A comparison study for shortest-path queries over heterogenous spacial networks

Nichlas Bo Nielsen

June 13, 2016

AALBORG UNIVERSITET
STUDENTERRAPPORT

Title:

A comparison study for shortest-path queries over heterogenous spacial networks

Project period:
P10, Spring Semester 2016

Project Group:
mi101f16

Authors:
Nichlas Bo Nielsen

Supervisor:
Xike Xie

Total Pages: - 23 pages

Appendix: - 0

Completion date: 13-06-2016

Abstract::

This project uses a generator capable of integrating a road network with an indoor topology. The generator is also capable of generating moving objects used to simulate the behavior of people living in a city. To generate said objects many paths must be calculated, requiring an efficient and scalable shortest path algorithm. This project aims to do a comparison study over various algorithms for shortest path queries, which is to be used in the generator, as well as offer insights for said algorithms.

Contents

Chapter 1

Introduction

The task of finding the shortest path from one vertex to another is a very old and well studied problem. One of the reasons for this problem to be so popular is its many applications, such as road navigation and various map services.

In the last couple of years, these navigational services have become more common for people to have and is no longer only used in road vehicles. Today it is considered a standard functionality of smartphones, meaning that many people walk around with a navigational capable device in their pockets. Because of the continually rising popularity of smartphones, it would not be unrealistic to think that in some years it might be relevant to think about using them for navigation inside buildings. This could especially be useful in places such as large office buildings or shopping malls.

Since the shortest-path problem is so old, there have been developed many different methods of solving this problem. There are now methods that, with a bit of preprocessing, have very low query times on very large datasets. Some of these methods have been developed specifically for road networks and exploit the fact that these networks, for the most part, are completely planar.

In this report I will explore some of these modern pathfinding techniques and test them using a road network that has been modified to contain buildings. Since buildings can have multiple floors, the network will not be planar, it is thus the purpose of this report to see if these techniques still work well when non-planar elements are present.

In Section 1.1 i will explain the background and motivation for this project and Section 1.2 will provide an outline of the rest of the report.

1.1 Background

In order to understand the motivation behind this project it is first necessary to introduce the generator, in which these modern pathfinding algorithms could be used. A brief introduction to the generator can be found in Section 1.1.1, while a more detailed explanation of the generator, along with some new extensions

will be presented in Chapter 2. The pathfinding algorithms will be briefly presented in Section 1.1.2 and a more detailed explanation of how they work will be presented in Chapter 3.

1.1.1 Generator

The generator to be used is based on the one from [3], which is an extension of the original generator from [1]. The original generator was used to generate moving objects on a spatial network. These moving objects could be seen as cars or other vehicles moving around on a road network. The objects have many advanced behaviors, e.g. if many moving objects is present on the same edge, each moving object would slow down, simulating a traffic jam and if possible an object will also attempt to avoid roads affected by traffic jams. These features, among other things, are part of the reason that this generator have been quite popular.

The generator was extended in [3] to be able to add an indoor topology to ones road network. For example if you have the road network of a city, this program will find each face of the network, place a building randomly chosen from a number of user defined building templates and then connect the door of that building to the nearest point on the road network. This extension gave the possibility for the moving objects to not only simulate the behaviors of cars, but also simulate people living in a city. It also changed the behaviour of the moving objects, so they moved from building to building simulating a persons workday.

The generator uses Dijkstra's algorithms for calculating the trajectories for the moving objects. The motivation for this project is to find a suitable modern pathfinding algorithm, which could provide a speed up over Dijkstra for the generation of the moving objects and their trajectories. Besides modernizing the generator, this project will also provide useful insight into how some of these modern road network-based pathfinding algorithms behave when used in a non-planar network.

1.1.2 Algorithms

The algorithms chosen for comparison belong to a group of algorithms called *vertex importance based*, meaning that algorithms are built upon the assumption that some vertices are more important than others in a road network. This could for example be vertices that represent the entrance of a highway, as such a vertex would tend to be accessed more frequently, than a vertex corresponding to a road junction in a small residential neighborhood. The algorithms chosen are *Contraction Hierarchies (CH)*[4] and *Transit Node Routing (TNR)*[2].

While both these algorithms are well researched when it comes to their performance on road networks, neither of them seem to have been used much on networks that are less planar than roads. Another goal of this report is to test how these algorithms will behave in a road network, which has an added indoor topology consisting of non-planar multi storied buildings. Since this is not what

the algorithms are originally developed for, this could give some challenges. An discussion of possible challenges as well as how CH and TNR work can be seen in Section 3.1 and Section 3.2 respectively.

1.2 Outline

The rest of the report will be structured as follows:

In Chapter 2 the generator's technical functionalities will be explained. This includes its original functions as well as the extensions.

Chapter 3 will explain the theory behind the chosen pathfinding algorithms. This will include the intuition behind the methods as well as an explanation of how they work. It will also include an analysis of how these algorithms will behave in a non-planar network setting.

Chapter 4 will contain the test results of the different algorithms. This will include the used space and time of the algorithms during preprocessing, as well as during shortest-path queries.

Finally Chapter 5 will conclude on the results of the project.

Chapter 2

UrbanGen

We have made a generator called UrbanGen. In this chapter the UrbanGen generator and its functionalities will be introduced in Section 2.1, as well as a brief explaination of its UI. Section 2.2 i will provide some motivation for using the generator by discussing an alternative solution to the problem with the generators slow pathfinding.

2.1 Functionalities and UI of the generator

UrbanGen is an extension of Thomas Brinkhoff's network-based generator of moving objects. UrbanGen extended the original generator by integrating a model of indoor topologies with the original road networks, combining this new functionality with all of the original functionalities. The primary functionality of the original generator was to generate the moving objects, parameterize the movement of the objects, as well as visualize the generated trajectories.

The UI of the generator is as seen in Figure 2.1. The UI can be divided into three parts a visualization window, a moving object management panel and a indoor topology management panel.

The visualization window, as the name suggest, visualizes the data loaded by the generator. Edges in this window have different colors depending on the edge class. An edge class is used by the generated moving objects to determine whether an edge is inside a building or not and also what type of road it is, which could be a highway or a street in a city among other things.

The moving object management panel contains a few buttons for navigating the visualization window, but primarily contains various parameters that can be set before generating the moving objects. It is here possible to set the maximum time that the moving objects can move, each object will move a certain distance on the path to their destination depending on their speed. It is also in this window that you can specify how many moving objects are to be generated. The scrollbar in the button of the panel is used to scroll through each time slot, which will cause the visualization window to show each moving object at their

current position depending on the time slot selected by the scrollbar.

The indoor topology management panel allows for the control of certain parameters regarding the moving objects behavior in buildings, as well as a maximum number of floors for the buildings generated. Other than that it also allows you to show or hide different building related edges, and to select which floor of the buildings that are to be visualized by using the scrollbar on the right.

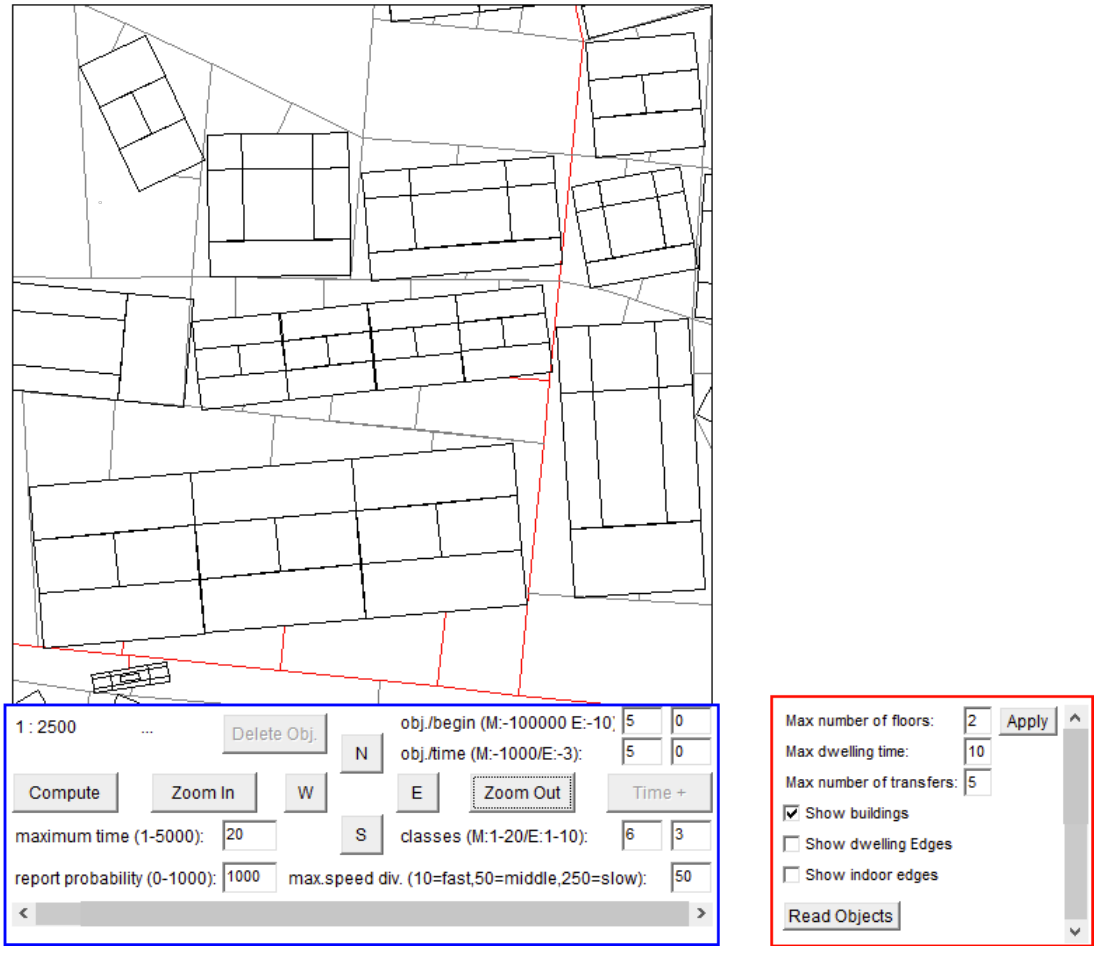

Figure 2.1: Main window of generator. Visualization window is the panel with the **black** border, moving object management panel is the one with blue and the indoor topology management panel is the one with a red border.

2.2 Motivation for using the generator

As mentioned earlier the generator currently uses Dijkstra's algorithm for calculating the trajectories of the moving objects. The problem with using Dijkstra together with the integrated indoor and outdoor topology is that it makes Dijkstra much slower, since it also has to consider a path through each building it meets.

To make Dijkstra more effective one could give it less paths to consider by not integrating the indoor and outdoor topology, but instead see them as separate graphs. Then when finding the shortest path, one could instead find the shortest path to the entrance of the building, containing the target vertex, and then when you are at the building, you could switch over to the indoor graph, find the shortest path from the door to the target vertex and then combine the two paths. This should make a good improvement of the query time of Dijkstra, especially when many buildings are present.

However, as seen in Figure 2.1, buildings are able to have multiple entrances and they can also be interconnected. So with this approach the path found, would not be guaranteed to be the shortest path, as the real shortest path might contain other buildings. This makes it necessary to use the integrated topology, when the goal is to find the shortest-path.

The approach, however, is still not entirely unusable, as when a moving object is on the road, they could function as a car, meaning that driving through a building would often not be possible. With that in mind a method that divides the indoor and outdoor topology could be useful. For this reason this method will also be tested in Chapter 4 using a Bidirectional Dijkstra's algorithm.

Chapter 3

Algorithm Background

In this chapter the two different pathfinding algorithms will be explained, both in terms of how they work and how they behave in a non-planar road network setting. Both algorithms have a preprocessing step and a particular way of querying, which will be explained. *Contraction Hierarchies (CH)* will be explained in 3.1 and *Transit Node Routing (TNR)* in 3.2. Even though a Dijkstra's algorithm based approach will also be part of the experiments, it is deemed unnecessary to explain how it works, as it is both well known and less sophisticated than CH and TNR.

3.1 Contraction Hierarchies

CH speeds up shortest-path queries by using a precomputed "contracted" version of the graph. Before the contracted version of the graph can be generated the nodes in the graph are first imposed a total order according to their relative importance. This ordering of nodes can be determined in different ways, [4] suggest several heuristic approaches for this.

3.1.1 Preprocessing

Once a node ordering have been determined, the *contraction* part of the precomputing step can begin. Assume that we have a graph G and in this graph CH has imposed a total order $v_1 < v_2 < \cdots < v_8$ on the vertices of G. CH will then go through each vertex, following the order, and investigate each vertex's neighboring vertices. It will then check if there exist two neighbors v_j and v_k, where the shortest path from v_j to v_k passes through v_i. If such a path exists, CH inserts an artificial edge c, called a *shortcut*, which connects the two neighbors v_j and v_k. This is continued till it has been done for all neighbors of v_i. The shortcuts are then tagged with the original vertex v_i and then v_i is removed from G. This process is what is known as *contraction*. The preprocessing is completed when all vertices are contracted. How the shortcuts are added

can be seen illustrated in Figure 3.1.

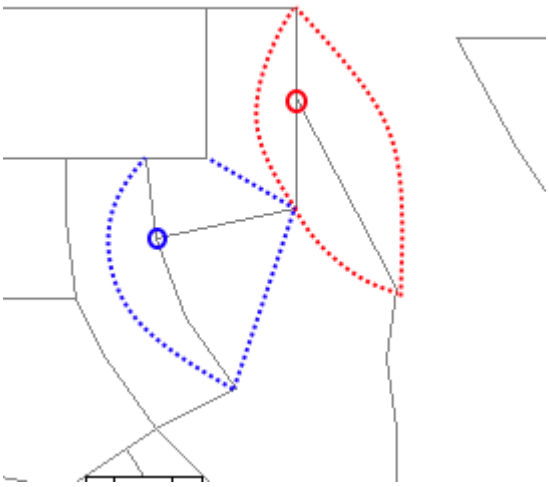

Figure 3.1: CH preprocessing. The red and blue vertex have both gotten their shortcuts added.

3.1.2 Querying

Querying with CH is done with a modified *Bidirectional Dijkstra's algorithm*. The difference being that CH only considers edges and shortcuts that connects a visited vertex to an unvisited vertex, which is higher in the total order. This results in a significantly quicker query time, when compared to the original Dijkstra's algorithm, since not as many nodes have to be visited. This modified Dijkstra's algorithm would not work on a network that has not been through the preprocessing step of adding shortcuts, as the shortcuts are added to represent the existing shortest paths that the algorithm would not take into consideration otherwise. Since these shortest paths can often contain shortcuts, it is also possible to derive the path containing only original edges and vertices, with no shortcuts. This is where the tag associated with each shortcut is used. If a shortcut c connects v_j and v_k and its tag is v_i, then c can be replaced with two edges (v_j, v_i) and (v_k, v_i).

3.1.3 CH in a non-planar road network

As mentioned in Section 3.1.1 the preprocessing step basically adds new edges and removes vertices from the network, and for querying CH uses a slightly modified Dijkstra's algorithm. None of these processes, both preprocessing and querying, require the algorithm to know the coordinates of the graph. Since

the coordinates of the vertices are not required for the algorithm to work, the algorithm does not know if the graph is planar or not. Taking that into consideration it should be safe to assume that the algorithm will work with a non-planar road network.

3.2 Transit Node Routing

TNR speeds up queries by imposing a grid on the road network and precomputing the shortest paths from within each grid cell C to a set of vertices called *access nodes* A. Each cell in the grid has an *inner* and *outer shell*, these shells are used to determine which vertices are part of A. A vertex is part of A if and only if the following two conditions are met. The first condition is that the vertex is an endpoint of an edge which intersects the inner shell of C. Secondly, any shortest path from a vertex in C to a vertex outside the outer shell must pass through at least one vertex in A. A visualization of TNR can be seen in Figure 3.2.

3.2.1 Preprocessing

For the preprocessing of TNR, the access nodes of each cell must first be determined. These can be determined by checking the two conditions previously mentioned.

When the access nodes are found TNR precomputes two sets of distance information. First it computes the distance from any vertex v to each access node in A of the cell that contains v. Secondly it computes the distance from any two access nodes of any two different cells.

3.2.2 Querying

When querying, TNR can use these precomputed distances to efficiently derive the distance between any two vertices s, t, if t lies beyond the outer shell of the cell containing s. TNR does this by finding the shortest total distance from s to an access node in A_s, then to an access node in A_t and finally to the vertex t. This can be written as seen in Equation 3.1, where $dist(s, t)$ is equivalent to the distance from s to t.

$$dist(s,t) = \min_{v_s \in A_s, v_t \in A_t} dist(s, v_s) + dist(v_s, v_t) + dist(v_t, t) \tag{3.1}$$

However, if t lies inside the outer shell of the cell containing s TNR can no longer use the precomputed distances to calculate the shortest path. If this is the case, other methods has to be used such as CH or bidirectional Dijkstra's algorithm.

The performance of TNR depends on the size of the grid imposed on the network. Smaller grids generally leads to higher space overhead, due to a larger number of access nodes, but it also allows more shortest path queries to be calculated using TNR.

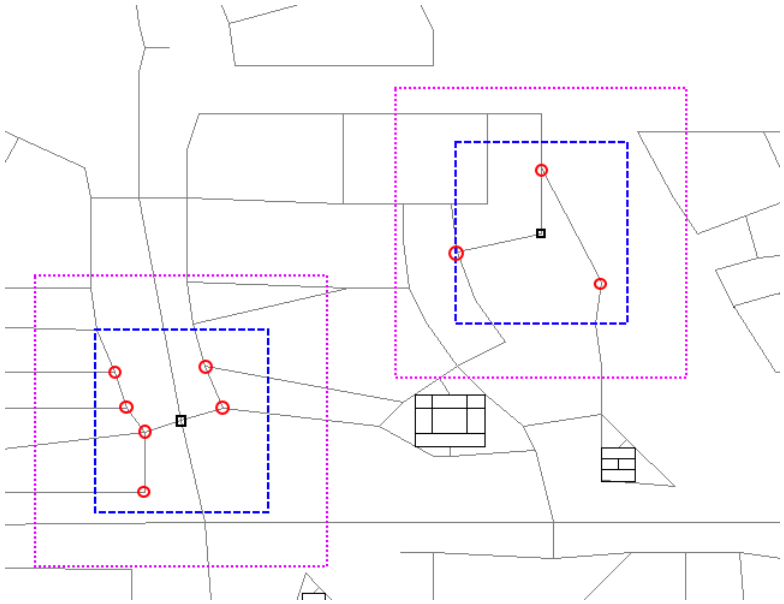

Figure 3.2: TNR preprocessing. C is the **black** rectangle, the blue rectanlge is the inner shell and the purple rectangle is the outer shell. The red circles are access nodes.

3.2.3 TNR in a non-planar road network

Unlike CH, TNR does require the coordinates of all vertices in the network, so TNR will know if the graph is planar or not. Another thing to consider with TNR is how to handle the fact that the graphs are not only non-planar, but they are also three-dimensional, because of the different floors of the buildings. For handling the extra dimension of the network i propose two different approaches.

The first approach is adding the extra dimension to all points in the network, so that the network is no longer in 2d space, but 3d space. The cells and shell of TNR could then be expanded to include this third dimension as well. This approach could add more flexibility to the algorithm, by giving it the possibility of only including a certain number of floors its shell. By limiting the number of floors included withing a shell, the graph could have a larger number of access nodes compared to a non-third-dimensional approach, eg. a 4 story building could have only 2 of its floors included in the outer shell. A larger amount of access nodes have the advantage that TNR is able to compute a larger variety of shortest-paths, without resorting to using CH or Dijkstra's algorithm, as the possibility of the target node being inside the outer shell of the source node is smaller. However, this also makes preprocessing slower, as more paths have to be precomputed, and this approach also require some modifications to the original TNR algorithm in order to work.

The second approach is to not consider the third dimension, and treat all

vertices as if they were at the same height overlapping each other. The advantages of this approach is that it is first of all easier to implement, as it does not require any modification of TNR. It will also have a shorter precomputation time than the other approach, as there will be less access nodes when all vertices in each building is considered to be on the same level. This also means that there is a bigger risk of TNR having to resort to Dijkstra's algorithm or CH for shortest-path queries, depending on how many buildings there are, how many floors they have and how close they are. If buildings are far apart, the risk of a source and target vertex of a shortest-path query being located in each others outer shells is lower. If the average floor number is low, the chances of a source-target pair being in each others outer shells is also lowered, as there would be less vertices within the same building. Finally if there are not many buildings the chances of a source-target pair of a shortest-path query being inside each others outer shell is again lowered.

I choose to use the second approach, both because it is easier to implement, and has faster preprocessing, but also because the first approach only really has any benefits if each building has a huge amount of floors.

The correctness of the TNR algorithm with the chosen approach should not be affected by a non-planar nature of the indoor/outdoor integrated graph. This does depend on which algorithm is used for precomputing the shortest paths which TNR uses for preprocessing. However, if we consider only Dijkstra or CH for these paths, then the precomputation of the shortest paths will work if CH is used, for the same reason explained in 3.1.3. While TNR might know if a graph is planar, the only thing the coordinates are used for is to determine access nodes and determine whether or not TNR is able to calculate the shortest path using the precomputed paths with Equation 3.1. Since all paths compared in Equation 3.1 are calculated using CH or Dijkstra, and these algorithms are not affected by the planarity of a graph, then we can conclude that TNR must not be affect by planarity either.

Chapter 4

Experiments

In this chapter several experiments will be conducted using CH and TNR. Before the experiments the setup for the tests will be explained in Section 4.1. The results will be presented in Section 4.2 and analyzed in Section 4.3.

4.1 Experimental Setup

This section will first provide some basic information about the computer used for testing and then present the query sets used for the performance tests as well as some basic informations about how the tests will be conducted.

First of all the computer used has the specifications seen in table 4.1

CPU	Intel Core i5-3570K 3.4GHz
Memory	4GB DDR3 1800MHz
Harddrive	Samsung SSD 850 PRO 256GB

Table 4.1: Test computer specifications

4.1.1 Dataset and Queryset

All tests will be performed using three different datasets, Oldenburg, San Joaquin and San Francisco Bay Area, which will be referred as Dataset 1, 2 and 3 respectively. All datasets have an indoor topology added by using the generator. To ensure the correctness of the paths found by CH and TNR their found paths, for all queries, will be compared to ensure that they are the same. The maximum number of floors for each dataset have been set to 2. The unmodified datasets are found on [1]. Additional information about the datasets can be seen in Table 4.2.

As seen in Table 4.2 a large percentage of both nodes and edges are located inside buildings. Keeping this in mind, as well as knowing that the moving objects will primarily be moving from building to building, it would make a

	Oldenburg	San Joaquin	San Francisco
Total num. of nodes	21,466	69,334	541,063
Total num. of edges	29,087	98,456	759,072
Num. of indoor nodes	15,008	50,533	378,194
Num. of indoor edges	20,353	69,134	514,012
Percentage of indoor nodes	70%	72%	70%
Percentage of indoor edges	70%	70%	67%

Table 4.2: Information about the three different used datasets

lot of sense for the majority of the query sets used for testing, to either have one node in a building or both the source and the target node located inside a building.

The query sets are largely chosen at random, though it is made sure that most of them have a node located in a building, because of the previously mentioned reasons. Information about the query sets can be seen in Table 4.3

	Oldenburg	San Joaquin	San Francisco
Set 1 % of both nodes indoor	48%	48%	53%
Set 2 % of both nodes indoor	50%	54%	47%
Set 1 % of one node indoor	42%	46%	40%
Set 2 % of one node indoor	41%	41%	45%

Table 4.3: Query set information. Two sets of two different sizes are chosen for each dataset. Set 1 has 100 nodes, set 2 has 500 nodes.

4.1.2 Implementation notes

The implementation of TNR and CH is based on the source code from [5], with a few modifications in order for their program to contain the same information about the vertices and edges located inside buildings. However, there were a few complications with the usage of this code, as i had to scale down the coordinates of the datasets in order to avoid overflow and i also had to make some minor modifications to the structure of the data in order to get it to work. It should also be noted that TNR uses the preprocessed network from CH, as this speeds up preprocessing for TNR and also makes it possible to use CH when TNR cannot find the shortest-path. For the tests the size of the grid for TNR will be 128×128.

For the tests with the divided indoor and outdoor topologies, explained in Section 2.2, the bidirectional Dijkstra's algorithm also found in the source code from [5] will be used. This method will use the same query sets as the other methods, except that each source or target node located inside a building, will instead be placed on the node originally connecting the building to the road, which is closest to the source or target node.

4.2 Results

This section contains the results of the experiments. The experiments consists of two primary categories preprocessing, in Section 4.2.1 and query tests, using the query sets seen in Table 4.3, in Section 4.2.2.

4.2.1 Preprocessing tests

The preprocessing test is divided into two categories, time and space consumption. The time used for both CH and TNR can be seen in Figure 4.1 and the space consumption can be seen in Figure 4.2.

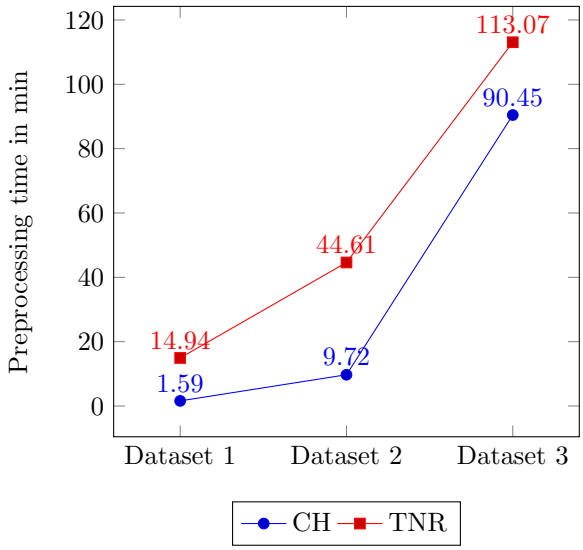

Figure 4.1: Preprocessing time for both CH and TNR on all three datasets

Looking at Figure 4.1 one can conclude that TNR preprocessing is slower than CH. This result is expected as the preprocessing step of TNR is more complex. It should also be noted that if TNR did not use CH for preprocessing, the difference between the two would probably be even higher.

Figure 4.2 again shows the clear difference between the two algorithms preprocessing steps. Depending on the granularity of the graph, the space consumption of TNR can vary greatly. A dataset being bigger than another does not guarantee that the space consumption will be larger with TNR, as seen with dataset 3.

4.2.2 Query tests

Figure 4.3 shows the average query time, with query set 2. Dijkstra is slower than the two other methods, though this is also expected since there is no

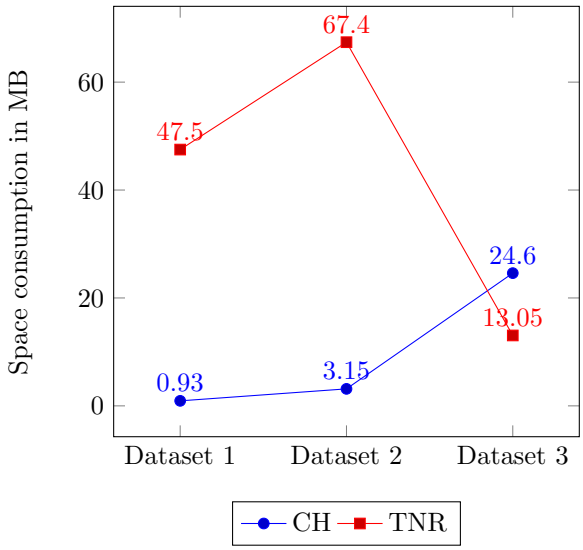

Figure 4.2: Space consumption after preprocessing for both CH and TNR on all three datasets

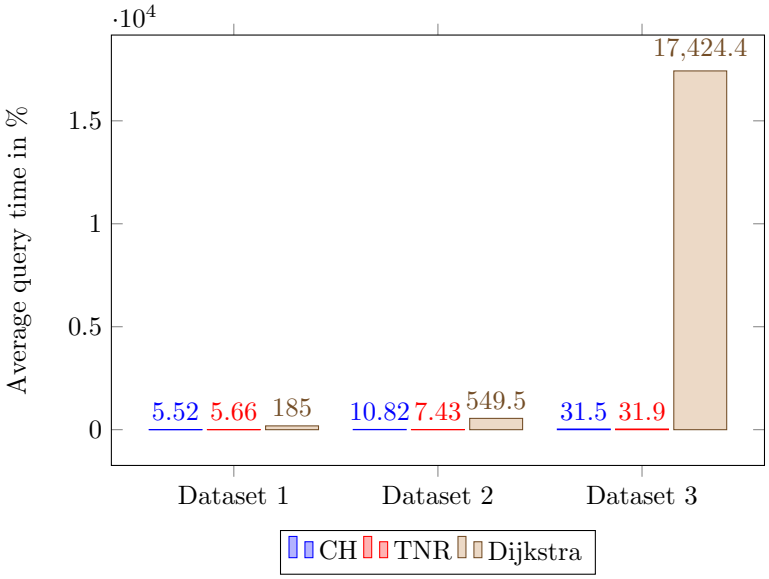

Figure 4.3: Average query time with query set 2

preprocessing. Preprocessing for Dijkstra is possible, though it would require the computation of all-pair shortest paths, which would require far too much space and time to be practical. Here we can also see that because the paths

calculated, by TNR for dataset 3, are so few, TNR will often end op using CH, which explains their similar performance. TNR does get faster with dataset 2, which is to be expected as it has more precomputed data to work with.

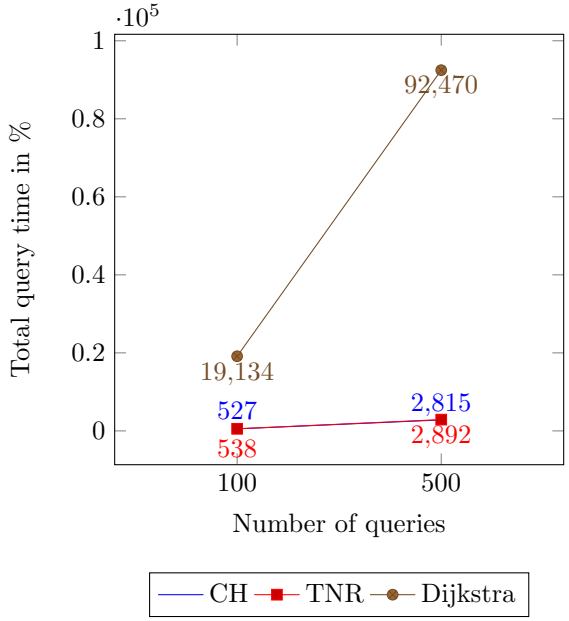

Figure 4.4: Total query time with different query sets using dataset 1

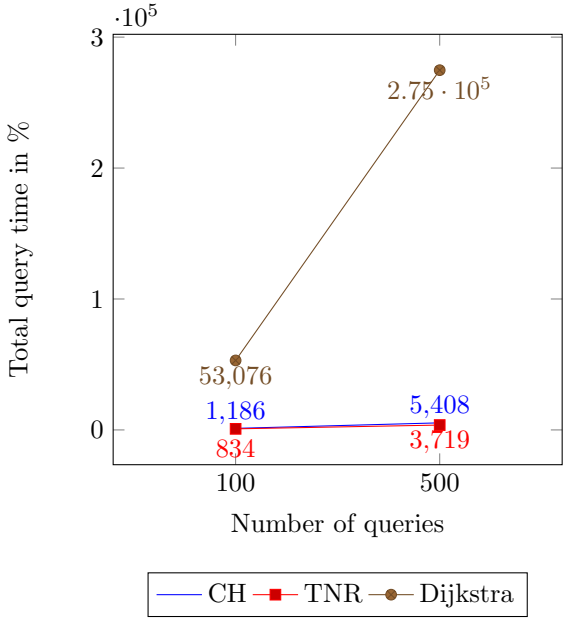

Figure 4.5: Total query time with different query sets using dataset 2

For Figure 4.4, Figure 4.5 and Figure 4.6 the same conclusions can be drawn as Figure 4.3.

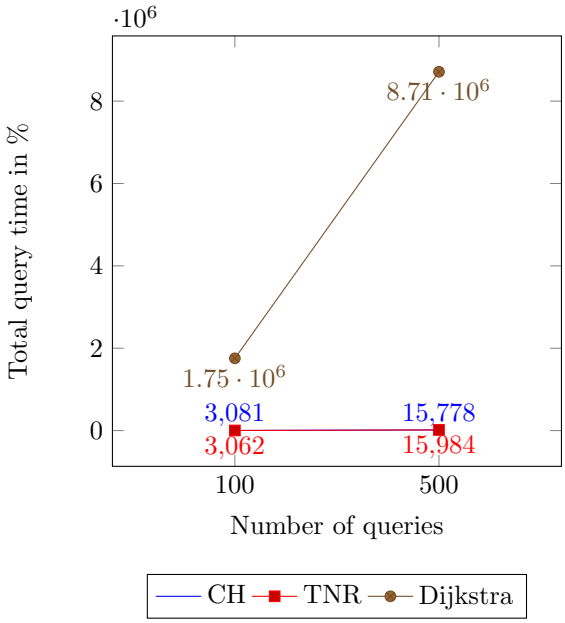

Figure 4.6: Total query time with differnt query sets using dataset 3

4.3 Experiments Conclusion

As seen in Section 4.2 both TNR and CH worked very well in a non-planar setting. Both algorithms are very suitable for use in the generator for pathfinding, though which one is preferred depends on data being used. For the dataset used in this report, the differences between the two algorithms are very small, because TNR would often end up using CH. This is even a problem with the low floor number that was used for these tests, and if one were to use a larger number of floors this problem will only be worse. However, if time and space is not an issue one might as well use TNR as it will either be faster than CH, or just as fast in the worst case.

As for the Dijkstra approach, even when separating indoor and outdoor topologies it is still much slower than CH and TNR. Considering the relative low preprocessing time of CH it would almost always be preferable to this method.

Chapter 5

Conclusion

5.1 Project Results

This project were to explore modern alternative pathfinding algorithms for use in UrbanGen, which is used to generate in-/outdoor trajectories for moving objects in an urban environment. The non-planar nature of the data used for testing in this project did not seem to have an effect on the performance of CH and TNR. The buildings does make the graph more dense, which hurts the performance for TNR. While both algorithms proved to be highly effective, the extra time and space spend on TNR did not seem to be worth it with the chosen data.

In the end a faster alternative to the current pathfinding method used by the generator was found, which could make generating trajectories much faster for the generator. Based on experiments it also seems like TNR and CH have no trouble with non-planar graphs.

5.2 Future Work

5.2.1 Test on bigger datasets

The datasets used in this project are relatively small and in order to fully realize the potential of highly effective algorithms such as CH and TNR larger datasets would have been a good idea to use. This would more clearly show how fast the algorithms truly are and it would also make the difference between the two more clearly.

5.2.2 Other algorithms

In the source code provided by [5] there were also two other algorithms, *Spatially Induced Linkage Cognizance (SILC)* and *Path-Coherent Pairs Decomposition*

(PCPD), that could have been explored. However, due to time constraints and some issues with getting the source code to work, this was not possible.

Bibliography

[1] Thomas Brinkhoff. Network-based generator of moving objects, 2005. URL http://iapg.jade-hs.de/personen/brinkhoff/generator/.

[2] Domagoj Matijevic, Holger Bast, Stefan Funke. Transit ultrafast shortest-path queries with linear-time preprocessing, 2006. URL http://people.mpi-inf.mpg.de/~dmatijev/papers/DIMACS06.pdf.

[3] Xike Xie, Nichlas Bo Nielsen, Torben Bach Pedersen, Ulf Simonsen, Hua Lu and Maite Ainciburu. Urbangen: Generating combined in/outdoor trajectories, 2015.

[4] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks, 2008. URL http://algo2.iti.kit.edu/schultes/hwy/contract.pdf.

[5] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu and Shuigeng Zhou. Shortest path and distance queries on road networks: An experimental evaluation, 2012. URL http://vldb.org/pvldb/vol5/p406_lingkunwu_vldb2012.pdf.