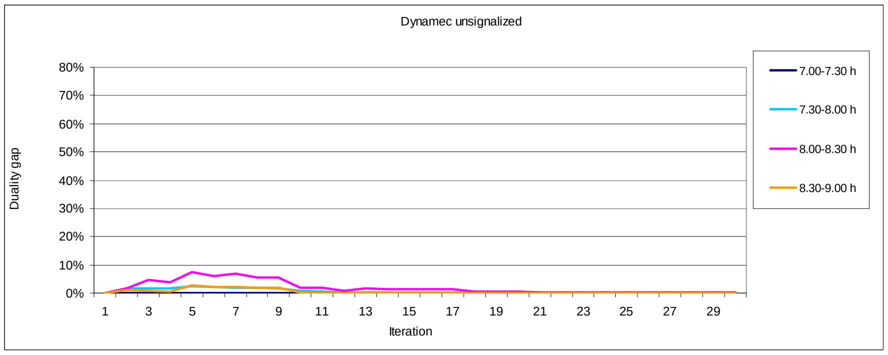


Figure 3.35 : Duality gaps of Dynameq in the case without signalized intersections.

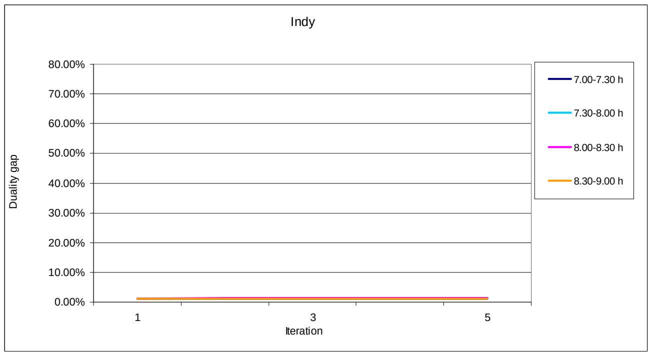


Figure 3.36 : Duality gaps of Indy.

Finally, in Figure 3.37 the computation times per iteration are shown. In this case, Indy was run with a time step of 5 seconds. This time step is slightly larger than the free flow link travel time of 141 links. The smallest free flow travel time is 1.3 seconds. This implies that for these links the link lengths have been extended during the simulation in such a way that they have a free-flow travel time of 5 seconds. With this time step the computation time per iteration is still almost 10 times as high as in Dynameq, which is likely to be caused by the fact that Dynameq is event based and many links have a much higher free flow travel time than 5 seconds. A second explanation might be the number of used paths. In total there are 2988 OD-pairs with a demand larger than 0. Indy generated 8399 paths and there is flow on all these paths. Dynameq generated 19692 paths, but there is only flow on 6200 paths. Since the computation time of Indy depends on the number of paths that are used in the evaluation phase, this could also be an explanation for the longer computation times.

In the comparison in which the Dynameq paths are used, a time step of 1 second is used in Indy. Therefore, in these runs the link lengths didn't have to be extended. For these runs only one iteration was needed. This iteration took 37 minutes, which is more than 5 times (9.4) higher than the case with a time step of 5 seconds. The explanation for this might as well be the number of paths. Although, only 6200 paths are used, all 19692 paths are considered in Indy.

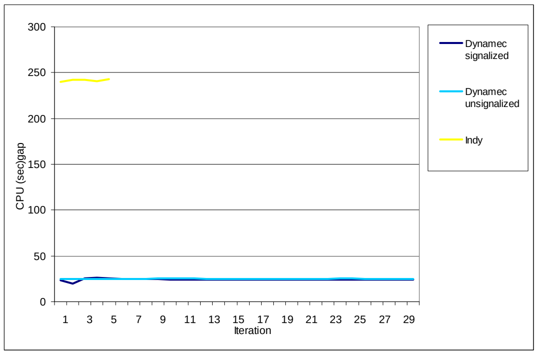


Figure 3.37 : Computation times on the Bakersfield network.

4 Conclusions and recommendations

The comparison of the specifications of Dynameq and Indy showed there are several important similarities and differences between both models. The models are comparable in the sense that both are equilibrium models in which paths are generated, path choice plays a role and dynamic network loading takes place in an iterative process. The newest and most accurate network loading model in Indy (LTM) works according to Newell's kinematic wave theory and so does the network loading algorithm in Dynameq. The main differences are:

- Dynameq generates paths in the first iterations of the simulation, whereas Indy generates paths before the simulation starts.
- Dynameq has a deterministic path choice, whereas Indy has a stochastic path choice.
- Dynameq is lane based, whereas Indy is linked based.
- Dynameq models individual vehicles, whereas Indy models aggregated flows (per path). This enables Dynameq to model gap acceptance at intersections and lane changing behavior, which can't be done by Indy.
- Dynameq has a more detailed intersection model than Indy. It can deal with traffic signals and priority flows, whereas Indy can only approximate this by introducing a maximum link outflow capacity.
- Dynameq is event based, whereas Indy works with fixed time steps.

Both models are run on three networks to show how the above mentioned differences influence the model outcomes. The first test network was a network in which delays at intersections where excluded and in principle two equal paths are available for the single OD-pair. It showed that:

- That stochastic path choice of Indy in combination with generating trips can have advantages over deterministic path choice, since already in the first iteration the traffic is spread over all the pre-specified paths. In this way Indy converges faster to an equilibrium than Dynameq. This general conclusion is illustrated by the fact that in the first scenario Dynameq doesn't find a second path because there is no congestion on the first path. Therefore, in Dynameq the capacity of the network is not used fully which results in the fact that it takes Dynameq twice as long to get all the traffic over the network. It must be said that this is an extreme example, because in larger networks links will be used by multiple paths from multiple OD-pairs, which always causes some delays and therefore extra paths to be generated. The differences in path choice also became clear in the second scenario where Indy finds a perfect equilibrium in which 50% of the traffic uses each path. The spread in Dynameq after the first 30 iterations is 53%-47%.
- In the second scenario Dynameq does find two paths, which makes the results of Dynameq and Indy more comparable. In fact if Indy uses the paths of Dynameq, the results are almost identical. Which shows that both network loading models are the same if delays at intersections and lane changing behavior doesn't play a role as was to be expected based on the model specifications.
- In this test network the links are very long which allows for a high time step Indy. The time step could even be set to 180 seconds. However a time step of 60 seconds is used. Even with this time step Indy is about 4 times as fast as Dynameq. This is probably caused by the fact that Indy is a macroscopic model and, therefore, doesn't have to keep track of individual vehicles and the way in which they behave. On the other hand, if the links, or even only one of the links, would have been shorter, a smaller time step had to be chosen which would increase the computation time of Indy. In that case Dynameq becomes faster than Indy which is illustrated better on the Bakersfield network. This shows the consequences of having event based or time step based models.

The second test network was a network with four zones and flows between all zones. There is only one path between each OD-pair. Therefore path choice doesn't play a role. This example emphasizes the differences between both models at intersections:

- In the first scenario the intersection is unsignalized. In this case the differences in model outcomes are very large. The total travel in Indy is about twice as low as in Dynameq and the network is almost an hour earlier empty. The explanation for this is that Indy doesn't consider delays at intersections that are caused by waiting for gaps that are needed to cross an intersection. In Indy cars can virtually drive over each other, which is of course not realistic. The fact that the travel times are twice as low shows that the delays at intersections can add up to a substantial amount. On the other hand, the flows in this network are high. If there is less traffic in the network, the delays at the intersections also reduce.

- The two scenarios with signalized intersections showed that it is possible to model signalized intersections by outflow constraints, because the results of Indy and Dynameq are exactly the same for both scenarios. This is however only possible in the situation in which there are no conflicts in lane usage, because in those situations the outflow is a result of the arrival pattern of traffic on the intersection. Therefore, the maximum outflow cannot be computed based on the link capacity, green times and cycle lengths. For those situations an approximate outflow capacity has to be found.

Finally, both models are compared on a realistic network: the Bakersfield network. All the differences mentioned above become more clear in this network. Besides that some additional differences became clear:

- In the Bakersfield network congestion occurs on the motorway due to lane changing behavior. This congestion is recognized by Dynameq. Since Indy doesn't model the described lane changing behavior, it doesn't find the congestion on the motorway. An approximate outflow restriction is imposed in Indy on the link upstream of the junction where the congestion starts. However, this appears not to capture the dynamics in the traffic flow caused by lane changing behaviour completely.
- Dynameq is run with signalized and unsignalized intersections and Indy is run twice with the path and path flows of these runs and once with its own paths and path choice. The average speeds in the Dynameq outcomes of the signalized network are about 12 km/hour lower than in all the other model runs. Thus, also in the model run in which Indy uses the paths and path flows produced by the same Dynameq run. This is explained by the fact that Indy doesn't model the delays at the intersections. In the case in which Dynameq is run with unsignalized intersections the average speeds come very close to the average speeds computed by Indy.
- The results of the three model runs with Indy are very close to each other. This suggests that the equilibrium that is found by Indy is close to the equilibrium that is found by Dynameq. It is remarkable that the equilibrium run of Indy is in fact closer to the equilibrium run of Dynameq with signalized intersections than the equilibrium run with Dynameq with unsignalized intersections, because Indy doesn't use signals. From the total vehicle kilometers driven it can be seen that slightly shorter paths (shorter in distance) are chosen if the signals are not used. This suggests that the signals on the shorter paths cause delays which makes travelers choose longer (in distance) paths.
- The development of the average speed, the total travel time and the average lane density in the network over time for the five different model runs is more or less the same over time in all Dynameq and Indy runs. Which indicates that, despite all the before mentioned differences, the model outcomes also show a lot of similarities.
- A comparison over time per link showed that there is a very high correlation between Dynameq and Indy if the inflow and outflow are compared. In the case in which Indy uses the paths of Dynameq this is logical. Although even in that case delays can cause differences in inflow and outflow over time (but not over links). The fact that the inflow and outflow have a R^2 above 0.95 indicates that the equilibrium route choice of both models doesn't differ much. However, the R^2 of the speeds is low (0.24). For a large part this is caused by links at intersections. The R^2 of the density is even worse (0.18 or 0.19) than the speed in case of signalized intersections. However, in the case of unsignalized intersections the R^2 goes up to 0.67 and 0.76. This emphasizes the need to model delays at intersections explicitly.
- Dynameq converges slower and keeps higher gaps also in the later iterations. The first cause for this is that Dynameq models more delays (and therefore bigger differences between paths). This hypothesis is strengthened by the fact that the gaps in the case without signals are already much lower (below 10%) than in the case with signals in which the gaps go up to 70%. A second cause is that Dynameq hasn't found all the paths yet in the first 10 iterations, which causes the high gaps in the first iterations. Finally, the fact that Dynameq uses a deterministic assignment results in the fact that the traffic is less spread over all available paths.
- In the equilibrium run with Indy a time step of 5 seconds was used. This time step is slightly larger than the free flow link travel time of 141 links. The smallest free flow travel time is 1.3 seconds. This implies that for these links the link lengths had to be extended during the simulation in such a way that they have a free-flow travel time of 5 seconds. With this time step the computation time per iteration is still almost 10 times as high as in Dynameq, which is likely to be caused by the fact that Dynameq is event based and many links have a much higher free flow travel time than 5 seconds. A second explanation might be the number of used paths. In total there are 2988 OD- pairs with a demand larger than 0. Indy generated 8399 paths and there is flow on all these paths. Dynameq generated 19692 paths, but there is only flow on 6200 paths. Since the computation time of Indy depends on the number

of paths that are used in the evaluation phase, this could also be an explanation for the longer computation times. In the comparison in which the Dynameq paths are used, a time step of 1 second is used in Indy. Therefore, in these runs the link lengths didn't have to be extended. For these runs only one iteration was needed. This iteration took 37 minutes, which is more than 5 times (9.4) higher than the case with a time step of 5 seconds. The explanation for this might as well be the number of paths. Although, only 6200 paths are used, all 19692 paths are considered in Indy.

The above mentioned differences lead to the following recommendations for Dynameq and Indy:

Dynameq:
- Generating the paths before the simulation starts or storing the paths of a previous model run can reduce the number of iterations that is needed to reach an equilibrium and therefore decrease the computation time. It might be worthwhile to investigate if this is possible.
- Stochastic route choice instead of deterministic route choice could lead to faster convergence as well. However, changing this has bigger implications, since it is a fundamental change in the assumed route choice behavior.

Indy:
- The modelling of delays at intersections can be improved. For the level on which Indy is currently mostly used (high level, with mainly motorways) this is less important than for cases in which local networks are used. However, it could still be large improvement. This requires more input data, but could make the calibration easier in the end. A first improvement which is probably relatively easy is the introduction of the option the include maximum outflow constraints per movement instead of per link. This prevents that links has to be split in three separate links to replicate the structure of a node. Other improvements that are for example needed to include the gap acceptance principle might be investigated as well which is already being done by the university of Leuven.
- Including lane changing behavior is not possible in Indy. It might be worthwhile to investigate how this behaviour can be approximated.
- A practical suggestion is not to use the adjusted link capacities from networks that are used in static assignment models and adjust those capacities a bit further in the calibration, but to reset the capacities to the level that is to be expected based on free-flow speeds, average vehicle lengths and a response time. In the calibration the maximum outflow capacities can then better be adjusted instead of the capacities themselves.
- Investigate the possibility and the gains in computation time of switching from a time step based network loading model to an event based model. This is probably relatively easy since the university of Leuven already has an event based version of LTM.
- Remove paths that are not used or barely used during the simulation to reduce the memory usage and increase the computation speed.

References

Astarita, V. (1996) A Continuous Time Link Model for Dynamic Network Loading Based on Travel Time Function. In: J.-B. Lesort (ed.) Transportation and Traffic Flow Theory, Pergamon, pp. 79-102.

Astarita, V., Er-Rafia, K., Florian, M., Mahut, M., Velan, S. (2001). Comparison of Three Methods for Dynamic Network Loading. Transportation Research Record, 1771:179-190.

Bliemer , M.C.J. and P.H.L. Bovy (2003) Quasi-Variational Inequality Formulation of the Multiclass Dynamic Traffic Assignment Problem. Transportation Research B, Vol. 37, pp. 501-519.

Bliemer, M.J.C., INDY 2.0 Model Specifications. Delft Univerisity of Technology, 2005.

Bliemer, M.J.C. Dynamic Queuing and Spillback in an Analytical Multiclass Dynamic Network Loading Model. In Transportation Research Record: Journal of the Transportation Research Board, No. 2029, Transportation Research Board of the National Academies, Washington D.C., 2007.

Chabini, I. (2001) The Analytical Network Loading Problem: Formulation, Solution Algorithms and Computer Implementations. Transportation Research Records, No. 1771, TRB, National Research Council, Washington DC, USA, pp. 191-200.

Daganzo, C.F. (1994) The Cell Transmission Model: A Dynamic Representation of Highway Traffic Consistent with Hydrodynamic Theory. Transportation Research B, Vol. 28, No. 4, pp. 269-287.

Daganzo, C.F. (1995) The Cell Transmission Model, Part II: Network Traffic. Transportation Research B, Vol. 29, No. 2, pp. 79-93.

Florian, M., Mahut, M., Tremblay, N. (2008), A simulation based dynamic traffic assignment: the model, European Journal of Operational Research, Volume 189, Issue 3, pp. 1381-1392.

He, Y. (1997) A flow-based approach to the dynamic traffic assignment problem: Formulations, algorithms and computer implementations. MSc. Thesis, Massachusetts Institute of Technology, MIT Press, Cambridge MA, USA.

Mahut, M. (2000). Discrete flow model for dynamic network loading. Ph.D. Thesis, Département d'informatique et de recherhe opérationelle, Université de Montréal. Published by the Center for Research on Transportation (CRT), University of Montreal.

Mahut, M., Florian, M., Tremblay, N., Campbell, M., Patman, D. and McDaniel., Z. (2004) Calibration And Application Of A Simulation-Based Dynamic Traffic Assignment Model, Transportation Research Record 1876: 101-111.

Mahut, M., Florian, M.,Tremblay, N. (2008),Comparison of assignment methods for simulation-based dynamic-equilibrium traffic assignment, TRB 2008.

Ran, B. and D.E. Boyce (1996) Modeling Dynamic Transportation Networks : An Intelligent Transportation System Oriented Approach. Second edition, Springer-Verlag, Berlin, Germany.

Roels, G. and G. Perakis (2004) An Analytical Model for Traffic Delays. Proceedings of the TRISTAN V Conference, Guadeloupe, France.

Xu, Y.W., H. Wu, M. Florian, P. Marcotte and D.L. Zhu (1999) Advances in the Continuous Dynamic Network Loading Problem. Transportation Science, Vol. 33, No. 4, pp. 341-353.

Yperman I. The Link Transmission Model for Dynamic Network Loading, Katholieke Universiteit Leuven, Faculteit Ingenieurs Wetenschappen, Afdeling Verkeer en Infrastructuur, Leuven, June, 2007.

Wu, J.H., Y. Chen and M. Florian (1998) The Continuous Dynamic Network Loading Problem: A Mathematical Formulation and Solution Method. Transportation Research B, Vol. 32, No. 3, pp. 173-187.

Appendix: conversion algorithms

Export2Dynameq.rb

```
#This job exports an Indy network to Dynameq

#parameters
mode = 1
jamdensity = 150
centroidtype = 1		#pointtype of nodes that are centroides
normalnodetype = 2			#pointtype of nodes that aren't centroides
starttime = 700
endtime = 800

#Create network outputfile starting with vehicle class and nodes
file = File.open("#{$Ot.variantDirectory()}BasenetDynameq.txt", "w")
file.puts("VEH_CLASS")
file.puts("PCU")
file.puts("NODES")
file.puts("*id	x	y	z	control	priority	type")
file.close

#Export Nodes
file = File.open("#{$Ot.variantDirectory()}BasenetDynameq.txt", "a")
sql = OtQuery.new("SELECT pointnr, pointtype, x, y " +
   "FROM '#{$Ot.projectDirectory()}point' pt " +
   "WHERE pt.pointtype = #{normalnodetype}")
sql.open
recordCounter = 0
nbrOfRecords = sql.recordCount
while (!sql.eof?)
	record = sql.get()
	pointnr = record[0]
	x = record[2]
	y = record[3]
	# write to file
	file.puts("#{pointnr}	#{x}	#{y}	1	0	0	1")
	sql.next
end
sql.close

#Export Centroids
file.puts("CENTROIDS")
file.puts("*id	noderef	x	y	z")

sql = OtQuery.new("SELECT pointnr, pointtype, x, y " +
   "FROM '#{$Ot.projectDirectory()}point' pt " +
   "WHERE pt.pointtype = #{centroidtype}")
sql.open
recordCounter = 0
nbrOfRecords = sql.recordCount
while (!sql.eof?)
	record = sql.get()
	pointnr = record[0]
	x = record[2]
	y = record[3]
	# write to file
```

```
        file.puts("#{pointnr}     -1      #{x}    #{y}    1")
        sql.next
end
sql.close

file.puts("LINKS")
file.puts("*id     start    end     reverse  type    faci    len     vfree   kjam    qsat    lanes   rabout
        wpen    class(/lane)")

sql = OtQuery.new("SELECT linenr, pointnra, pointtypea, pointnrb, pointtypeb, direction, capacity,
freespeed, lk.'length', lk1.'lanes', lk2.'typenr'" +
   "FROM '#{$Ot.projectDirectory()}line' ln, link lk, link1_1data2 lk1, link2_1data1 lk2, link3_1data1 lk3
" +
   "WHERE ln.linenr = lk.linknr and " +
   "lk3.mode = #{mode} and " +
   "ln.linenr = lk3.linknr and " +
   "lk2.linknr = lk3.linknr and lk2.direction = lk3.direction and " +
   "lk1.linknr = lk3.linknr and lk1.direction = lk3.direction")
   #ln.pointtypea = #{normalnodetype} and " +
   #"ln.pointtypeb = #{normalnodetype} and "+
sql.open
nbrOfRecords = sql.recordCount
while (!sql.eof?)
        record = sql.get()
        linenr = record[0]
        pointnra = record[1]
        pointtypea = record[2]
        pointnrb = record[3]
        pointtypeb = record[4]
        direction = record[5]
        capacity = record[6]
        freespeed = record[7]
        length = record[8]
        lanes = record[9]
        type = record[10]
        linkid = linenr*10 + direction
        capacity = capacity/lanes
        #jamdensityroad = jamdensity*lanes
        # write to file
        sql2 = OtQuery.new("SELECT link2_1data1.linknr FROM link2_1data1 WHERE
link2_1data1.linknr=#{linenr}")
        sql2.open
        reversecount = sql2.recordCount
        sql2.close


        if (direction == 1)
                if (pointtypea == normalnodetype) and (pointtypeb == normalnodetype)
                        if (reversecount == 1)
                                file.puts("#{linkid}     #{pointnra}     #{pointnrb}     -1      0
        #{type} #{length}       #{freespeed}    #{jamdensity}   #{capacity}     #{lanes}        0
        0       *")
                        else
                                file.puts("#{linkid}     #{pointnra}     #{pointnrb}     #{linenr}2
        0       #{type} #{length}       #{freespeed}    #{jamdensity}   #{capacity}     #{lanes}
        0       0       *")
                        end
                end
        else
                if (pointtypea == normalnodetype) and (pointtypeb == normalnodetype)
                        if (reversecount == 1)
                                file.puts("#{linkid}    #{pointnrb}     #{pointnra}     -1      0
        #{type} #{length}       #{freespeed}    #{jamdensity}   #{capacity}     #{lanes}        0
        0       *")
                        else
                                file.puts("#{linkid}    #{pointnrb}     #{pointnra}     #{linenr}1
        0       #{type} #{length}       #{freespeed}    #{jamdensity}   #{capacity}     #{lanes}
        0       0       *")
                        end
                end
        end
        sql.next
end
sql.close

file.puts("CONNECTORS")
file.puts("*id  start   end     vnode   linkref faci    len     vfree   kjam    qsat    lanes   rabout
        wpen    class(/lane)")

sql = OtQuery.new("SELECT linenr, pointnra, pointtypea, pointnrb, pointtypeb, direction, capacity,
freespeed, lk.'length', lk1.'lanes', lk2.'typenr'" +
   "FROM '#{$Ot.projectDirectory()}line' ln, link lk, link1_1data2 lk1, link2_1data1 lk2, link3_1data1 lk3
" +
   "WHERE ln.linenr = lk.linknr and " +
   "lk3.mode = #{mode} and " +
   "ln.linenr = lk3.linknr and " +
   "lk2.linknr = lk3.linknr and lk2.direction = lk3.direction and " +
   "lk1.linknr = lk3.linknr and lk1.direction = lk3.direction")
   #ln.pointtypea = #{normalnodetype} and " +
   #"ln.pointtypeb = #{normalnodetype} and "+
sql.open
nbrOfRecords = sql.recordCount
while (!sql.eof?)
        record = sql.get()
        linenr = record[0]
        pointnra = record[1]
        pointtypea = record[2]
        pointnrb = record[3]
        pointtypeb = record[4]
        direction = record[5]
        capacity = record[6]
        freespeed = record[7]
        length = record[8]
        lanes = record[9]
        type = record[10]
        linkid = linenr*10 + direction
        #jamdensityroad = jamdensity*lanes
        # write to file
        sql2 = OtQuery.new("SELECT link2_1data1.linknr FROM link2_1data1 WHERE
link2_1data1.linknr=#{linenr}")
        sql2.open
        reversecount = sql2.recordCount
        sql2.close
        if (direction == 1)
```

```
                if (pointtypea == centroidtype)
                        file.puts("#{linkid}        -#{pointnra}        #{pointnrb}        -1        -1
        #{type} #{length}        #{freespeed}        #{jamdensity}        #{capacity}        #{lanes}        0
        0        *")
                elsif (pointtypeb == centroidtype)
                        file.puts("#{linkid}        #{pointnra}        -#{pointnrb}        -1        -1
        #{type} #{length}        #{freespeed}        #{jamdensity}        #{capacity}        #{lanes}        0
        0        *")
                end
        else
                if (pointtypea == centroidtype)
                        file.puts("#{linkid}        #{pointnrb}        -#{pointnra}        -1        -1
        #{type} #{length}        #{freespeed}        #{jamdensity}        #{capacity}        #{lanes}        0
        0        *")
                elsif (pointtypeb == centroidtype)
                        file.puts("#{linkid}        -#{pointnrb}        #{pointnra}        -1        -1
        #{type} #{length}        #{freespeed}        #{jamdensity}        #{capacity}        #{lanes}        0
        0        *")
                end
        end
        sql.next
end
sql.close
file.puts("MOVEMENTS")
file.puts("*at        in        out        vfree        class        lanes        inlane        outlane  tfollow")
file.close

#Create matrix outputfile
file = File.open("#{$Ot.variantDirectory()}BasematrixDynameq.txt", "w")
file.puts("VEH_CLASS")
file.puts("PCU")
file.puts("DATA")
file.puts("#{starttime}")
file.puts("#{endtime}")
file.puts("SLICE")
file.puts("#{endtime}") #it might be necessary to change into multiple time slices

mc = OtMatrixCube.open
mat = mc[1,1,1420,1]
mat.export("#{$Ot.variantDirectory()}matrix.txt",TRIPS_ASCII)

file2 = File.open("#{$Ot.variantDirectory()}matrix.txt", "r")
header = file2.gets
header = file2.gets
header = file2.gets
header = file2.gets
header = file2.gets
while (!file2.eof?)
        text = file2.gets
        linedata = text.split(" ")
        orig = linedata[0]
        dest = linedata[1]
        demand = linedata[2]
        file.puts("#{orig}        #{dest} #{demand}")
end
file.close
file2.close
```

ImportDynameqNetwork.rb

```
#This job imports a Dynameq network into Indy

require "fileutils"
require 'win32ole'

#parameters
purpose = 1
mode = 1
time = 1000
minconnectornr = 100000
convert = 1.609344 #conversion for miles per hour to km per hour
connectortype = 31
path = "C:\\users\\msnelder\\Bakersfield DEMO\\Scenarios\\BaseNet_v16_suc\\"
scenario = "BaseNet_v16_suc"

#delete tables
deletesql = OtQuery.new("delete from '#{$Ot.projectDirectory()}point'")
deletesql.execute

deletesql = OtQuery.new("delete from '#{$Ot.projectDirectory()}line'")
deletesql.execute

deletesql = OtQuery.new("delete from node")
deletesql.execute

deletesql = OtQuery.new("delete from centroid")
deletesql.execute

deletesql = OtQuery.new("delete from link")
deletesql.execute

deletesql = OtQuery.new("delete from link1_1data2")
deletesql.execute

deletesql = OtQuery.new("delete from link2_1data1")
deletesql.execute

deletesql = OtQuery.new("delete from link3_1data1")
deletesql.execute

def findcentroidnr(centroid, path, scenario)
        file_network = File.open("#{path}#{scenario}-base", "r")
        inCentroids = false
        stop = false
        nrcentroids = 0
        while (stop == false)
                text = file_network.gets
                linedata1 = text.split("\n")
                linedata = linedata1[0].split(" ")
                if (linedata[0] == "CENTROIDS")
                        inCentroids = true
                        text = file_network.gets
                elsif (linedata[0] =="LINKS")
                        stop = true
                        writeln("centroid ", centroid, " not found")
                elsif (inCentroids ==true)
                nrcentroids += 1
                if (linedata[0].to_i == centroid)
                    stop = true
                    centroidnr = nrcentroids
                end
            end
        end
        file_network.close
        return centroidnr
end

file_network = File.open("#{path}#{scenario}-base", "r")
inNodes = false
inCentroids = false
inLinks = false
inConnectors = false
stop = false
links = 0
reverselinks = 0
nodes = 0
connectors = 0
centroids = 0
while (stop == false)
        text = file_network.gets
        linedata = text.split(" ")
        if (linedata[0] =="NODES")
            inNodes = true
            inCentroids = false
            inLinks = false
            inConnectors = false
            text = file_network.gets
        elsif (linedata[0] =="CENTROIDS")
            inNodes = false
            inCentroids = true
            inLinks = false
            inConnectors = false
            c = 0
            text = file_network.gets
        elsif (linedata[0] =="LINKS")
            inNodes = false
            inCentroids = false
            inLinks = true
            inConnectors = false
            text = file_network.gets
        elsif (linedata[0] =="CONNECTORS")
            inNodes = false
            inCentroids = false
            inLinks = false
            inConnectors = true
            text = file_network.gets
        elsif (linedata[0] =="MOVEMENTS")
            stop = true
        elsif (inNodes == true)
            id = linedata[0].to_i
            x = linedata[1].to_f
            y = linedata[2].to_f
```

```
				#writeln("node: ", id, " ", x, " ", y)

				sql_insert = OtQuery.new("INSERT INTO '#{$Ot.projectDirectory()}point' VALUES
('#{id}',2, '#{x}', '#{y}',0)")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO node VALUES ('#{id}')")
				sql_insert.execute

				nodes+=1

			elsif (inCentroids == true)
				id = linedata[0].to_i
				x = linedata[2].to_f
				y = linedata[3].to_f
				c += 1
				id = c
				#writeln("centroid: ", id, " ", x, " ", y)

				sql_insert = OtQuery.new("INSERT INTO '#{$Ot.projectDirectory()}point' VALUES
('#{id}',1, '#{x}','#{y}',0)")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO centroid VALUES ('#{id}',0)")
				sql_insert.execute

				centroids += 1
			elsif (inLinks == true)
				id = linedata[0].to_i
				fnode = linedata[1].to_i
				tnode = linedata[2].to_i
				reverse = linedata[3].to_i
				type = linedata[5].to_i
				length = linedata[6].to_f*convert #ToDo: length = -1
				vfree = linedata[7].to_f*convert
				capacity = linedata[9].to_f
				lanes = linedata[10].to_i

				capacity = lanes*capacity

				#writeln("link: ", id, " ", fnode, " ", tnode, " ",reverse, " ",type, " ",length, " ",vfree, "
",capacity, " ",lanes)

				#if (reverse == -1) or (reverse > id)
				sql_insert = OtQuery.new("INSERT INTO '#{$Ot.projectDirectory()}line' VALUES
('#{id}',1,'#{fnode}',2,'#{tnode}',2,0)")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link VALUES ('#{id}','#{length}')")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link1_1data2 VALUES ('#{id}',1,'#{lanes}')")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link2_1data1 VALUES
('#{id}',3,1,'#{type}')")
				sql_insert.execute
			sql_insert = OtQuery.new("INSERT INTO link3_1data1 VALUES
('#{id}','#{mode}','#{time}',1,'#{vfree}','#{capacity}','#{vfree}','#{capacity}','#{vfree}')")
			sql_insert.execute
			#else
			#	sql_insert = OtQuery.new("INSERT INTO link2_1data1 VALUES
('#{reverse}',3,2,'#{type}')")
			#	sql_insert.execute
			#
			#	sql_insert = OtQuery.new("INSERT INTO link1_1data2 VALUES
('#{id}',2,'#{lanes}')")
			#	sql_insert.execute
			#
			#	sql_insert = OtQuery.new("INSERT INTO link3_1data1 VALUES
('#{reverse}','#{mode}','#{time}',2,'#{vfree}','#{capacity}','#{vfree}','#{capacity}','#{vfree}')")
			#	sql_insert.execute
			#
			#	reverselinks +=1
			#end

			links += 1
		elsif (inConnectors == true)
			id = linedata[0].to_i+minconnectornr
			fnode = linedata[1].to_i
			tnode = linedata[2].to_i
			type = connectortype
			length = linedata[6].to_f*convert #ToDo: length = -1
			vfree = linedata[7].to_f*convert
			capacity = linedata[9].to_f
			lanes = linedata[10].to_i

			capacity = lanes*capacity
			#writeln("link: ", id, " ", fnode, " ", tnode, " ", type, " ",length, " ",vfree, " ",capacity, "
",lanes)

			if (fnode < 0)
				fnode = -fnode
				fnode = findcentroidnr(fnode, path, scenario)

				sql_insert = OtQuery.new("INSERT INTO '#{$Ot.projectDirectory()}line'
VALUES ('#{id}',1,'#{fnode}',1,'#{tnode}',2,0)")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link VALUES ('#{id}','#{length}')")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link1_1data2 VALUES
('#{id}',1,'#{lanes}')")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link2_1data1 VALUES
('#{id}',3,1,'#{type}')")
				sql_insert.execute

				sql_insert = OtQuery.new("INSERT INTO link3_1data1 VALUES
('#{id}','#{mode}','#{time}',1,'#{vfree}','#{capacity}','#{vfree}','#{capacity}','#{vfree}')")
				sql_insert.execute
```

```
                        connectors +=1
                elsif (tnode < 0)
                        tnode = -tnode
                        tnode = findcentroidnr(tnode, path, scenario)

                        sql_insert = OtQuery.new("INSERT INTO '#{$Ot.projectDirectory()}line'
VALUES ('#{id}',1,'#{fnode}',2,'#{tnode}',1,0)")
                        sql_insert.execute

                        sql_insert = OtQuery.new("INSERT INTO link VALUES ('#{id}','#{length}')")
                        sql_insert.execute

                        sql_insert = OtQuery.new("INSERT INTO link1_1data2 VALUES
('#{id}',1,'#{lanes}')")
                        sql_insert.execute

                        sql_insert = OtQuery.new("INSERT INTO link2_1data1 VALUES
('#{id}',3,1,'#{type}')")
                        sql_insert.execute

                        sql_insert = OtQuery.new("INSERT INTO link3_1data1 VALUES
('#{id}','#{mode}','#{time}',1,'#{vfree}','#{capacity}','#{vfree}','#{capacity}','#{vfree}')")
                        sql_insert.execute

                        connectors +=1
                end
        end
end

writeln("links: ", links)
writeln("reverse links: ", reverselinks)
writeln("nodes: ", nodes)
writeln("centroids: ", centroids)
writeln("connectors: ", connectors)
```

ImportDynameqMatrix.rb

```
#This job imports a Dynameq matrix in Indy

#parameters
mode = 1
centroidtype = 1            #pointtype of nodes that are centroides
normalnodetype = 2                  #pointtype of nodes that aren't centroides
nrheaders_Dynameq = 5
nrheaders_indy = 5

path = "C:\\Maaike\\Dynameq_1_43\\Bakersfield DEMO\\Scenarios\\BaseNet_v16\\"
scenario = "BaseNet_v16"
file_in = []
pcu = []
matrix = []
matrix << "amcar2"
matrix << "amtruck"
nrmatrices = 2
pcu << 1
pcu << 1
starttimeIndy = 1420
timeinterval = 30

def findcentroidnr(centroid, path, scenario)
        file_network = File.open("#{path}#{scenario}-base", "r")
        inCentroids = false
        stop = false
        nrcentroids = 0
        while (stop == false)
                text = file_network.gets
                linedata = text.split(" ")
                if (linedata[0] =="CENTROIDS")
                        inCentroids = true
                        text = file_network.gets
                elsif (linedata[0] =="LINKS")
                        stop = true
                        writeln("centroid ", centroid, " not found")
                elsif (inCentroids ==true)
                        nrcentroids += 1
                        if (linedata[0] == centroid)
                                stop = true
                                centroidnr = nrcentroids
                        end
                end
        end
        file_network.close
        return centroidnr
end


0.upto(nrmatrices-1) do |m|
        file_in[m] = File.open("#{path}#{matrix[m]}","r")
end

0.upto(nrmatrices-1) do |m|
        1.upto(nrheaders_Dynameq) do |h|
                header = file_in[m].gets
        end
        text = file_in[m].gets
        linedata = text.split(" ")
        starttime = linedata[0]

        text = file_in[m].gets
        linedata = text.split(" ")
        endtime = linedata[0]

        timeslice = 0
        while (!file_in[m].eof?)
                text = file_in[m].gets
                linedata = text.split(" ")
                if (linedata[0] == "SLICE")
                        timeslice += 1
                        file_out = File.open("#{$Ot.variantDirectory()}matrix#{timeslice}_#{m}.txt",
"w")
                        1.upto(nrheaders_indy) do |h|
                                header = file_out.puts
                        end
                        text = file_in[m].gets
                        linedata = text.split(" ")
                        #writeln("time", text)
                        #time = starttimeIndy - timeinterval + (linedata[0].to_i-starttime)
                else
                        #fixed format orig: 5; dest: 5; trips: 15
                        o = linedata[0]
                        d = linedata[1]
                        orig = findcentroidnr(o,path, scenario)
                        dest = findcentroidnr(d,path, scenario)

                        trips = linedata[2]
                        #writeln("line", text)

                        origlength = orig.to_s.length
                        origstring = ""
                        1.upto(5-origlength) do |j|
                                origstring = origstring + " "
                        end
                        origstring = origstring + orig.to_s

                        destlength = dest.to_s.length
                        deststring = ""
                        1.upto(5-destlength) do |j|
                                deststring = deststring + " "
                        end
                        deststring = deststring + dest.to_s

                        tripslength = trips.to_s.length
                        tripsstring = ""
                        1.upto(15-tripslength) do |j|
                                tripsstring = tripsstring + " "
                        end
                        tripsstring = tripsstring + trips.to_s

                        file_out.puts("#{origstring}#{deststring}#{tripsstring}")
                end
        end
end

#mc = OtMatrixCube.open
#mat = mc[1,1,1420,1]
#mat.export("#{$Ot.variantDirectory()}matrix.txt",TRIPS_ASCII)
```

ImportDynameqPaths.rb

```
#This jobs imports Dynameq Paths and paths flows into Indy

require "fileutils"
require 'win32ole'
#Maaike Snelder 6 July 2009

#This job imports the pathresults of Dynameq into Indy

#!!!!!!!!! adjust the parameters if needed
starttime = 1000
timestep = 30
p = 1
m = 1
u = 1
importPaths = true
importPathCost = true
minconnectornr = 100000
path = "C:\\users\\msnelder\\Bakersfield DEMO\\Scenarios\\BaseNet_v16_SUC_unsignalized\\"
scenario = "BaseNet_v16_SUC_unsignalized"

def findcentroidnr(centroid, path, scenario)
        file_network = File.open("#{path}#{scenario}-base", "r")
        inCentroids = false
        stop = false
        nrcentroids = 0
        while (stop == false)
                text = file_network.gets
                linedata1 = text.split("\n")
                linedata = linedata1[0].split(" ")
                if (linedata[0] == "CENTROIDS")
                        inCentroids = true
                        text = file_network.gets
                elsif (linedata[0] =="LINKS")
                        stop = true
                        writeln("centroid ", centroid, " not found")
                elsif (inCentroids ==true)
                        nrcentroids += 1
                        if (linedata[0].to_i == centroid)
                                stop = true
                                centroidnr = nrcentroids
                        end
                end
        end
        file_network.close
        return centroidnr
end

def findODflow(p, m, t, u, origin, destination, timestep)
        mc = OtMatrixCube.open
        matrix = mc[p,m,t,u]
        flow = matrix.get(origin,destination)
        flow = flow*timestep/60
        return flow
end

def findconnector(id,path,scenario)
```

```
        file_network = File.open("#{path}#{scenario}-base", "r")
        inConnectors = false
        stop = false
        connector = -1
        while (stop == false)
                text = file_network.gets
                linedata = text.split(" ")
                if (linedata[0] =="CONNECTORS")
                        inConnectors = true
                elsif (linedata[0] =="MOVEMENTS")
                        stop = true
                elsif (inConnectors == true)
                        if(linedata[4].to_i == id)
                                stop = true
                                connector = linedata[0].to_i
                        end
                end
        end
        return connector
end

def writePaths(text,origin, destination,path,scenario,minconnectornr)
        linedata = text.split(" ")
        lengthPath = linedata.length
        pathnr = linedata[0]
        sql_insert = OtQuery.new("INSERT INTO path VALUES ('#{origin}','#{destination}','#{pathnr}')")
        sql_insert.execute
        1.upto(lengthPath-1) do |i|
                linkid = linedata[i].to_i
                if (i==1)  or (i== lengthPath-1)
                        linkid        =        findconnector(linkid,path,scenario)        +        minconnectornr
#findconnector(linkid,path,scenario)
                end
                #if (linkid != -1)
                #       link = (linkid/10).round
                        direction = 1
                        sql_insert      =      OtQuery.new("INSERT      INTO      pathlinks      VALUES
('#{origin}','#{destination}','#{pathnr}','#{i}','#{linkid}','#{direction}')")
                        sql_insert.execute
                #end
        end
end

def writePathCost(text,origin, destination,time,mode, timestep,u,p)
        linedata = text.split(" ")
        lengthFlow = linedata.length
        traveltime = 0
        tollcost =0
        pathcost = 0
        pathlength = 0
        1.upto(lengthFlow-1) do |i|
                path_fract = linedata[i]
                path_fract2 = path_fract.split(":")
                pathnr = path_fract2[0]
                fraction = path_fract2[1].to_f
                flow = findODflow(p, mode, time, u, origin, destination, timestep)
                departureflow = fraction*flow
                if (departureflow < 0.0001)
                        departureflow = 0
                end
                #text                                                                                    =
"'#{origin}','#{destination}','#{pathnr}','#{time}','#{mode}','#{traveltime}','#{tollcost}','#{pathcost}','#{de
partureflow}','#{pathlength}'"
                #writeln(text)
                sql_insert      =      OtQuery.new("INSERT      INTO      pathcost      VALUES
('#{origin}','#{destination}','#{pathnr}','#{time}','#{mode}','#{traveltime}','#{tollcost}','#{pathcost}','#{dep
artureflow}','#{pathlength}')")
                sql_insert.execute
        end
end

if (importPaths == true)
        deletesql = OtQuery.new("delete from path")
        deletesql.execute

        deletesql = OtQuery.new("delete from pathlinks")
        deletesql.execute
end

if (importPathCost == true)
        deletesql = OtQuery.new("delete from pathcost")
        deletesql.execute
end

file = File.open("#{path}AvgVehicle.path", "r")
text = file.gets #header
inOD = false
inPath = false
inFlow = false
prev_orig = 0
prev_dest = 0
while (!file.eof?)
        text = file.gets
        linedata = text.split("\n")
        linedata = linedata[0].split(" ")
        if (linedata[0] == "<od>")
                inOD = true
                inPath= false
                inFlow = false
        elsif  (linedata[0] == "<path>")
                inOD = false
                inPath= true
                inFlow = false
        elsif  (linedata[0] == "<flow>")
                inOD = false
                inPath= false
                inFlow = true
                interval = 0
        else
                if (inOD)
                        o = linedata[0].to_i
                        d = linedata[1].to_i
                        if (o != prev_orig)
                                writeln("origin: ", o)
                end
                prev_orig = o
                origin = findcentroidnr(o, path, scenario)
                destination = findcentroidnr(d, path, scenario)
            elsif (inPath)
                if (importPaths == true)
                    writePaths(text, origin, destination,path,scenario,minconnectornr)
                end
            elsif (inFlow)
                if (importPathCost == true)
                    time = starttime + (interval)*timestep
                    interval = interval +1
                    writePathCost(text, origin, destination, time, m, timestep,u,p)
                end
            else
                writeln("error in file specification")
            end
    end
end
file.close

writeln("done!")
```

ImportDynameqResults.rb

```
#This job import Dynameq Results into OmniTRANS such that they can be compared with Indy results.

require "fileutils"
require 'win32ole'

#parameters
starttime = 1000
timestep = 5                    #minutes
purpose = 1
mode = 1
user = 1
result = 1
iteration = 1
transitlinenr = 0
path = "C:\\users\\msnelder\\Bakersfield DEMO\\Scenarios\\BaseNet_v16_suc_unsignalized\\"
scenario = "BaseNet_v16_suc_unsignalized"
convert = 1.609344

def findcentroidnr(node,path,scenario)
	file_network = File.open("#{path}#{scenario}-base", "r")
	nheaders =5
	centroid = -1
	1.upto(nheaders) do |i|
		text = file_network.gets
	end
	inNodes = true
	stop = false
	nodefound = false
	while (stop == false)
		text = file_network.gets
		linedata = text.split(" ")
		if (linedata[0] =="CENTROIDS")&(nodefound == false)
			stop = true
		elsif (linedata[0] =="CENTROIDS")&(nodefound == true)
			inCentroids = true
			inNodes = false
		elsif (linedata[0] =="LINKS")
			stop = true
		elsif (inNodes == true)
			if(linedata[0] == node)
				x = linedata[1]
				y = linedata[2]
				nodefound = true
			end
		elsif (inCentroids == true)
			if (linedata[2] == x)&(linedata[3] == y)
				centroid = linedata[0]
				stop = true
			end
		end
	end
	return centroid
end

#delete result table
deletesql = OtQuery.new("delete from link5_2data1")
```

```
deletesql.execute

file_density = File.open("#{path}link_adensity.out", "r")
file_inflow = File.open("#{path}link_aflowi.out", "r")
file_outflow = File.open("#{path}link_aflowo.out", "r")
file_speed = File.open("#{path}link_aspeed.out", "r")
while (!file_density.eof?)
        text_density = file_density.gets
        text_inflow = file_inflow.gets
        text_outflow = file_outflow.gets
        text_speed = file_speed.gets
        linedata_density = text_density.split(" ")
        linedata_inflow = text_inflow.split(" ")
        linedata_outflow = text_outflow.split(" ")
        linedata_speed = text_speed.split(" ")
        fnode = linedata_density[0]
        tnode = linedata_density[1]
        #find linknr and direction
        found = true
        sql = OtQuery.new("SELECT linenr, pointnra, pointnrb, pointtypea, pointtypeb " +
        "FROM '#{$Ot.projectDirectory()}line' " +
        "WHERE pointtypea = 2 and pointtypeb = 2 and pointnra = #{fnode} and pointnrb = #{tnode}")
        sql.open
        if (sql.recordCount == 1)
                record = sql.get()
                linknr = record[0]
                direction = 1
        else
                sql = OtQuery.new("SELECT linenr, pointnra, pointnrb, pointtypea, pointtypeb " +
                "FROM '#{$Ot.projectDirectory()}line' " +
                "WHERE pointtypea = 2 and pointtypeb = 2 and pointnra = #{tnode} and pointnrb =
#{fnode}")
                sql.open
                if (sql.recordCount == 1)
                        record = sql.get()
                        linknr = record[0]
                        direction = 2
                else
                        centroid = findcentroidnr(fnode,path,scenario)
                        if (centroid != -1)
                                        sql = OtQuery.new("SELECT linenr, pointnra, pointnrb,
pointtypea, pointtypeb " +
                                        "FROM '#{$Ot.projectDirectory()}line' " +
                                        "WHERE pointtypea = 1 and pointtypeb = 2 and pointnra =
#{centroid} and pointnrb = #{tnode}")
                                        sql.open
                                        if (sql.recordCount == 1)
                                                record = sql.get()
                                                linknr = record[0]
                                                direction = 1
                                        end
                        else
                                centroid = findcentroidnr(tnode,path,scenario)
                                if (centroid != -1)
                                        sql = OtQuery.new("SELECT linenr, pointnra, pointnrb,
pointtypea, pointtypeb " +
                                        "FROM '#{$Ot.projectDirectory()}line' " +
                                        "WHERE pointtypea = 1 and pointtypeb = 2 and pointnra =
#{centroid} and pointnrb = #{fnode}")
                                        sql.open
                                        if (sql.recordCount == 1)
                                                record = sql.get()
                                                linknr = record[0]
                                                direction = 2
                                        end
                                else
                                        writeln("record ", fnode, "-", tnode, " not found!")
                                        found = false
                                end
                        end
                end
        end
        sql.close

        if (found == true)
                sql_maxspeed = OtQuery.new("SELECT freespeed " +
                "FROM '#{$Ot.variantDirectory()}link3_1data1.DB' " +
                "WHERE linknr = #{linknr} and direction = #{direction}")
                sql_maxspeed.open
                record_maxspeed = sql_maxspeed.get()
                maxspeed = record_maxspeed[0].to_f
                sql_maxspeed.close

                0.upto(linedata_density.length-1) do |t|
                        time = starttime + t*timestep
                        density = linedata_density[t+2].to_f/convert
                        inflow = linedata_inflow[t+2]
                        outflow = linedata_outflow[t+2]
                        calcspeed = linedata_speed[t+2].to_f*convert
                        load = 0
                        cost = 0
                        queued = 0
                        if (maxspeed >0)
                                speedratio = calcspeed/maxspeed
                        else
                                speedratio = 1
                        end
                        sql_insert = OtQuery.new("INSERT INTO link5_2data1 VALUES
('#{linknr}','#{purpose}', '#{mode}',
'#{time}','#{user}','#{result}','#{iteration}','#{direction}','#{transitlinenr}','#{load}','#{cost}','#{calcspeed}'
,'#{queued}','#{density}','#{inflow}','#{outflow}','#{speedratio}')")
                        sql_insert.execute
                end
        end
end
file_density.close
file_speed.close
file_outflow.close
file_inflow.close
```

ImportLengths.rb

```
#This job import the lengths of a Dynameq network into Indy (only those with -1 in the network file)

require "fileutils"
require 'win32ole'

#parameters
#minconnectornr = 100000
convert = 1.609344 #conversion for miles to km
path = "C:\\users\\msnelder\\Bakersfield DEMO\\Scenarios\\BaseNet_v16_suc\\"

file_network = File.open("#{path}length2", "r")
text = file_network.gets #header
i = 0
while (!file_network.eof?)
        text = file_network.gets
        linedata = text.split("\n")
        linedata = linedata[0].split(" ")
        if (linedata[1].to_f > 0)
                i = i +1
                id = linedata[0].to_i
                length = linedata[1].to_f*convert
                sql_update = OtQuery.new("UPDATE link SET link.'length' = #{length} WHERE
link.'linknr' = #{id}")
                sql_update.execute
        end
end
writeln(i, " link lengths updated")
```