Shortest paths in dynamic road networks

Master's thesis

Dániel Góbor

Applied mathematics MSc.

Specialization in computer science

Supervisor:

Zoltán Király

Department of Computer Science

Eötvös Loránd University, Institute of Mathematics

Eötvös Loránd University, Budapest

Faculty of Science

2015

Contents

1 Introduction

With the advance of GPS technology and mobile internet access, it has become increasingly important to quickly and precisely answer shortest path queries in large graphs both on the GPS device and on servers that might be accessed by hundreds of people at the same time.

In 2005, with the beginning of the 9th DIMACS implementation challenge [22], numerous new methods were introduced with query times hundreds of times less than the existing classical algorithms for finding shortest paths. At the same time, the graphs of the European and American road networks became available for the public [10].

With the further advancement in technology, there are ways to measure the traffic flow of certain roads, so we have real time data of possible traffic jams and accidents. This means that the algorithms must be prepared to provide correct results even if the arc costs change.

We will see that there is no "best" algorithm. Some of them may provide good query times but require a large amount of RAM, while others may be easily adapted to the dynamic scenario, but provide smaller speedups.

Our aim was to examine a few algorithms, implement them using C++ and LEMON [20], and to test them under different conditions, including the changing of arc costs and simulating traffic jams.

In this section, we will describe the classical algorithms, since most of the new methods use them as a subroutine. We also examine the results of several papers that study the same methods we chose.

1.1 Dijkstra's algorithm

Dijkstra's algorithm [18], developed by computer scientist Edsger Dijkstra in 1956, is an algorithm for finding shortest paths from a node s to the other nodes of the graph $G = (V, E)$ given a cost function $c : E \to \mathbb{R}^+$.

1.1.1 Description of the algorithm

During the algorithm, we build a shortest paths tree B rooted at s. A node can belong to one of the following three sets:

- S: the *unvisited* nodes
- L: the *reached* nodes
- Q: the *finalized* nodes

Initially, $S = V \setminus \{s\}, L = \{s\}, Q = \emptyset$.

We assign a distance label $dist(v)$ to every node v. At the beginning of the algorithm, $dist(s) = 0$ and $dist(v) = \infty$ for every other node v.

We also store the parent pointers $p(v)$. Initially, $p(v) = null$ for every node v.

A typical step of the algorithm is when we finalize a *reached* node $v \in L$ whose distance label $dist(v)$ is minimal. Finalizing the node v means the following:

- v becomes *finalized*, that is $L = L \setminus \{v\}$ and $Q = Q \cup \{v\}$.
- we *relax* every edge $e = (v, w)$ by doing the following:
  - if w is *unvisited*, that is $w \in S$, then $S = S \setminus \{w\}$ and $L = L \cup \{w\}$.
  - if $dist(w) > dist(v) + c(v, w)$, we update the distance label of w, so the new label will be $dist(w) = dist(v) + c(v, w)$.
  - if we updated the distance label, we also update the parent pointer: $p(w) = v$.

When the algorithm terminates, the distance label of a *finalized* node v will be the cost of the shortest $P(s, v)$ path. The path can be obtained by traversing the parent pointers from v to s.

Note. The algorithm works on directed graphs without modification.

1.1.2 Finding the minimal element

During the algorithm we have to find a node $v \in L$ whose distance label $dist(v)$ is minimal. Using the naive approach, we find the minimal element in $O(|V|)$ time, so we get $O(|V|^2)$ for the total running time of Dijkstra's algorithm.

However, if we use a binary heap data structure by storing the *reached* nodes in the heap, we get a total running time of $O(|E| \log |V|)$, since every operation in a binary heap takes $O(\log n)$ time, where n is the number of elements in the heap.

Note. Instead of saying that a node v is reached, we may say it is in the heap.

Note. We will use the following terms throughout the thesis: finalize a node, relax an edge, finalized nodes. They all mean what we described above unless otherwise noted.

1.1.3 Bidirectional Dijkstra

We want to find the shortest path from node s to node t. We alternate between two Dijkstra algorithms. The forward search runs from s in the original graph, while the backward search runs from t on the reverse graph (the graph where the arcs are reversed). When a node v becomes finalized in both algorithms we stop. The shortest path will be either $P(s, v) \cup P(v, t)$ or there is an arc (u, w) so the shortest path will be $P(s, u) \cup \{(u, w)\} \cup P(w, t)$ where node u is finalized in the forward search and node w is finalized in the backward search.

1.2 The A* algorithm

The A* algorithm [16], proposed by Nils Nilsson, is an extension of Dijkstra's algorithm.

When performing an s, t query, similarly to Dikstra's algorithm, we have *unvisited*, *reached* and *finalized* nodes.

We also have a function $h : V \times V \to \mathbb{R}^+$ with the following attribute: $\forall (u, v) \in E \; h(v, t) - h(u, t) \leq c(u, v)$, that is, h is a consistent lower estimate for the distance between a node v and t.

We have a modified arc cost function c', where $c'(u, v) = c(u, v) + h(v, t) - h(u, t)$. To answer an s, t query, we run a simple Dijkstra algorithm on the graph from s using this modified cost function.

If $h \equiv 0$, then the A* algorithm is the same as Dijkstra's algorithm. If $h(u, v)$ is the cost of the shortest path between u and v, the algorithm will only finalize the nodes on the shortest path.

In road networks, if $c(u, v)$ is the distance we must travel, we can also use the Euclidean distance for $h(u, v)$.

By using the function h, we try to reduce the number of nodes we finalize, since the search will select nodes that are closer to the target node.

Similarly to Dijkstra's algorithm, there is a bidirectional version of the A* algorithm.

1.3 Methods with preprocessing

Most modern methods require a preprocessing phase before we can answer queries. In this phase we calculate same data, which we can use later to speed up the queries. The methods we examined all depend on finding "important" nodes. The importance of a node is determined by heuristics.

Most articles claim that the algorithm they describe is very efficient and thousands of times faster than Dijkstra's algorithm. These claim are true under certain conditions, however, we have to keep in mind that comparing methods are difficult, as the query speeds largely depend on the chosen s, t pairs and the structure of the graphs as well.

1.3.1 Highway Hierarchies

Highway Hierarchies (from now on HH) tries to take advantage of the hierarchy of road networks. Basically, the more shortest paths go through an edge, the higher it is in the hierarchy. We examine this method more closely in section 3.

In [11] we can clearly see that the method provides large speedup only with very distant s, t pairs in the USA and European graphs, which have 18 million nodes. An average query will only be a few hundred times faster than Dijkstra's algorithm.

In [12], they claim that the method is almost ten thousand times faster than Dijkstra's algorithm and that Dijkstra's algorithm finalizes 12000 times as many nodes.

1.3.2 Highway Node Routing

Highway Node Routing (HNR from now on) is very similar to Highway Hierarchies. In fact, in [1], they use the results of the preprocessing method of HH to obtain the level of the nodes. We examine this method very closely in section 4, since in the article, they claim that it can be easily adapted to the dynamic scenario. A more recent version of the same article is present in [8]. They only compare the method with HH in [8] and [13], and it turns out that the performance is almost the same.

In [2] they obtained the original implementation of HNR and integrated it into their application. The tests they performed show that HNR is at most a thousand times faster than Dijkstra's algorithm. However, the same measurements show that calculating a larger distance table yields a significantly larger speedup, which suggests that the original implementation contains additional functionality that is not mentioned in the original article.

1.3.3 Contraction Hierarchies

Contraction Hierarchies (from now on CH) is a special version of HNR that is faster and simpler [4]. We examined this method in the BSc thesis [19]. Section 2 contains a summary. We expanded our implementation to account for the dynamic case.

In the original article [4], they show that CH is 5-6 times faster than HNR.

In [5] they suggest that using the same "importance" of the nodes that was determined during the preprocessing can be used when arc costs change, so the preprocessing will be faster. We will examine this later in section 6.5.2.

In [7], they split the preprocessing into a first metric-independent and second fast metric-dependent phase, so when the arc costs change, we only have to repeat the second phase. They claim that this does not reduce the query times significantly.

1.3.4 Transit Node Routing

Calculating all pairs of shortest paths in the preprocessing phase is too expensive both in terms of time and memory required.

In a road network however, every node s has a set of access nodes $A(s)$ such that every long distance s, t shortest paths goes through a node in $A(s)$. Good examples are bridges on a river or the endpoints of the routes into towns.

We can select a suitable subset $T \subseteq V$ of transit nodes and calculate the node sets $A(v) \subseteq T$ for every node v. Finally, we create a distance table on the nodes of T and we store the shortest paths between v and the nodes of $A(v)$ for every v. Then we can answer a query with a few table lookups, that is $d(s,t) = min\{d(s,v) + d(v,w) + d(w,t) : v \in A(s), w \in A(t)\}$ [3].

We can combine Transit Node Routing with other methods. We can either select T as the set of most important nodes that the other method finds, or we can use the other method to run the search on the less important nodes and use the distance table of Transit Node Routing on the most important nodes.

We examined the combination of Transit Node Routing and CH in the BSc thesis [19], and expanded the implementation and tested it in section 6.4.1.

1.4 Summary

A summary of the different methods can be found in [9]. The following figure, taken from the article, shows preprocessing times and speedup of the different techniques.

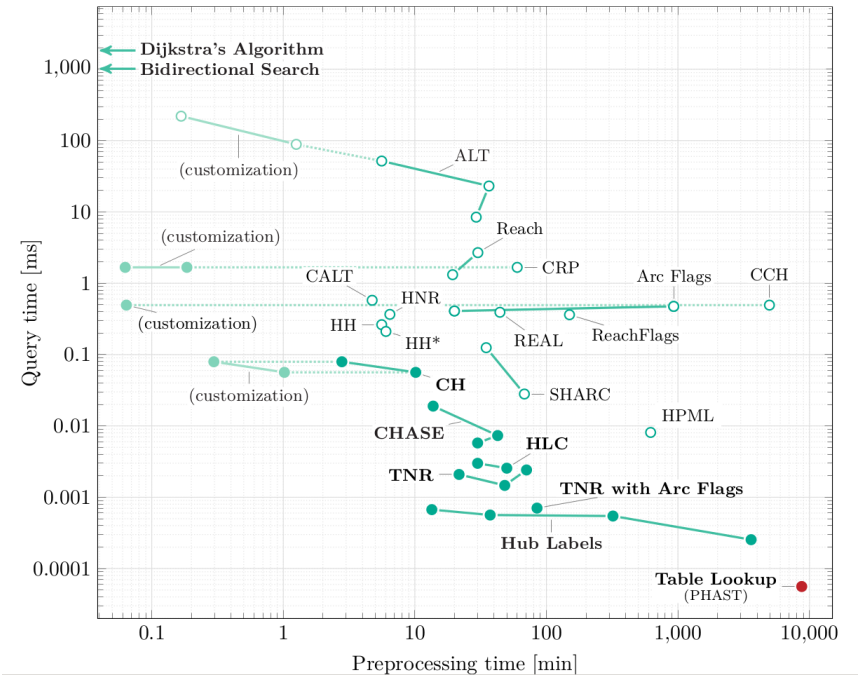

Figure 1: Summary of the different techniques taken from [9]

As we can see, there are several different methods and combinations, and even those can have different versions with small changes that affect both the preprocessing and the query times.

2 Contraction hierarchies

In a road network there are more important intersections, such as large interchanges. And there are less important intersections which lie only on local

shortest paths, for example in small villages. If we could eliminate such unimportant nodes to reduce the search space, we could achieve much shorter query times.

The aim of Contraction Hierarchies (CH) method is to achieve this goal. The algorithm contains a preprocessing phase in which we identify unimportant nodes using heuristics. Later we answer the queries using a slightly modified bidirectional Dijkstra algorithm.

2.1 Preprocessing

Our input is a (directed) graph $G = (V, E)$ and the cost function $c : E_0 \to \mathbb{R}^+$.

During the preprocessing we assign priorities to the nodes of the graph and *contract* the least important node until there are nodes in the graph.

Definition. *Contracting* a node v means the following: for every pair of $u, w \in V$ where $(u, v) \in E$ and $(v, w) \in E$ we examine the shortest $u \to w$ path. If this path is $P = u, v, w$, then we add an arc (u, w) to E with $c(u, v) = c(u, v) + c(v, w)$, the cost of the path. When we have checked all pairs, we remove v from the graph for the remainder of the preprocessing.

Figure 2: Contracting node v: we need to add the edge (u, w_1), but we do not need (u, w_2).

Note. After contracting a node, the cost of the shortest paths between the remaining nodes do not change.

Calculating the priority of the nodes is based on heuristics.

The most important one is the *edge difference* of a node v [4]. It is the number of new arcs we need to insert when contracting v minus the arcs incident to v. If this is small, it means that locally only a few shortest paths go through v. In the example above, the edge difference of v is -2.

It is also important to contract nodes in a uniform fashion in the graph. That is, the priority of a node v is larger if there are more contracted nodes around it. A good heuristic is the number of contracted neighbors. This will hopefully speed up the search between both local and distant pairs of nodes.

There are many additional heuristics we can use to gain small amounts of additional speedup [6]. However, when we contract a node, the priority of the

other nodes change, so we have to recalculate the priorities. So if we use too many or too complex heuristics, then the preprocessing will be slower.

When the preprocessing is finished, we get the order of the nodes they were contracted in, and the original graph with the new arcs. We create two new graphs: $G_\uparrow = (V, E_\uparrow)$ that contains arcs (u, v) where u comes before v in the order, and $G_\downarrow = (V, E_\downarrow)$ that contains the rest.

Notation. Let $P_\uparrow(s, v)$ denote the shortest path between s and v in $G_\uparrow$, and $d_\uparrow(s, t)$ the cost of this path. Similarly, $P_\downarrow(v, t)$ and $d_\downarrow(v, t)$ is the shortest path and its cost between v and t in $G_\downarrow$.

2.2 Finding the shortest paths

To find the shortest path between nodes s and t, we simply run a bidirectional Dijkstra algorithm. The forward search runs from s in $G_\uparrow$ and the backward search runs from t in $G_\downarrow$. We synchronize the searches by alternately finalizing the nodes in them.

The shortest path will be of the form $P_\uparrow(s, v) \cup P_\downarrow(v, t)$, where the node v was finalized in both searches (v can also be s or t). When we find such a node v, we only get an upper estimate $d' = d_\uparrow(s, v) + d_\downarrow(v, t)$ for the cost of the shortest path. During the searches, we keep the value *dmin*, which is the minimum of all d' estimates we have found so far. Since this in only an estimate, we must not stop the searches. We have to continue the forward search until the distance $d_\uparrow(s, v)$ of the next finalized node v is smaller than the value of *dmin*. Similarly, we have to continue the backward search as well. When the searches terminate, the value of *dmin* will be the cost of the shortest path. The following figure shows a small example:

Figure 3: Example CH search: the nodes are in the order of contraction

The forward search will finalize v with distance $d_\uparrow(s, v) = 1$, and the backward search will finalize it with $d_\downarrow(t, v) = 1000$, giving an estimate for the shortest path cost of $d' = 1001$.

The backward search terminates, while the forward search will finalize b and finally t with $d_\uparrow(s, t) = 1000$, which is the cost of the shortest path.

Note. If we have a node v that was finalized in both directions, then the nodes of $P_\uparrow(s, v)$ and the nodes of $P_\downarrow(v, t)$ are in ascending order according to the order of contraction. That is, from the nodes on the path $P_\uparrow(s, v) \cup P_\downarrow(v, t)$, node v was contracted last.

Note. We can reuse the information of the backward search if we want to answer multiple queries whose targets are the same node t. Similarly, we can reuse the forward search if the sources are the same.

2.3 Retrieving the path

The path we find usually contains arcs that we added during the preprocessing algorithm. If we want to find the actual path, for example to draw it on a GPS, then we have to store additional data during the preprocessing phase.

A new arc that we add during the contraction of v represents two arcs incident to v: (u, v) and (v, w). It is enough if we store these two arcs on the new arc. Note, that a new arc can represent other new arcs, not only original arcs.

Additionally, during the search, we have to maintain the node *nodemin*, from which we got the current value of *dmin*.

When the searches terminate, we obtain the paths $P_\uparrow(s, dmin)$ and $P_\downarrow(dmin, t)$ from the searches using the parent pointers from *dmin*. We can now recursively "unpack" the arcs of the path $P_\uparrow(s, dmin) \cup P_\downarrow(dmin, t)$ to obtain the original path.

2.4 Combination with Transit Node Routing

We can gain additional speedup of the queries if we compute a distance table between the last few nodes $T \subseteq V$ that we contract. In our experiments, we chose the last $|T| = \sqrt{|V|}$ nodes.

The searches do not continue from these nodes. Instead, when we finalize a node $v \in T$ in the forward search, we have to examine every node $w \in T$ that was already finalized in the backward search, whether $d_\uparrow(s, v)+d(v, w)+d_\downarrow(w, t)$ is a better estimate than the current *dmin*, and update it if necessary. We do the same with the backward search.

If we want to obtain the actual path, we have two choices. We can explicitly store the shortest paths between the $\binom{|T|}{2}$ nodes, which requires a lot of memory. Or when the searches terminate, we find out which two nodes from T are on the path, and run a CH search between these two nodes, which is fast, since they are both at the end of the contraction order.

3 Highway Hierarchies

The road networks are hierarchical by nature. The farther we are from our starting location toward our destination, the more we want to take a faster road, such as a highway.

The Highway Hierarchis (HH) identifies the *highway edges* of a graph, and creates a hierarchy in the preprocessing phase to make the queries faster.

3.1 Preprocessing

Our input is a graph $G = (V, E)$ and the cost function $c : E_0 \to \mathbb{R}^+$.

During the preprocessing we will create new graphs $G_{i,A} = (V_{i,A}, E_{i,A})$ and $G_{i,B} = (V_{i,B}, E_{i,B})$ for every $i = 1, \ldots, L$, if we want to create L number of additional levels in the hierarchy. The original graph G will be $G_{0,B}$ with our notation.

Definition. For every level $i = 0, \ldots, L-1$, we have a *neighborhood radius* $r_i(v)$ for every node v of the graph $G_{i,B}$. The radius is usually chosen so that for $R_i(v) = \{w : d(v,w) < r_i(v)\}$, $|R_i(v)| = H$ for a parameter H.

Definition. We call an edge (u, v) a *highway edge* in level i if there exists s, t such that the shortest path between s and t contains (u, v) and $v \notin R_i(s)$ and $u \notin R_i(t)$. The following figure contains an example.

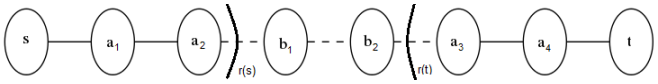

Figure 4: The dashed edges are the highway edges. The edge (a_2, b_1) for example is a highway edge, since $b_1 \notin R_0(s)$.

We have two additional methods:

- *edgeReduction*: this method creates $G_{i,A}$ from $G_{i-1,B}$ by removing edges that we do not consider highway edges.
- *nodeReduction*: this method creates $G_{i,B}$ from $G_{i,A}$ by contracting certain nodes of the graph. The contraction algorithm is the same as the one used in Contraction Hierarchies. A node v is contracted if $\#newEdges \leq c \cdot \#deletedEdges$ for a parameter c. This method will always remove the isolated nodes.

Finally the algorithm to preprocess a graph:

Algorithm 1 preprocessHH(G)

$G_{0,B} = G$
for $i = 1 \ldots L$:
    calculate $r_i(v)$ for every $v \in V_{i-1,B}$
    $G_{i,A} = edgeReduction(G_{i-1,B}, r_i)$
    $G_{i,B} = nodeReduction(G_{i,A})$

3.2 Finding the shortest paths

After the preprocessing is done, we have access to the graphs $G_{i,A}$, $G_{i,B}$ and the radii functions r_i for $i = 0, ..., L$. For consistency, $r_L(v) = \infty$ for every node v, and $r_i(v) = \infty$ if $v \in V_{i,A} \setminus V_{i,B}$.

To answer an s, t query, we use a modified bidirectional Dijkstra algorithm similarly to CH.

The idea is that we limit the search using the radii functions, that is, when the search leaves the neighborhood of a node, it continues on the next level.

In addition, when searching on level i, we can bypass the nodes that were contracted during the *nodeReduction* step, that is the nodes in $V_{i,A} \setminus V_{i,B}$.

The rules of the forward search are the following (the rules for the backward search are the same):

- We begin searching from s in $G_{0,B}$, that is the original graph.
- The Dijkstra search is not modified if a node $v \in R_0(s)$.
- When relaxing an edge (u, v) and $v \notin R_0(s)$ we only insert v into the heap if (u, v) is a level 1 edge. We call such a node v an access node to level $1A$ or $1B$.
    1. If $v \in V_{1,B}$ then we continue the search from v in $G_{1,B}$ using the same rules.
    2. If $v \in V_{1,A}$ and $v \notin V_{1,B}$, we continue the search from v in $G_{1,A}$. In this case, $r_1(v) = \infty$, so we have to search the whole component that was contracted until we find the access nodes to $G_{1,B}$.

Note. In *nodeReduction*, we only contract less important nodes, similarly to the *edge difference* heuristic used in CH. This way, a typical search is less likely to enter a contracted component, so the query will be faster.

Note. The algorithm works on directed graphs as well. We only need to modify the definition of $r_i(v)$. In the new definition, $d(u, v)$ means the distance between u and v using the directed arcs of the graph as undirected edges.

3.2.1 An example

The rules are very complicated. Hopefully the following figure will make it easier to understand.

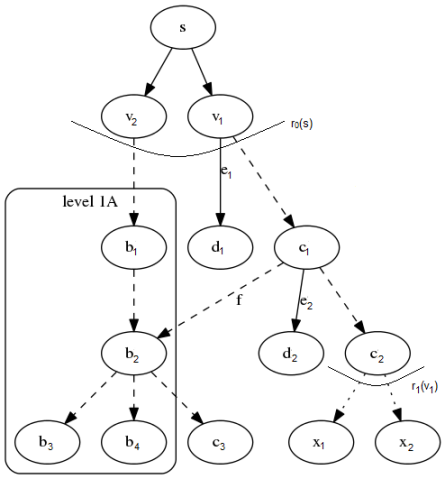

Figure 5: An example of the HH search algorithms.

The solid lines are the level 0 arcs, the dashed lines are the level 1 arcs, and the dotted lines are the level 2 arcs.

We begin the search from s. The nodes v_1 and v_2 are in $R_0(s)$, so they will be finalized. When finalizing v_1, we do not relax the arc e_1, since $d_1 \notin R_0(s)$.

The node b_1 is a level $1A$ access node. It is located in a contracted component whose nodes are not in $G_{1,B}$. So from b_1, we have to continue the search until every node in the component is finalized (the nodes b_2, b_3, b_4 in the figure) and we find c_3 who will become the access node to level $1B$.

The node c_1 will be a level 1 access node, so we will only search in graph $G_{1,B}$ from there. Therefore, the edge e_2 will not be relaxed. We also do not relax the edge f since it goes to a contracted component, so f is not part of $G_{1,B}$.

Finally, the level 2 access node will be x_1 and x_2.

3.2.2 Stopping the search

Similarly to CH, the shortest path will be of the form $P_\uparrow(s,v) \cup P_\downarrow(v,t)$ where node v was finalized in both directions. The stopping criterion is the same as well: when we finalize a node v in both directions, we get an upper estimate for the cost of the shortest path. Let *dmin* be the best estimate we found. We can stop a search if the next finalized node has a greater distance label than *dmin*.

4 Highway Node Routing

In a road network, most shortest paths run through a relatively small number important intersections. We will denote these nodes with S_0 and the graph

representing the road network by G_0. If we precompute the shortest paths information between the nodes of S_0, we can speed up the queries, since the information can be used to bypass some nodes during the search.

We store the precomputed information in an overlay graph G_1. This graph will contain the nodes from S_0, while the edges of the graph will have the following property: the cost of the shortest path between $s, t \in S_0$ in G_1 must be the same as the shortest path between the same nodes in G_0.

The same approach can be used again: we identify important nodes S_1 in G_1, and construct G_2, and so on.

Definition. By the *level of a node* v we mean the following: $level(v) = \max\{i : v \in G_i\}$.

Definition. The *level of an edge* $e = (u, v)$ belonging to the graph G_i is i, that is $level(u, v) = i$.

Finding a shortest paths between nodes s, t can be achieved by using a modified bidirectional Dijkstra algorithm. During the search, we do not go down in the hierarchy, that is, we only relax an edge (u, v) if $level(u, v) = level(u)$.

Note. The important nodes are obtained from the Highway Hierarchies preprocessing method. Node $v \in S_i$ if v was not removed during the *nodeReduction* phase of the construction of level i.

4.1 Preprocessing

We are given the graph $G_0 = (V_0, E_0)$, the cost function $c : E_0 \to \mathbb{R}^+$ and the level one nodes S. From this input we have to create the graph G_1. We will provide more details about constructing G_1. The other levels can be created in a similar fashion.

During the preprocessing, we have to find the shortest path between $s, t \in S$ in G_0 and add an (s, t) edge to G_1. We do not need to calculate the shortest path between every pair of node in S. So our aim is to minimize the number of additional edges.

When we run a Dijkstra algorithm from $s \in S$ in G_0, we get a shortest paths tree B. We have to find a set of nodes $C \subseteq S$ that "cover" the branches of B.

Definition. We are given S, s and B as before. The set $C \subseteq S$ is a *covering set* if $\forall t \in S \setminus C$ the path P from s to t in B contains an internal node from C.

Using this terminology, to create the overlay graph G_1, we have to find a covering set C for $s \in S$ and add the edges (s, v) to G_1 for every $v \in C$. We have to do this for every $s \in S$.

Note. The Contraction Hierarchies method is a special case of Highway Node Routing: in CH, $S_{i+1} = S_i \setminus \{v\}$, where v is the ith node that we contract $(i = 0, 1, \ldots, n-1)$.

4.2 Finding covering sets

We are given the set of level 1 nodes and a node s. We have a shortest paths tree I rooted at node s obtained from a Dijkstra search in G_0. We get the tree I' by deleting the descendants of every level 1 node from I. Our aim is to determine the correct distance labels of the nodes in I' without running a full Dijkstra search from s.

Note. The tree I' will contain only nodes v for which the shortest $P(s, v)$ path does not contain level 1 internal nodes. Only the leafs of the tree can be level 1 nodes, but not necessarily all of them.

Since we do not know when to stop the search exactly, we need some stopping criterion. We examined four approaches. All of them involve running a modified Dijkstra's algorithm from s [1].

Note. As we will see, some of these approaches will incorrectly label some of the nodes. However, this will not be a problem, since these nodes are not in the tree I'.

To describe the techniques, we will use the following notations:

Notation. We will refer to the current shortest paths tree built by the modified Dijkstra algorithm by B.

Notation. We will denote the path from s to v that the modified Dijkstra found by $P(s, v)$.

Notation. The distance from s to v in the tree B will be denoted by $d(s, v)$. This is also the distance label of v in the Dijkstra algorithm, and the cost of $P(s, v)$.

An example can be found in 4.3 with explanations regarding all the techniques that we will now define.

4.2.1 The conservative approach

We stop the regular Dijkstra search when every node in the heap has at least one ancestor in level 1.

If there is a long path that is not covered by a level 1 node for a while, this method will be very inefficient as we can see in the example later.

In the implementation, we store on every node v whether the $s \to v$ path in B is covered. And we store a counter which stores the number of uncovered nodes that are in the heap. The counter can be easily updated. The algorithm stops, when it reaches zero.

4.2.2 The aggressive approach

We modify the Dijkstra search by not relaxing the edges leaving a finalized level 1 node.

The problem with this approach is that the search may continue around these nodes on level 0 and flag many more level 1 nodes as covering nodes. The

distance labels of such nodes can be incorrect. However, since these nodes are not in the tree I', this is not a problem.

4.2.3 Stall-in-advance

We continue but limit the search from a level 1 node to compensate for the deficiency of the previous method.

The article suggests that we do not continue the search from v, if v is covered by c number of level one nodes. If c is one, this corresponds to the aggressive variant.

The implementation is similar to the conservative approach. We store a counter so we know the number of uncovered nodes in the heap. And on a node v, instead of a boolean value, we store how many level 1 nodes are on the $s \to v$ path in B.

Note. All of the three previous algorithms work on directed graphs without modification.

4.2.4 Stall-on-demand

In this technique, every node can have two additional states: *stopped* or *stalled*.

We do not relax the edges leaving a level 1 node when finalizing it. Such a node will be *stopped*. As we have seen earlier, because of this, the search continues around stopped nodes, and some can have wrong distance labels. During the search, if we find a node w whose distance label is incorrect, then w becomes *stalled*. We also do not relax the edges leaving a *stalled* node when finalizing it.

Finding incorrectly labeled nodes: When we finalize a node u, and relax the edge (u, v), we check if v is *stalled* or *stopped*. If that is the case, we begin a search from u, for example a BFS. We are only interested in nodes w that are in B, the partial shortest paths tree that we are currently building, so we limit the search to the nodes of this tree. If we find that $d(s, v) + c(v, u) + d'(u, w) < d(s, w)$, then w becomes *stalled*. The values of $d(s, v)$ and $d(s, w)$ are known from the distance labels. The value of $d'(u, w)$ can be calculated by traversing the path we found during the BFS from u to w. When we stall a node, we also update its distance label as we have just realized that the previous one was not correct. That is, we set $d(s, w) = d(s, v) + c(v, u) + d'(u, w)$. This way, it is possible that later on more nodes will be stalled because of w.

Note. Instead of the BFS, we could use another Dijkstra algorithm. However, in [1] they claim that a BFS produces similarly good results and has a much smaller overhead.

Note. During the BFS, we have to use all edges of the graph, not only the edges of B since these additional edges can also belong to a shorter $P(s, w)$ path. We can see an example of this in 4.3.

Note. If we stall a reached but not finalized node w, it is possible that later we find a shorter path from s to w than the current label of w that we updated when we stalled w. In this case, we have to remove the *stalled* state from w.

Note. Continuing the search from an unstalled node will not stall additional nodes. Therefore it is enough to add nodes the are stalled to the queue of the BFS.

Note. If a node v has a *stalled* ancestor, then v will be *stalled* as well.

Note. We do not add a level one node v to the covering set at the end of the algorithm if it is *stalled*, since we know that the shortest s, v path contains another level one node.

The stall-on-demand technique was only described for undirected graphs in the articles. In the directed case, there are multiple problems that need to be addressed. We need one additional notation before we continue.

Notation. $\overleftarrow{P}(u, v)$ will denote a path from u to v that uses arcs of the reversed graph.

4.2.5 Directed stall-on-demand

In the undirected case, we checked if v is *stalled* or *stopped* when relaxing an edge (u, v). If we found that the path $P(s, v) \cup \{(v, u)\}$ is shorter than the current path $P(s, u)$ then we stalled u.

In the directed case, regardless if the arc (u, v) does not exists, the "backward arc" (v, u) can still be used to stall u. So we have to consider paths of the form $P(s, w) \cup \overleftarrow{P}(w, v)$ when we decide whether we stall w.

We store two labels on every node v: $d_f(s, v)$ is the regular distance label, while $d_b(s, v)$ will be the cost of a path that has the form $P(s, w) \cup \overleftarrow{P}(w, v)$ for some node w.

When we finalize a node u because $d_f(s, u)$ was minimal, we relax the out-arcs (u, v) by updating $d_f(s, v) = d_f(s, u) + c(u, v)$, and the in-arcs (v, u) by updating $d_b(s, v) = d_f(s, u) + c(v, u)$ if necessary. We will not finalize u again.

When we finalize a node u because $d_b(s, u)$ was minimal, we only relax the in-arcs (v, u) and update $d_b(s, v) = d_b(s, u) + c(v, u)$ if necessary. In this case, u may be finalized again if its $d_f(s, u)$ label becomes minimal.

When relaxing the in-arc (v, u) in either case, if v is *stopped* or *stalled*, we begin a search from u to stall some nodes.

The modified search: Since there are paths whose arcs are not directed the same way, a simple search is not enough. It would not stall every node that has a *stalled* ancestor.

We limit our search to the nodes of our treelike object as can be seen in figure 7.

There are two phases. In both phases we use a modified BFS. We have two queues of nodes. Initially the first queue contains u, while the second one is empty.

In the first phase we use the first queue. We add the out-neighbors of the currently searched node to the first queue, and the in-neighbors to the second queue. During this phase, we wish to stall nodes that have a regular $P(s, w)$ path that do not contain reversed arcs.

In the second phase, starting when the first queue becomes empty, we use the second queue and add the in-neighbors to the second queue. We do not finalize nodes that were finalized in the first phase as they are stalled already. However, the first phase will not stall nodes whose path contains a reversed arc.

In the first phase, if we find a node w for which $d_f(s, v) + c(v, u) + d'(u, w) < d_f(s, w)$, then w becomes *stalled*. The values of $d_f(s, v)$ and $d_f(s, w)$ are known from the labels. The value of $d'(u, w)$ must be calculated during the search.

In the second phase, if we find a node w for which $d_f(s, v) + c(v, u) + d'(u, w) < d_b(s, w)$ and $d_f(s, w) = \infty$, then w becomes *stalled*. This is necessary, since we only want to stall nodes in phase two that have stalled ancestors (that we stalled in phase one). If $d_f(s, w) \neq \infty$ and phase one did not finalize w, then one of the arcs on the $P(s, v) \cup \{(v, u)\} \cup P(u, w)$ path faces in the wrong direction. See the second example in 4.3.

Note. The notes regarding the undirected case apply here as well.

Note. Suppose we stall node w because $d_b(s, w)$ was incorrect (in the second phase of the search). This means that $d_f(s, w) = \infty$. When w gets a new $d_f(s, w)$ label, the stalled status has to be removed.

Proposition. *The algorithms correctly determine the distance labels of every node in I'.*

Proof. By limiting a Dijkstra search, we know that the distance label of a node v can only be equal to or larger than the distance label of v in the unmodified Dijkstra search.

We prove by induction by the order in which the nodes were finalized in I' that the distance labels $d(s, u)$ (and $d_f(s, u)$ in the case of the directed stall-on-demand) are equal for the nodes in I'.

The node s will be the first finalized node, with distance label 0.

Suppose the first k finalized nodes in I' have correct labels. Let the next finalized node in I' be v. We know that the shortest $P(s, v)$ path does not contain a level 1 internal node. This means that none of the approaches will alter the behavior of a regular Dijkstra on this node. More specifically, in the stall-on-demand approach, we do not stall a node w for which the shortest $P(s, w)$ path does not contain a level 1 internal node. Since the labels of the previous nodes were correct, the label of v will be correct as well. □

4.3 Examples

We will describe how the different techniques operate on the following graph.

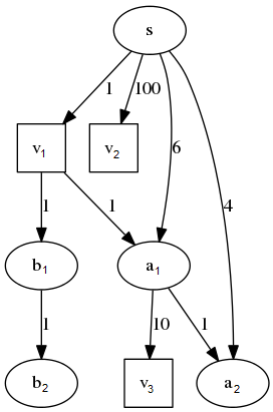

Figure 6: The graph

The square node v_1, v_2, v_3 are the level one nodes. We begin the Dijkstra search from s. The tree I' will consist of the three nodes s, v_1 and v_2.

It is easy to see that the *conservative approach* will finalize every node, since v_2 will be finalized last.

The *aggressive approach* will not relax the arc (v_1, a_1), since v_1 is a level one node. Therefore a_1, a_2 and v_3 will be finalized with wrong distance labels, and v_3 will be falsely marked as a covering node.

In the case of the *stall-in-advance* technique, if we continue the search from v_1 for a while (for example we finalize its children a_1 and b_1), then v_3 will not be finalized and marked a covering node.

When using the *stall-on-demand* technique, the arc (v_1, a_1) will not be relaxed, since v_1 is a *stopped* node. When we finalize a_1 with distance label $d_f(s, a_1) = 6$, v_3 will be inserted into the heap. When relaxing the in-arc (a_1, v_1), we notice that v_1 is stopped, so we begin our modified BFS from a_1. Since the path s, v_1, a_1 is shorter then s, a_1, we stall a_1. The arc (a_1, a_2) is not part of the shortest paths tree, but it can still be used to stall a_2, since the path s, v_1, a_1, a_2 is shorter than the path s, a_2. Finally the BFS finds v_3 and stalls it so it will not be a covering node.

The next figure shows the stall-on-demand technique in more detail on a different example.

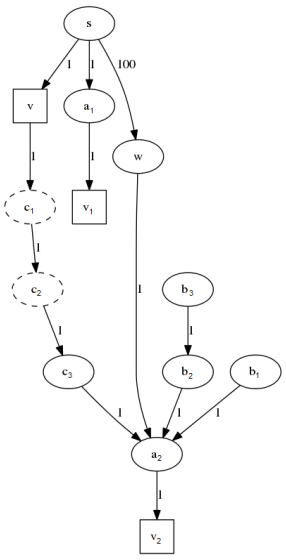

Figure 7: The treelike object

The square nodes v, v_1, v_2 are the level one nodes. Only the dashed nodes are not finalized. The heap currently contains the node c_2 with the label $d_b(s, c_2) = 103$ (and $d_f(s, c_2) = \infty$)

Later, when we finalize c_1, we will begin the modified BFS from c_1. The first phase will stall the following nodes: c_1, c_2, c_3, a_2, v_2. However, the first phase will not reach the nodes b_1, b_2 and b_3. Nevertheless, b_1 and b_2 will be inserted into the second queue, so the three nodes will be stalled in the second phase.

The second phase also reaches node w. However, since $d_f(s, w) = 100 \neq \infty$, we do not stall w.

In conclusion, we needed to relax the in-arcs during the Dijkstra search to find the shorter v, c_1, c_2, c_3, a_2 path, and we needed the second phase of the BFS to stall the b_1, b_2, b_3 nodes.

4.4 Finding the shortest paths

When the preprocessing is finished, we get the level of the nodes and arcs.

To answer a query, we use a modified biderictional Dijkstra algorithm where we only relax an arc (u, v) if $level(u, v) = level(u)$, that is, we do not go down in the hierarchy. This approach corresponds to the *aggressive* variant of the preprocessing method, so it has the same problems.

Fortunately we can apply the *stall on demand* technique as well. The only modification we have to make is due to the fact that the definition of *stopped* nodes is no longer valid, since we continue the search from every node. So instead, when finalizing node u and relaxing the backward arc (v, u) we begin

the stalling process if v is finalized and $level(v) > level(u)$ (or if v is already stalled).

The stopping criterion is the same as in the case of CH. Nodes that we relay in both searches give an upper estimate for the cost of the shortest path. We keep track of the best estimate *dmin*, and continue the forward (and the backward) search, until the distance label of the next finalized node is greater than *dmin*.

5 Dynamic scenario

Road conditions constantly change. There are traffic jams resulting from accidents and high road congestion in rush hours or during road constructions. These conditions translate to the increase of the arc costs in our graphs. When these conditions come to an end, the arc costs revert to their original values.

The methods we previously examined all rely on precomputed data. When the cost of an arc changes, it invalidates a large portion of this data which we have to correct to ensure that the results of the queries are correct.

5.1 Contraction Hierarchies

In CH, we add a "shortcut edge" (u, w) to the graph instead of the arcs (u, v) and (v, w) when contracting a node v if $P = u, v, w$ is the shortest $u \to w$ path. Thus an added edge e always represents a path P_e that contains only original edges.

When we increase the cost of an original edge f, there are two problems that arise:

1. We must also change the length of all edges a whose path P_a contains f. Finding edges e such as a is possible by storing the edge e on the arcs of P_e. This, however, requires a lot of memory and it does not solve the second problem.

2. It is possible that there is an edge b that we did not add during preprocessing because another path trough f was shorter. After the change however, we should add b. But by adding b, we might have to insert additional "shortcut edges" as well. So a small change can ripple through the entire hierarchy.

5.1.1 Disabling arcs

We examined what happens when we do not account for the second case.

"Unpacking" an edge $e = (u, w)$ to obtain P_e is done recursively. During this recursive procedure we can mark every arc f we encounter if P_f contains a changed arc as the following algorithm shows:

Algorithm 2 unpackMark(e)

```
input: the arc e
if (original(e)):
  if (changed(e)):
    mark(e)
    return true
  else: return false
else:
  a = unpack(e.first)
  b = unpack(e.second)
  if (a || b): mark(e)
  return a || b
```

Note. As explained before, if the arc e is not an original arc, it represent the arc $e.first = (u, v)$ and $e.second = (v, w)$ arcs.

We modify the search the following way:

1. We run a CH search using the original data structure and obtain a path P.
2. We unpack the arcs of P using the unpackMark(e) method to obtain its original arcs and find the actual cost of this path taking into account the changed costs.
   (a) If the unpackMark(e) returns with true for at least one arc we run the search again, but we exclude the marked arcs, that is, they will not be relaxed.
   (b) If we do not find a changed arc on the path, or we do not find a path at all, we stop and return the path with the minimal cost from the previous searches.

Note. Naturally, when we run a new s, t query, we unmark every marked arc.

This method will not always find the shortest path since it is possible that we disable an arc that is actually part of it. In addition, since we do not consider the second problem, that is, we do not run the construction algorithm so we do not account for arcs that need to be inserted, we may not be able to find the shortest path at all.

5.1.2 Preprocessing in the same order

To ensure that the queries return the exact shortest paths after changing the arc costs, the easiest solution is to run the preprocessing again.

We can use the same order of the nodes so we do not have to calculate and update the priorities. Hopefully, if the changes are small, using the same order will not result in much slower queries, but the preprocessing will be much faster.

5.1.3 Combining with A*

We may use the A* algorithm for an s, t query. As explained in section 1.2, we need an estimate $h(v, t)$ for every node v in the heap. Since a CH query is fast, we can use the unmodified data structure to provide lower estimates for the $v \to t$ shortest path costs [17]. This estimate will be the value of $h(v, t)$.

This estimate should be very accurate, so hopefully we only need to finalize a few nodes. Specifically, when the shortest path does not contain changed arcs, this method will only finalize the nodes on the shortest path. In addition, since during an s, t query we only need estimates for $h(v, t)$, it is enough to run the backward search only once.

Note. We can only use this method if the arc costs increase.

5.2 Highway Node Routing

HNR is very similar to CH. The added edges represent paths here as well, and this presents the same problems.

However, one advantage we have over CH is the limited number of levels and the method we use for construction. When we change the cost of a level 0 arc, that is an original arc, there is a relatively well defined and hopefully small set of nodes on level 1 that are affected by the change. If we run the level 1 construction again only on these nodes, we can "repair" our data structure so the queries will be answered correctly. And if we find that a level 1 arc changed its cost, that change will only affect a small portion of level 2 nodes, and so on.

5.2.1 The formal algorithm

The following algorithm is a formal description of the above idea:

Algorithm 3 HNR.update($E^{'}$)

input: $E^{'}$, the set changed arcs
$V_0^{'} = \{u : (u, v) \in E\}$
for $l = 1, \ldots, L$:
  $V_l^{'} = \emptyset$
  $R_l = \cup_{u \in V_{l-1}} A_l^u$
  for $v \in R_l$:
    repeat the construction step of v in level l
    if something changes, put v into $V_l^{'}$

The variables are the following: L is the number of levels of the hierarchy, A_l^u is the set of nodes whose level l construction depends on the node u, and $V_l^{'}$ is the changed node set on level l (since V_l denotes the nodes of the overlay graph G_l).

5.2.2 Managing the sets

We must somehow store the sets A_l^u. Fortunately, during the preprocessing phase, we can do this.

Recall, that during preprocessing, we run a construction algorithm on every level $l = 1, \ldots, L$ for every node $v \in V_l$. Suppose we are constructing a node v on level l. For this we run a modified Dijkstra algorithm from v as explained in section 4.2.

The faster but more memory-consuming approach is to store node v on every finalized node during the algorithm. When we complete the preprocessing of level l, every node u will contain a set of nodes, which is exactly A_l^u.

A slower, less precise but more memory-efficient approach is to store the maximum distance on the finalized nodes. That is, we store a number d_l^u on every node u, initially $d_l^u = 0$, for every u and $l = 1, \ldots, L$. When we run the construction on node v in level l, for every finalized node u we update d_l^u: $d_l^u = max(d_l^u, d(v, u))$. This way, when we want to obtain the set A_l^u, we simply run a Dijkstra algorithm from u in the reverse graph of G_l until the distance of the next finalized node is bigger than d_l^u. Then A_l^u will be set of finalized nodes. Since this is a superset of the set we would obtain using the previous method, the update algorithm will be slower.

6 Tests

We implemented the algorithms for Contraction Hierarchies, Highway Hierarchies and Highway Node Routing using the LEMON C++ library. We tested the implementations on an Intel Core i7 4770 processor clocked at 3400 MHz, and 16 gigabytes of RAM.

6.1 Implementation difficulties

The details about these algorithms in the articles are vague at best, except for HH, which is described well in [8].

For example, as we already mentioned, the stall-on-demand technique originally only works on undirected graphs, and some details and explanations were missing from the article.

Because of the lack of information, our implementations cannot hope to match the efficiency reported by the authors. However, we found that we get similar performance variations between our implementations as we can find in [9].

6.2 The graphs

We tested the implementations on some of the graphs provided for the 9th DIMACS Implementation Challenge in 2006 [22], and a graph of the Hungarian road network available on the LEMON home page [20].

The graphs, all of them directed, are the following:

Location	Abbreviation	Nodes	Arcs
Rome	RO	3353	8870
New York	NY	264346	733846
Hungary	HU	292366	777988
Colorado	COL	435666	1057066
Florida	FLA	1070376	2712798
California	CAL	1890815	4657742
West USA	WUSA	6262104	15248146

Table 1: The graphs

We have previously found that the methods work differently on distance graphs and travel time graphs. In the results, the -d after a graph name means that it is the distance graph, and the -t means it is the travel time graph.

Note. Not all graphs have both distance and travel time information.

6.3 LEMON

The open source project was started in 2003 by Egerváry Research Group on Combinatorial Optimization (EGRES) at the Department of Operations Research, Eötvös Loránd University.

The aim of the LEMON library is to efficiently implement the algorithms and data structures in connection with graphs and networks [20].

6.3.1 Features

There are multiple graph implementations in the library which have different goals. The *ListDigraph* contains methods with which we can add and delete nodes and edges, while the *StaticDigraph*, as its name suggests, is static, that is we cannot add additional edges to it, however, the implementation is faster.

We can store any type of data we want on the nodes and the arcs of a graph using NodeMaps and ArcMaps. These map implementations are faster than the standard template library map implementation.

There are other auxiliary data structures that we use, for example a binary heap data structure. There is also support for reading graphs in DIMACS format.

These algorithms and data structures are well documented at [21].

6.3.2 Disadvantages

The authors of [10] claim that using an existing library for implementing such algorithms, where the performance is key, is not recommended. With a specialized implementation, we could gain additional speedups.

For example, Dijkstra's algorithm, while fast in LEMON, does not consider a scenario where we have to run the algorithm multiple times and only in a small part of the graph. When resetting the algorithm, every data structure, that is the node maps containing the distances, predecessors and the heap are deleted and new ones are allocated. Instead of this, we reset the values of the nodes that were affected by the previous search, which is much faster, since in a local search only a handful of nodes are affected.

Another example are the graph implementations. For example, if we could iterate through the outarcs of a node in a predefined order, we could gain additional speedups. However, this is not possible using LEMON.

Unfortunately we did not have time for a custom tailored implementation of the necessary data structures.

6.4 Static tests

We tested the different methods by focusing on the query speed, the number of finalized nodes and the time needed for the preprocessing.

The times are given in seconds. The speedup and finalized nodes are always compared to a simple Dijkstra search. A value of x means that Dijkstra's algorithm finalized x times more nodes than the method under test. Similarly, if the speedup is x, it means that the tested method was x times faster.

We always ran 100 tests between random source and target node pairs but we use the same 100 pairs for the different tests on the same graphs.

We measured what happens if we increase the number of tests from 100 to 1000 and 10000, and we found very little difference in the results regardless of the method we examined and the graph we run the tests on.

6.4.1 Contraction Hierarchies

We tested multiple implementations of CH in the BSc thesis [19]. The following table shows the results of the default implementation.

Graph	Preprocess time	Finalized nodes	Speedup
RO-d	0.73	15.75	9.66
NY-d	149.636	205.21	134.57
HU-d	200.466	303.2	219.02
COL-d	92.001	444.64	233.92
FLA-d	182.991	473.54	507.13
CAL-d	540.016	884.53	442.87
WUSA-d	1920.09	1935.29	769.15
NY-t	72.133	345.52	258.2
HU-t	94.220	496.27	404.18
COL-t	34.326	960.26	501.25
FLA-t	75.025	2072.94	1131.8
CAL-t	205.420	1966.31	1018.86
WUSA-t	680.905	5233.08	1841.77

Table 2: CH results

As we can see, the CH is multiple times faster than a simple Dijkstra algorithm, but performs less efficiently on distance graphs. The additional speedup compared to the BSc thesis can be contributed to a small refactoring of the code and the better hardware we used for testing. Since the CH method has more data structures associated with it than a simple Dijkstra algorithm, the faster memory and larger CPU cache will have greater effect on its performance.

We also tested the CH and Transit Node Routing combination. Finding the actual path instead of the total cost was not implemented in the BSc thesis. Now we do this by using the second method explained in section 2.4, namely, to obtain the path after the search finished, we run a CH search between the two transit nodes to find the path between them instead of storing it explicitly.

Graph	Preprocess time	Speedup	Speedup with path output
NY-t	74.463	392.21	297.67
HU-t	97.502	574.94	363.35
CAL-t	215.659	2281.05	1049.7

Table 3: CH + Transit Node Routing results

The results show that the CH and Transit Node Routing combination is 10-20% faster than the simple CH with only a small increase in preprocessing time. The retrieval of the actual paths makes the queries slower.

6.4.2 Highway Hierarchies

We implemented the HH preprocessing and search algorithms based on [8]. Compared to CH and HNR, the algorithmic details are relatively well written.

Parameters: As a reminder, H is the neighborhood size to determine highway edges, and c is a constant that determines when to contract a node.

We chose the following parameters for the preprocessing: $H = 50$, $c = 0.5$, and we increased H by 20 per every level.

These parameters turned out to be a safe choice for most of the graphs. Precisely finding the best ones would require weeks of trial and error as the preprocessing time is largely affected by these parameters.

However, we have found the following correlations:

- Increasing H reduces the number of nodes on each level, makes the queries a little faster, and significantly increases the preprocessing time.
- The values for c below 0.5 result in hours of preprocessing time and less nodes per level.
- The values for c above 1 usually removes all nodes from the graph in the first few levels.

Note. We run the preprocessing until no more nodes remain, that is all of them are contracted by the *nodeReduction* method.

Speedup test: The following table contains the results:

Graph	Preprocess time	Finalized nodes	Speedup
NY-d	75.547	5.33	2.16
HU-d	178.614	6.71	2.88
CAL-d	289.375	20.65	6.48
NY-t	32.345	7.64	3.81
HU-t	94.428	11.08	4.66
CAL-t	130.123	48.43	17.16
WUSA-t	447.533	111.13	36.71

Table 4: HH results

As we can see, similarly to CH, HH works better on the travel time graphs. Compared to CH, we see similar differences in performance as we can see in [9].

Testing termination condition: We examined what happens during an s, t query if we immediately stop the search when we find a node v that was finalized in both directions and take a path of the form $P_{\uparrow}(s, u) \cup P_{\downarrow}(w, t)$ where u and w were finalized in the forward and backward search respectively. Since we do not always find the shortest path this way, we also measured the average error of the found path, that is, how many times longer it is than the shortest path. We found that only 10% of the found arcs are incorrect, and even in those cases the error is small.

Graph	Preprocess time	Finalized nodes	Speedup	Error	Original speedup
NY-t	32.501	17.69	8.39	1.00090	3.81
HU-t	92.320	25.38	11.33	1.00033	4.66

Table 5: HH results: early stop

The results are similar on the other graphs as well. As we can see, we gained additional speedups and the errors are not substantial, which suggests that other termination conditions might be more efficient.

Finalized nodes per level: Since we only have a limited number of levels, we also examined how many nodes the searches finalize on each level. We ran the test on the HU-t graph. The following chart shows the number of nodes per level, that is, how many nodes remain on the level after the *nodeReduction* step.

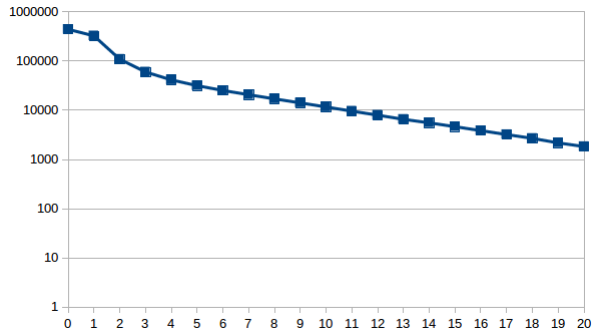

Figure 8: Nodes per level

As we can see, most of the nodes are eliminated in the beginning and only a small amount of nodes remain on the upper levels.

The following chart shows the number of finalized nodes per level:

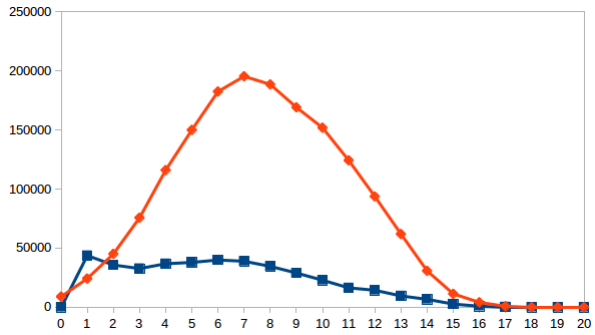

Figure 9: HH finalized nodes per level

The blue line with square points shows the number of finalized nodes in the contracted components, while the red line with the diamond symbols shows the number of finalized nodes that are not in a contracted component on that level.

As we can see, the most nodes were finalized in the middle levels relative to the number of nodes in the corresponding level.

6.4.3 Highway Node Routing

Covering set test: In addition to the speedup test, we examined the different methods for finding the covering sets. Here we ran the HH algorithm, and selected 1000 random level 0 nodes from which we ran the covering set finding algorithm treating the non level 0 nodes as level 1 nodes.

We ran the tests on the NY-t graph. We measured the time, the number of finalized nodes and the number of covering nodes. There are 70396 nodes on level 0 and 193950 on level 1. Since the times are too short, we give them in milliseconds. The results are the following:

Method	Time (ms)	Finalized nodes	Covering nodes
Conservative	7.07	23534	1961
Aggressive	4.80	13709	2585
Stall-in-advance	5.824	16407	2079
Stall-on-demand	522.629	13379	2492

Table 6: Finding covering nodes

As a reminder, the less covering nodes we find the better, since during preprocessing we would have to add a new arc for every found covering node.

As we can see, the conservative approach finds the least amount of covering nodes but it is slower than the aggressive and Stall-in-advance methods. The stall-in-advance technique has the best overall performance in that it is faster than the conservative approach and provides a smaller covering set than the aggressive method, so we use this for the preprocessing.

The surprising result comes from the Stall-on-demand approach. It finalizes the least amount of nodes, but has poor performance. It only stalled a total of 315 nodes. This means that the stalling process in Stall-on-demand is relatively expensive and we should not expect wonders from it. However, this method was originally developed to increase the query speeds and not for preprocessing. As we will later see, it performs better during the search.

Speedup test: Here we compared the aggressive and stall on demand techniques, as the conservative approach would be the same as running a simple bidirectional Dijkstra algorithm, and the Stall-in-advance technique cannot even be defined appropriately for the search. The following tables contains the results:

		Stall-on-demand		Aggressive	
Graph	Preprocess time	Finalized nodes	Speedup	Finalized nodes	Speedup
NY-d	83.121	137.11	0.48	49.10	0.79
HU-d	201.334	439.23	0.495	154.89	0.858
COL-d	64.791	104.03	1.48	47.35	2.27
FLA-d	116.163	162.166	5.07	96.32	6.0
CAL-d	309.307	917.87	1.41	228.6	2.22
NY-t	35.078	62.52	3.43	20.22	2.68
HU-t	108.808	223.099	0.66	66.64	2.26
COL-t	27.118	188.61	17.34	43.22	10.59
FLA-t	60.209	344.42	74.59	105.41	25.19
CAL-t	141.037	457.43	25.08	71.64	10.59
WUSA-t	480.722	1309.82	63.64	147.40	19.13

Table 7: HNR results

The results are very surprising. The preprocessing contains the preprocessing of HH to obtain the node levels as well. Compared to this, HNR only takes a few seconds.

Similarly to CH and HH, we obtained better results in the travel time graphs. The method does not really work well with distance graphs.

On larger graphs, the stall on demand technique works better than the aggressive variant of the search on the travel time graphs. And it seems that the aggressive method performs better on the distance graphs.

The most surprising result is the number of finalized nodes compared to the speedup. For example in CAL-d, Dijkstra's algorithm finalizes 900 times as many nodes as HNR, but we only got a very small speedup. This suggests that the stall-on-demand technique has a large overhead.

We performed an additional test where the arcs were ordered according to their level in descending order. This way, the searches were faster by two times then without it. Unfortunately, in the dynamic scenario, the ordering of the arcs is lost.

The HNR described in [1] was only tested on undirected graphs. In that case, the stall on demand technique is simpler and faster, since we do not have to perform additional backward steps during the search as explained in section 4.2.5. We tested this method with our modifications on directed ones and we can see that it works on larger graphs with the travel time metric.

Finalized nodes per level: We examined the number of finalized nodes per level the same way as in the HH test on the same HU-t graph.

The graph had 25 levels and there were 107 nodes on the last level. The following figure shows the number of finalized nodes per level:

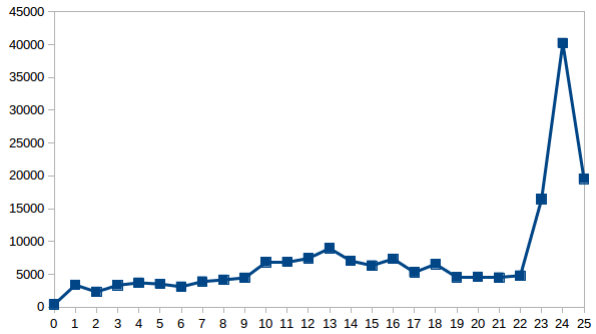

Figure 10: HNR finalized nodes per level

Tests on other graphs show similar patterns. Therefore a good improvement could be made by either breaking up the last 2-3 levels into more smaller ones, or creating a distance table on their nodes, since they only contain a few hundred nodes. However, this would defeat the purpose of why we examined HNR more closely, namely we want to update the data structure quickly when the arc costs change, and if we have a distance table, we cannot do that.

In HH, we limit the search on a level to the neighborhood of the access nodes. In the beginning of the search, we will have an increasing number of access nodes on the successive levels, but after a while, this number will start to decrease as there are fewer nodes on the upper levels. In contrast, the HNR

search quickly reaches the upper levels since we jump to the highest possible level as soon as we can.

6.5 Dynamic scenario tests

We tested all the methods we described in section 5 using different scenarios.

6.5.1 Testing methods

We examined two methods: randomly changing the cost of a few arcs, and simulating a traffic jam.

Random method: After the static tests, we examined how many of the shortest paths went through each arc. In order to successfully test the dynamic scenario, we selected arcs that were used by many paths, since if we had chosen randomly, there would have been no guarantee that the shortest paths would be different. Finally, we multiplied the costs of these arcs by a random number $c \in (1, m]$ for each arc for a parameter m.

Simulating a traffic jam: It is very hard to describe traffic jams mathematically. There are many variables to consider from vehicle density to the number of lanes and weather conditions [15].

Since we only know the travel times on the arcs of the road network we simulated a traffic jam the following way: we chose a random arc (u, v) whose cost was multiplied by a random number $c \in (1, m]$ with probability $1 - p$ and with probability p, the arc was completely blocked. Then iteratively we chose a new arc (a, b), where b is the tail of an already changed arc, and we either block that arc with probability p or multiply its cost with $1 + c \cdot 0.75^i$ where i is the number of arcs between a and the initially selected tail of (u, v), that is u. We choose a total of x arcs.

We also examined what happens when we simulate 5 traffic jams.

Note. We tried other distributions as well. However, the tests show that there is no significant difference between the distributions when they result in similar changes to the arc costs. The only general observation we can make is that the greater the change, the less efficient the methods become.

6.5.2 Contraction Hierarchies

As mentioned before, we have three methods from handling the dynamic scenario in CH. We can use the same order of the nodes to preprocess that graph again, we can try to find the right path by iteratively disabling changed arcs and we can use the CH in an A* algorithm.

Using the same order: We measured the preprocessing time and the speedup compared to Dijkstra's algorithm. The following table contains the results for changing 100 arcs with $m = 5$ and $p = \frac{1}{10}$:

Graph	Method	Update time	Speedup	Original speedup
NY-t	Random	7.0	253.23	258.2
	Jam simulation	5.03	259.05	
HU-t	Random	10.05	355.14	404.18
	Jam simulation	10.127	362.20	
FLA-t	Random	21.763	1119.53	1131.8
	Jam simulation	21.433	1149.98	
	5 Jam	21.649	1130.01	

Table 8: CH results, preprocessing using same order

As we can see, preprocessing the graph using the same order is relatively fast, it can be done under a minute even on larger graphs. The speedups are a little lower compared to the static CH.

We also tested what happens when we increase the number of changed arcs. We ran the test using 1000 random s, t pairs. The following chart shows the measurements on the NY-t graph (the time is in seconds).

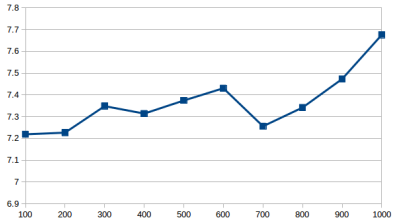

Figure 11: Update time per number of changed arc

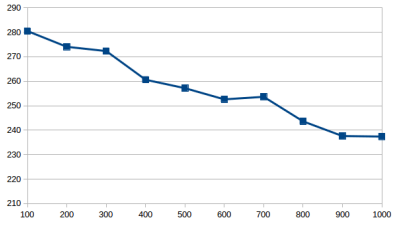

Figure 12: Speedup per changed arc

As the figures suggests, if the number of changed arcs increase, the queries take longer. However, the update time does not seem to be affected significantly, although we can see a small increase. To explain the drop in the update time at 700 changed arcs we have to remember that the construction time depends on the order that we process the nodes in. This order gets worse and worse as we change more arcs. However, it is possible that by changing those additional 100 arcs, the original order became "good" again.

Increasing the number of changed arcs or the value of p or m has similar effects on other graphs as well.

Disabling arcs: Here we tested the method explained in 5.1.1, that is, after running a search on the unmodified CH, we disable the arcs whose path contain changed arcs and run the search again until we find an unchanged path.

We used the same graphs and scenarios as in the previous test. We counted the number of times we had to repeat the search, and since we do not always get the shortest path, we also measured the error of the method. We ran 100 tests and we used $m = 5$ and $p = \frac{1}{10}$.

Graph	Method	Average number of repeated searches	Error
NY-t	Random	2.688	1.08
	Jam simulation	2.43	1.0002
HU-t	Random	4.89	1.086
	Jam simulation	5.64	1.003
FLA-t	Random	3	1.105
	Jam simulation	2.12	1.003
	5 Jam	2.066	1.002

Table 9: CH results, disabling arcs

When selecting the arcs, it is possible that more than one changed arc will be on the tested shortest path, making this method less reliable as shown by the error amounts.

Furthermore, it seems that trying to find an alternate route using CH in a city is easier than doing so in a more rural area as we can see from the number of tries we had to make on the different graphs.

The number of tries we have to make is independent of the severity of the change, since we always run the searches on the same unmodified CH.

Increasing m or p or the number of changed arcs produces worse results. We tested this on NY-t with 1000 changed arcs instead of the original 100. The results were 4.64 average repeats and 1.21 error ratio.

Combining with A*: We ran an A* search on the NY-t graph using the static CH as lower estimates for the distance from the target. Unfortunately, we had to run the CH too many times, so overall the algorithm was slower than a simple Dijkstra algorithm. It took 2.5 times as long to find the correct path.

The performance is similar on other graphs as well.

6.5.3 Highway Node Routing

Here we tested the method described in section 5.2. We also tested the same approach for disabling arcs as we did with CH.

Updating the data structure: We tested the method similarly to the reconstruction of CH using the same order. We changed the length of 100 arcs with $m = 5$ and $p = \frac{1}{10}$ in the case of traffic jam simulation. We also measured the speedup compared to a Dijkstra search.

The following table shows the results of our experiments:

Graph	Method	Update time	Speedup	Original speedup
COL-t	Random	0.72	17.4	17.34
	Jam simulation	0.66	18.27	
FLA-t	Random	0.27	73.47	74.59
	Jam simulation	0.22	71.51	
	5 Jam	0.23	75.84	
CAL-t	Random	4.08	23.62	25.08
	Jam simulation	2.745	25.07	

Table 10: HNR results, updating data structure

Updating the data structure when simulating a traffic jam is faster. This is not surprising, since the traffic jam is local, so we have to run the reconstruction on less nodes in the upper levels.

When we increase m or p or the number of arcs we change, the update times can increase. However, it is unlikely that more than a 100 arcs change their costs in less than 10 seconds.

We can also see that the query times are not significantly affected by the changes.

All in all, the results show that we can compensate for the changing of arc costs quickly even on large graphs.

Disabling arcs: This is the exact same experiment as we did when testing CH. The results are the following:

Graph	Method	Average number of repeated searches	Error
FLA-t	Random	5.7	1.12
	Jam simulation	4.83	1.002
	5 Jam	5.032	1.0001
CAL-t	Random	3.72	1.03
	Jam simulation	2.625	1

Table 11: HNR results, disabling arcs

The results show, that we have to run the algorithm more times than in CH, and the errors are similar. This is due to the fact that we have less levels, which means we have more paths that we can check, since in CH, paths that are not shortest paths sooner or later disappear.

However, since HNR is not as fast as CH, this method is not recommended, as we have to run HNR too many times to get any significant speedup.

6.6 Conclusion

We have seen that adapting algorithms for the dynamic case, which were designed to work well for the static scenario, is difficult. We have to find the balance between the efficiency of the method and the time required to update the data structures behind it.

Using HNR provides small speedups, but we can easily adapt to arc changes in seconds and continue to provide correct results.

Using CH, the update procedure takes longer. During this time, we could use the arc disabling method to quickly provide results with only small errors.

If we have a large server park with multiple machines, we could combine both methods. When there are no changes, we can use CH, and when the CH is updating, we can answer the queries a little slower using the already updated HNR.

References

[1] Schultes, D., Sanders, P. (2007). Dynamic highway-node routing. In Experimental Algorithms, LNCS 4525, (pp. 66-79). Springer Berlin Heidelberg.

[2] Van Wolffelaar, J., & Hoogeveen, J. A. (2010). Highway Node Routing: increasing flexibility and putting it into practice.

[3] Bast, H., Funke, S., Sanders, P., & Schultes, D. (2007). Fast routing in road networks with transit nodes. Science, 316(5824), (pp. 566-566).

[4] Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Experimental Algorithms (pp. 319-333). Springer Berlin Heidelberg.

[5] Geisberger, R., Sanders, P., Schultes, D., & Vetter, C. (2012). Exact routing in large road networks using contraction hierarchies. Transportation Science, 46(3), (pp. 388-404).

[6] Funke, S., & Storandt, S. (2013, June). Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In Sixth Annual Symposium on Combinatorial Search.

[7] Dibbelt, J., Strasser, B., & Wagner, D. (2014). Customizable contraction hierarchies. In Experimental Algorithms (pp. 271-282). Springer International Publishing.

[8] Schultes, D. (2008, February). Route Planning in Road Networks. In Ausgezeichnete Informatikdissertationen (pp. 271-280).

[9] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., ... & Werneck, R. (2014). Route planning in transportation networks. In Technical Report MSR-TR-2014-4. Microsoft Research, Microsoft Corporation.

[10] Delling, D., Sanders, P., Schultes, D., Wagner, D. (2009). Engineering route planning algorithms. In Algorithmics of large and complex networks (pp. 117-139). Springer Berlin Heidelberg.

[11] Sanders, P., & Schultes, D. (2005). Highway hierarchies hasten exact shortest path queries. In Algorithms–Esa 2005 (pp. 568-579). Springer Berlin Heidelberg.

[12] Sanders, P., & Schultes, D. (2006). Engineering highway hierarchies. In Algorithms–ESA 2006 (pp. 804-816). Springer Berlin Heidelberg.

[13] Sanders, P., & Schultes, D. (2007). Engineering fast route planning algorithms. In Experimental Algorithms (pp. 23-36). Springer Berlin Heidelberg.

[14] Sanders, P., & Schultes, D. (2012). Engineering highway hierarchies. Journal of Experimental Algorithmics (JEA), 17, (pp. 1-6).

[15] Bellomo, N., & Dogbe, C. (2011). On the modeling of traffic and crowds: A survey of models, speculations, and perspectives. (pp. 409-463). SIAM review, 53(3),

[16] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. Systems Science and Cybernetics, IEEE Transactions on, 4(2), (pp. 100-107).

[17] Ohshima, T., Eumthurapojn, P., Zhao, L., Nagamochi, H. (2010). An A* Algorithm Framework for the Point-to-Point Time-Dependent Shortest Path Problem. CGGA (pp. 154-163).

[18] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische mathematik, 1(1), (pp. 269-271).

[19] http://www.cs.elte.hu/blobs/diplomamunkak/bsc_alkmat/2013/gobor_daniel.pdf

[20] http://lemon.cs.elte.hu/trac/lemon

[21] http://lemon.cs.elte.hu/pub/doc/latest/index.html

[22] http://www.dis.uniroma1.it/challenge9/