

Review of “Finding Fastest Paths on A Road Network with Speed Patterns”

Matthias Frei

May 25, 2010

Contents

1	Introduction	1
2	Problem Definition	2
3	Shortest Path Computation	4
4	Fastest Path Computation	4
4.1	Travel Time	5
4.2	The IntAllFastestPaths Algorithm	6
4.2.1	Overview	6
4.2.2	Expanding a Path	7
4.2.3	Treatment of Piecewise Linear Functions	7
4.2.4	The Query Result	9
5	Example	9
6	Implementation and Experiments	12
7	Assessment	12

1 Introduction

Many modern cars have navigation systems built in. These systems calculate the shortest route from some starting point to a destination. However, they usually either ignore the driving speed on the road segments or assume that it is constant. In both cases, the calculated route is independent from the desired leaving time. This is not always a good choice since the *fastest* route to get somewhere might not be the same on a Sunday afternoon and during Monday rush-hour. To model these changes, it has been proposed to consider the velocity on a road segment to be a piecewise constant function of the time ([1]). Additionally, one can categorize days and use a separate velocity function for every category, assuming that the traffic on a road segment is equal for two days in the same category. The algorithm proposed to solve queries on a

such a network in a *continuous time model* will make it necessary to work with continuous functions.

2 Problem Definition

First we describe the input on which the problem will be defined.

Definition 1 (Category Set). A (day-)category set D is a set of categories such that each day belongs to exactly one category in D .

Definition 2 (CapeCod pattern). Given a category set D , a categorized piecewise constant speed (CapeCod) pattern consists of one piecewise constant speed function s_d

$$s_d : T \rightarrow \mathbb{R}^+$$

for each category $d \in D$, where $T \subset \mathbb{R}$ is the time of day.

Definition 3 (CapeCod network). Given a category set D , a CapeCod network is a directed graph $G(N, E)$. $N = \{(n_i, loc_i) | i \in 1, \dots, m\}$ is the set of nodes with loc_i their spatial locations. $E = \{(n_i, n_j, d_{i,j}, p_{i,j}) | i, j \in 1, \dots, m\}$ is the set of edges $n_i \rightarrow n_j$ where $d_{i,j}$ is the distance and $p_{i,j}$ is the CapeCod pattern assigned to the edge.

For example, let our network consist of the roads in a city, some rural roads and a highway to a sunny place.¹ An example for a day-category set may be: *workday*, *weekend*. A CapeCod pattern for a city road for this category set might be that for workdays, speed is 30km/h during the rush-hour and 50km/h during off-peak hours. For weekend, the speed could e.g. be 50km/h all day. On the highway, speed is maybe 80km/h during workday rush-hour and 120km/h during off-peak hours. Since everybody wants to go to the beach on weekends, the highway will be heavily congested and the speed will only be 30km/h. Since there is not a lot of variation in the amount of traffic on the rural roads, the speed is constantly 60km/h.

Definition 4 (singleFP query). Given a CapeCod network $G(N, E)$, a start node $s \in N$, an end node $e \in N$, and a leaving time interval I , the *single fastest paths* query returns a time instant $t \in I$ and the corresponding fastest path from s to e such that leaving from s at time t minimizes travel time from s to e .

Besides this simple query, which returns a path with minimal travel time and the corresponding leaving time, there is a slightly more general query which calculates the travel time function and the fastest path for *every* leaving time during the queried interval:

Definition 5 (allFP query). Given a CapeCod network $G(N, E)$, a start node $s \in N$, an end node $e \in N$, and a leaving time interval I , the *all fastest paths* query returns a full partitioning of $I : I_1, \dots, I_k$ where each sub-interval is associated with a fastest path such that two leaving time instants in the same sub-interval lead to the same fastest path and two leaving time instants in two adjacent sub-intervals lead to different fastest paths.

¹Note that this categorization of the roads is not the categorization we are talking about in the CapeCod patterns. It is just a shorthand to not have to define the CapeCod pattern for each road separately.

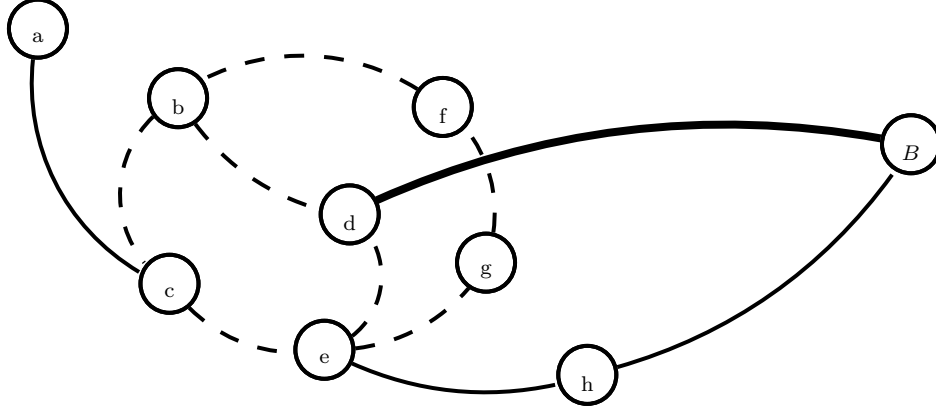


Figure 1: An example CapeCod network. The dashed lines represent city roads, the thin lines represent rural roads and the bold one represents a highway. Note that by definition, the graph is directed and every edge in the figure represents two directed edges.

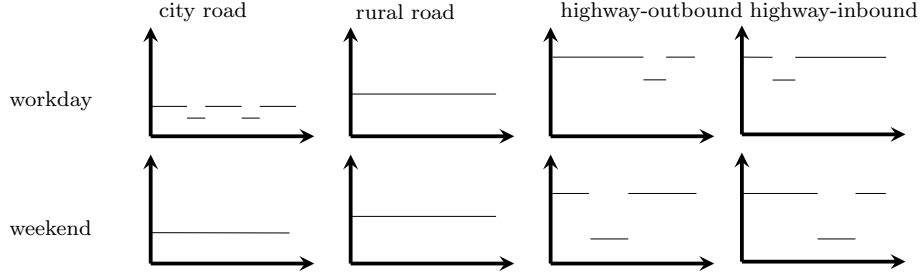


Figure 2: The CapeCod patterns corresponding to our example

Hence, the answer to the singleFP query is a part of the answer to the allFP query. In fact we will see that we can use the same algorithm for both queries.

Using the example network from before, we give an example for an allFP and a singleFP query.

A singleFP query might be “I want to go from my house near node e to the beach at node B on Saturday, leaving sometime between 6am and 10am. Give me the route and the leaving time such that the journey is as short as possible.” The answer to this query might be to leave at 6am and travel along the route $e \rightarrow d \rightarrow B$.²

Similarly an allFP query may be posed as “I want to go from my house near node e to the beach at node B on Saturday. Give me a listing which tells me the fastest route for every moment between 6am and 10am.” The answer would be to take the route $e \rightarrow d \rightarrow B$ when leaving between 6am and 7am and to take the route $e \rightarrow h \rightarrow B$ along the rural roads when leaving after 7am.

²Of course the answer can not be given exactly, since we did not specify exact edge length, node positions etc.

3 Shortest Path Computation

The algorithm to solve the singleFP and the allFP queries is based on the A* algorithm. To get a better insight into fastest path computation, we will quickly summarize this well known algorithm for shortest path computation.

In shortest path computation, each edge has assigned a certain cost and we want to minimize the sum of the costs of the edges along the path from the start node to the end node.

A* is based on Dijkstra's algorithm. Dijkstra uses a priority queue of nodes, ordered by the cost for the best known path from the start node to the particular node. The node with the lowest cost is picked from the priority queue. We expand this node by checking whether the paths via the node to its neighbours is shorter than the previously known paths to the neighbours. If a shorter path is found, the cost value of the particular neighbour is updated correspondingly. Since we picked the node with the lowest cost value, we know that there can not be a shorter path to this node via some node that is still in the queue. Therefore we put it into a set of 'closed' nodes which do not have to be considered as neighbours anymore when expanding some node. The computation ends as soon as the destination node is added to the 'closed' set. The running time of this algorithm depends highly on the implementation of the priority queue. Using a simple unsorted linked list as priority queue, the running time is about $\mathcal{O}(|N|^2 + |E|)$.

A* is basically the same algorithm but it adds a lower bound heuristic for the remaining cost of the path from a node to the destination. We add this estimated distance to the cost when inserting a node into the queue, such that we will not anymore pick the node with the shortest path from the start to the particular node, but pick the node for which the cost from the start to the node plus the estimated cost of the rest of the path will be minimal. The idea is to 'guide' the overall direction of the search. The estimation must always be lower than the real cost and it should be monotonic. This ensures that it is still true that whenever we pick a node from the queue, there can not be a shorter path to this node via some node that is still in the queue. This allows to use the 'closed' set. The worst-case running time is the same as for Dijkstra's algorithm, however A* might perform better in practice - the heuristic helps A* to narrow the set of nodes that will potentially lie on a shortest path. Note that if no 'closed' set can be used (e.g. if the heuristic is not monotonic) the A* algorithm degenerates to a procedure that potentially looks at all possible paths, only guided by its heuristic. Since the number of possible paths is exponential in the number of nodes, this version of the A* algorithm is considerably slower, especially on larger graphs.

4 Fastest Path Computation

The algorithm to solve an allFP query will be called *IntAllFastestPaths* and is based on the A* algorithm. The *IntAllFastestPaths* algorithm will minimize the *travel time*, in contrast to the normal A* algorithm which minimizes the distance. This is an important difference, since in a CapeCod network, the travel time is a *function* of the *leaving time*. So we will have to deal with continuous functions instead of single values in each step of the algorithm. Hence the

properties of those travel time functions and the techniques to use them in the fastest path search algorithm will be one focus in this section.

In this section, we will use the notation that the query is given by a CapeCod network $G(N, E)$, a start node $s \in N$, an end node $e \in N$, and a leaving time interval I .

4.1 Travel Time

The travel time is the time needed to travel along a path in the CapeCod network, driving at exactly the speed given by the CapeCod pattern of the particular edges and the current time. We define the travel time to be function of the leaving time. As an example, we derive a travel time function for one network segment: Consider an edge $(n_i, n_j, d_{i,j}, p_{i,j}) \in E$ and let $p_{i,j}(t) = v_1$ for $t \in [t_1, t_2]$ and $p_{i,j}(t) = v_2$ for $t \geq t_2$. The travel time function $T(t \in [t_1, t_2], n_i \rightarrow n_j)$ can be written as

$$T(t \in [t_1, t_2], n_i \rightarrow n_j) = \begin{cases} \frac{d_{i,j}}{v_1} & \text{if } t \in [t_1, t_2 - \frac{d_{i,j}}{v_1}) \\ (t_2 - t) + \frac{d_{i,j} - v_1(t_2 - t)}{v_2} & \text{if } t \in [t_2 - \frac{d_{i,j}}{v_1}, t_2] \end{cases}$$

Observe that until $t_2 - \frac{d_{i,j}}{v_1}$, an object traversing the edge can travel at speed v_1 on the complete distance. After this time, the object can travel at v_1 for some fraction of the distance but then the speed changes to v_2 for the rest of the distance. This leads to a *piecewise linear* travel time function.

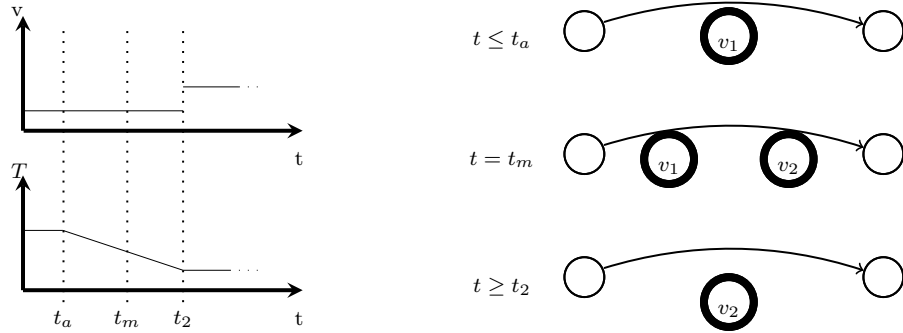


Figure 3: Visualization of the travel time example. On top left we see the speed pattern and below is the resulting travel time function. t is the leaving time and $t_a = t_2 - \frac{d_{i,j}}{v_1}$. t_m is just an arbitrary time between t_a and t_2 .

In general the travel time function depends also on the size of the neighbouring time intervals and the speed patterns for these intervals. This formalization could obviously be generalized to an arbitrary number of time intervals and for every time instance. We observe that the travel time function is *piecewise linear*. This will be important during the fastest path computation and makes the representation of travel time functions easy.

4.2 The IntAllFastestPaths Algorithm

The IntAllFastestPaths algorithm is a simple algorithm that has been proposed to solve the allFP query ([1]). As already mentioned, the singleFP query can be answered using the same algorithm since the answer to the singleFP query is a part of the answer to the allFP query.

The basic structure of the IntAllFastestPaths algorithm is very similar to the A* algorithm. Hence we need a priority queue of expanded paths, a lower bound estimator for the remaining travel time and some rules about how to expand a path.

4.2.1 Overview

Since the allFP and the singleFP query involve a time interval, we can not use A* directly. When we expand a path in A*, we choose the neighbour for which the resulting cost (including the estimated cost for the rest of the path) is minimal. The problem is that the leaving time is not a single value and for different leaving times, different neighbours must be picked. The main ideas of the IntAllFastestPaths algorithm are:

- Maintain a priority queue of expanded paths starting from s . For each path $s \Rightarrow n_i \equiv s \rightarrow \dots \rightarrow n_i$ ³, we maintain the piecewise linear function $T(s \Rightarrow n_i, t \in I) + T_{est}(n_i \Rightarrow e)$ where $T_{est}(n_i \Rightarrow e)$ is a lower bound estimator for the remaining travel time. A simple lower bound estimator is given by the Euclidian distance between n_i and e divided by the maximum speed in the network. We will later see a more elaborate estimator which will use the precomputed shortest path distance instead of the Euclidian distance.
- In each step we pick a path to expand from the priority queue. We pick the path for which the *minimum of the travel time function* is *minimal for all paths*. As in the A* algorithm, this choice is essential to ensure that the computed path is optimal.
- Maintain a special travel time function $T_{lb}(t \in I)$ called the *lower border function*. It is defined as the lower border of the travel time functions of all identified paths (i.e. paths already picked from the priority queue) that end in e . Hence this function consists of multiple travel time functions, each corresponding to some path $s \Rightarrow e$ and some subinterval of I during which this path is the fastest.
- The priority queue is initialized with a path that represents the unexpanded node s . The lower border function T_{lb} is initialized by setting it to infinity.
- Stop if either the priority queue is empty or if the path picked to expand has a minimum value that is not lower than the maximum value of the lower border function. The lower border function is the answer to the allFP query.

³We use the shorthand notation $s \Rightarrow n_i$ for $s \rightarrow \dots \rightarrow n_i$ to denote some *specific* path from s to e .

In the case of a singleFP query we can use the same algorithm and stop as soon as some path from s to e is picked from the priority queue. The optimal leaving time is then (obviously) given by the time at which the travel time function for this path is minimal.

4.2.2 Expanding a Path

Every entry of the priority queue consists of a path from s to n_i and the corresponding estimated lower bound travel time function $T(s \Rightarrow n_i, t \in I) + T_{est}(n_i \Rightarrow e)$. To ‘expand’ this path, we collect all neighbours n_j of n_i and calculate the travel time functions for the paths $s \Rightarrow n_i \rightarrow n_j$. This function will be given by the time needed to get to n_i plus the time needed to get from n_i to n_j . In section 4.2.3 we will formalize this idea and describe how the resulting function can be calculated. Besides this travel time function, we also have to compute the lower bound estimator $T_{est}(n_j \Rightarrow e)$ for the remaining path from n_j to e . Knowing the function $T(s \Rightarrow n_i \rightarrow n_j, t \in I) + T_{est}(n_j \Rightarrow e)$ we put the path $s \Rightarrow n_i \rightarrow n_j$ into the priority queue.

Note that this is different from A*. In A*, we check whether the cost for the path from s via n_i to n_j is lower than the cost for all previously known paths from s to n_j . The travel time function is not anymore a scalar value as the cost. Hence we either have to take the minimum of the functions and remember the correct predecessor node for each subinterval of I , or we put each path to n_j separately into the priority queue and do the ‘clean-up’ only in the end. In [1], the latter approach was chosen.⁴ Observe that in either case, we can not have a ‘closed’ set as we had in the A* algorithm. As already mentioned in section 3, this may lead to an exponential number of paths being expanded.

This ‘clean-up’ is nothing but constructing the lower border function T_{lb} from the paths ending in e , which implies that we need a special case to expand paths ending in e . This will be discussed in section 4.2.4.

4.2.3 Treatment of Piecewise Linear Functions

During the IntAllFastestPaths algorithm we have to combine piecewise linear functions in a very specific way: given the travel time function $T(t \in I, s \rightarrow \dots \rightarrow n)$ of a path from node s to node n and the travel time function $T(t, n \rightarrow m)$ for the edge from n to some neighbour node m of n , we want to calculate the travel time function $T(t \in I, s \rightarrow \dots \rightarrow n \rightarrow m)$ for the whole path from s via n to m . It is easy to see, that for each $t \in I$, $T(t, s \rightarrow \dots \rightarrow n \rightarrow m)$ is equal to $T(t, s \rightarrow \dots \rightarrow n) + T(t', n \rightarrow m)$, where t' is the time at which node n is reached, i.e. $t' = t + T(t, s \rightarrow \dots \rightarrow n)$. Observe that the new travel time function for the whole path is still piecewise linear.

To compute this combination, we store the piecewise linear travel time functions (at least conceptually) as a sequence of consecutive time intervals with a linear function assigned to. Now, to combine the two piecewise linear functions $T_{s,n}(t \in I) := T(t \in I, s \rightarrow \dots \rightarrow n)$ and $T_{n,m}(t) := T(t, n \rightarrow m)$ to $T_{s,m}(t \in I) := T_{s,n}(t) + T_{n,m}(t + T_{s,n}(t))$, we first have to identify

⁴This is not very clear from the description of the algorithm in [1]. The authors mention, that this approach would lead to “an exponential number of paths being expanded regardless the size of the answer set” and that they would propose the IntAllFastestPath algorithm instead. Surprisingly enough, IntAllFastestPath seems to use exactly this approach.

1. the time interval $I_{n,m}$ during which access to the function $T_{n,m}$ is needed,
2. the points in time $t_1, \dots, t_n \in I$ at which the resulting function $T_{s,m}$ will change from one linear piece to another, and
3. the linear function between two identified change points.

The first thing is easy. The time interval $I_{n,m}$ during which we need information about $T_{n,m}$ is given by $[\min_{t \in I}(t + T_{s,n}(t)), \max_{t \in I}(t + T_{s,n}(t))]$. We can easily maintain these two extrema during the combination. Note that this is not the same minimum as the one we need to maintain for the priority queue.

For 2), we identify two cases. We say that a function ‘changes’ if it changes from one linear piece to another.

- If $T_{s,n}$ changes at time t_c , then $T_{s,m}$ may⁵ change at time t_c .
- If $T_{n,m}$ changes at time t_c , then $T_{s,m}$ may change at all time points t with $t_c = t + T_{s,n}(t)$.

The linear function for 3) is easy to derive. Let $t \mapsto (\alpha_{s,n} \cdot t + \beta_{s,n})$ be a linear segment of $T_{s,n}$ and let $t \mapsto (\alpha_{n,m} \cdot t + \beta_{n,m})$ the corresponding segment of $T_{n,m}$. Then the new segment of $T_{s,m}$ is given by

$$\begin{aligned} t &\mapsto (\alpha_{s,n} \cdot t + \beta_{s,n} + \alpha_{n,m}(t + \alpha_{s,n} \cdot t + \beta_{s,n}) + \beta_{n,m}) \\ \iff t &\mapsto (\alpha_{s,n} \cdot t + \alpha_{n,m}(1 + \alpha_{s,n}) \cdot t + (\alpha_{n,m} \cdot \beta_{s,n} + \beta_{s,n} + \beta_{n,m})) \\ \iff t &\mapsto ((\alpha_{s,n} + \alpha_{n,m}(1 + \alpha_{s,n})) \cdot t + (\alpha_{n,m} \cdot \beta_{s,n} + \beta_{s,n} + \beta_{n,m})) \end{aligned}$$

Using 1), 2) and 3), we can describe an algorithm that calculates $T_{s,m}$ from $T_{s,n}$ and $T_{n,m}$.

Algorithm 1 Combine $T_{s,n}(t \in I)$ and $T_{n,m}$ to $T_{s,m}(t \in I)$

```

1:  $I_{n,m} := [\min_{t \in I}(t + T_{s,n}(t)), \max_{t \in I}(t + T_{s,n}(t))]$ 
2:  $H_c := ChgPts(T_{s,n}, I) \cup \{t | t + T_{s,n}(t) = t_c \wedge t_c \in ChgPts(T_{n,m}, I_{n,m})\}$ 
3:  $ChgPts(T_{s,m}, I) := H_c$ 
4: for all  $\{t_i, t_{i+1}\} \in ChgPts(T_{s,m}, I)$  do
5:    $(t \mapsto (\alpha_{s,n} \cdot t + \beta_{s,n})) := LinFunc(T_{s,n}, [t_i, t_{i+1}])$ 
6:    $(t \mapsto (\alpha_{n,m} \cdot t + \beta_{n,m})) := LinFunc(T_{n,m}, [t_i + T_{s,n}(t_i), t_{i+1} + T_{s,n}(t_{i+1})])$ 
7:    $\alpha_{s,m} := \alpha_{s,n} + \alpha_{n,m}(1 + \alpha_{s,n})$ 
8:    $\beta_{s,m} := \alpha_{n,m} \cdot \beta_{s,n} + \beta_{s,n} + \beta_{n,m}$ 
9:    $LinFunc(T_{s,m}, [t_i, t_{i+1}]) := (t \mapsto (\alpha_{s,m} \cdot t + \beta_{s,m}))$ 
10: end for
```

This is the approach taken by the authors of [1]. Its problems are ‘hidden’ in line 2. Let N and M be the number of change points of $T_{s,n}$ and $T_{n,m}$ respectively. We have to calculate $N \cdot (M - 1)$ possible line intersections and, even worse, as many as $N + N \cdot (M - 1)$ change points for the new function may be created.⁶ Since we will apply this algorithm every time when we add a new edge, the travel time function of a path with e edges consists of at most c^e change points where c is the number of change points of the travel time functions of the edges. This will obviously lead to a very bad worst-case running time.

⁵Of course it may happen that $T_{s,m}$ does not change at this point - the newly calculated linear segments to the left and to the right of the point may coincide.

⁶For example if $T_{s,n}$ is some sort of a zig-zag function.

4.2.4 The Query Result

The result of the allFP query is the lower border function T_{lb} which directly corresponds to a full partitioning of the leaving time interval. T_{lb} is the minimum of the travel time functions of all identified paths ending in e . It consists of multiple travel time functions, each corresponding to some path from s to e and some subinterval of I during which this path is the fastest.

To calculate T_{lb} (and hence to calculate the result of the allFP query) we have to define a special rule for path expansion. When we pick a path that ends in e from the priority queue, we do not expand it normally but add it to the lower border function. For this we have to calculate the minimum of the travel time function of the new path and T_{lb} . Since both functions are piecewise linear, calculating the minimum and the corresponding partitioning of I is easy and can be achieved by linearly traversing all change-points of both functions and calculating the intersection of the linear pieces between those change-points where the minimum changes from one function to the other.

We do not have to proceed with expanding paths, as soon as we pick a path $s \Rightarrow n_i$ from the priority queue for which the *minimum* of the function $T(s \Rightarrow n_i, t) + T_{est}(n_i \Rightarrow s)$ exceeds the *maximum* of T_{lb} . There can not be a path left in the priority queue, for which the travel time from s to e is lower than for any known path. This is based on the following observations:

- T_{lb} represents the travel time of the fastest expanded path for every time instant.
- We always pick the path for which the minimum of $T(s \Rightarrow n_i, t) + T_{est}(n_i \Rightarrow s)$ is minimal amongst the paths in the priority queue
- $T_{est}(n_i \Rightarrow e) \leq T(n_i \Rightarrow e, t \in I)$

For the singleFP query we can stop as soon as we pick a path from s to e from the priority queue. We report the time at which the travel time function of this path is minimal as the answer to the singleFP query. With the same arguments as before, there can not be a path left in the priority queue for which this minimum would be smaller.

5 Example

In this section we will work through an example problem. We will discuss a singleFP and an allFP query in very simple CapeCod network consisting of only 4 nodes. For simplicity, we use very unrealistic values for distances and speeds and we assume that everything happens on one single day so that we do not have to consider different categories. Furthermore, we assume that we use Euclidian distance divided by the maximum speed in the network as a lower bound estimator. The example network is given in figure 4 and the corresponding speed patterns are given in figure 5.

In the following, we will calculate the answer for both the singleFP and the allFP query asking for the fastest path(s) from s to e with the leaving time interval $[0, 5]$.

Before we calculate the answer to the query, we precompute the travel time functions for the separate edges (see figure 6).

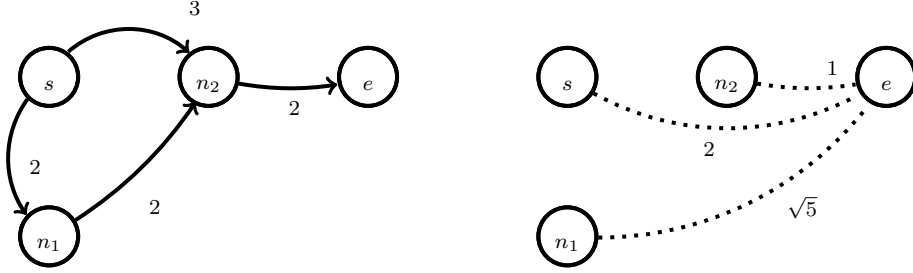


Figure 4: The network used in the example. On the left hand side, we see the edge lengths, on the right hand side, we see the Euclidian distances between the nodes and node e .

1. Initialize the priority queue with the ‘path’ s and the corresponding function $T(s, t) + T_{est}(s \Rightarrow e) = 0 + T_{est}(s) = 0 + \frac{dist(s, e)}{v_{max}}$. We initialize the lower border function by setting it to infinity.
2. Pick the path s from the queue. Expand its neighbours n_1 and n_2 by putting the paths $s \rightarrow n_1$ and $s \rightarrow n_2$ into the queue together with the corresponding functions. Since $T(s, t)$ is identically zero, the travel time functions of those paths are exactly the travel time of the particular edges.
3. Pick the path $s \rightarrow n_1$ from the priority queue since the minimum of its function is minimal for all paths in the queue. The only neighbour we have to look at is n_2 . Therefore we add the path $s \rightarrow n_1 \rightarrow n_2$ to the queue. The corresponding travel time function is computed using the algorithm described in section 4.2.3 (see figure 7).
4. Pick the path $s \rightarrow n_1 \rightarrow n_2$ from the priority queue. Expand this path to $s \rightarrow n_1 \rightarrow n_2 \rightarrow e$ and put it in the priority queue.
5. Pick the path $s \rightarrow n_1 \rightarrow n_2 \rightarrow e$ from the priority queue. Since it ends in e , merge it into the lower border function to expand the path. Regarding only the singleFP query, we can stop here and report the optimal leaving time is any time in $[0, 1] \cup [4, 5]$.
6. For the allFP query, we have to do some more work. Expand the path $s \rightarrow n_2$ to $s \rightarrow n_2 \rightarrow e$.
7. Pick $s \rightarrow n_2 \rightarrow e$ from the queue and merge it into the lower border function. Since the priority queue is now empty, the IntAllFastestPaths algorithm terminates here and reports the lower border function and the corresponding paths as the answer to the allFP query.

The algorithm would also have terminated, if there were still some paths left in the queue for which minimum of the estimated travel time was larger than the maximum of the lower border function.

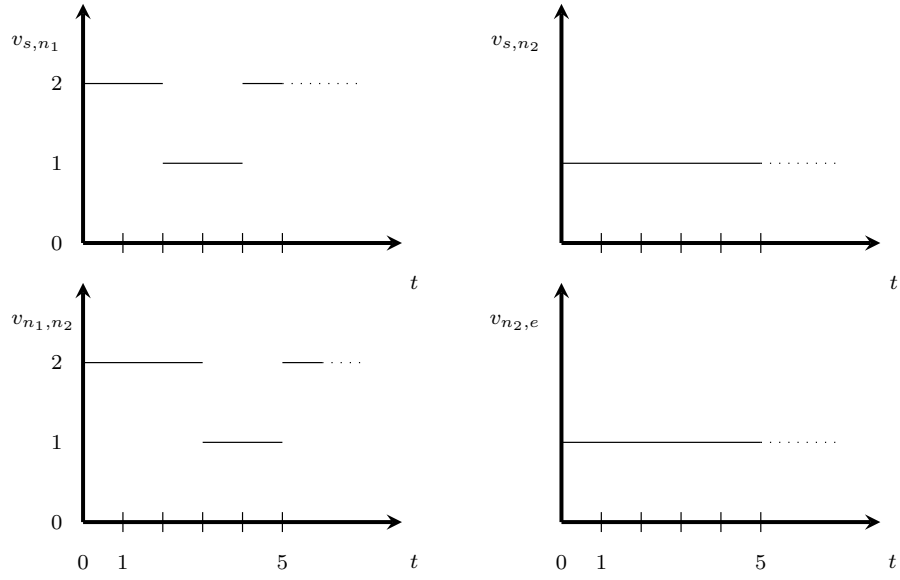


Figure 5: The piecewise constant velocity functions.

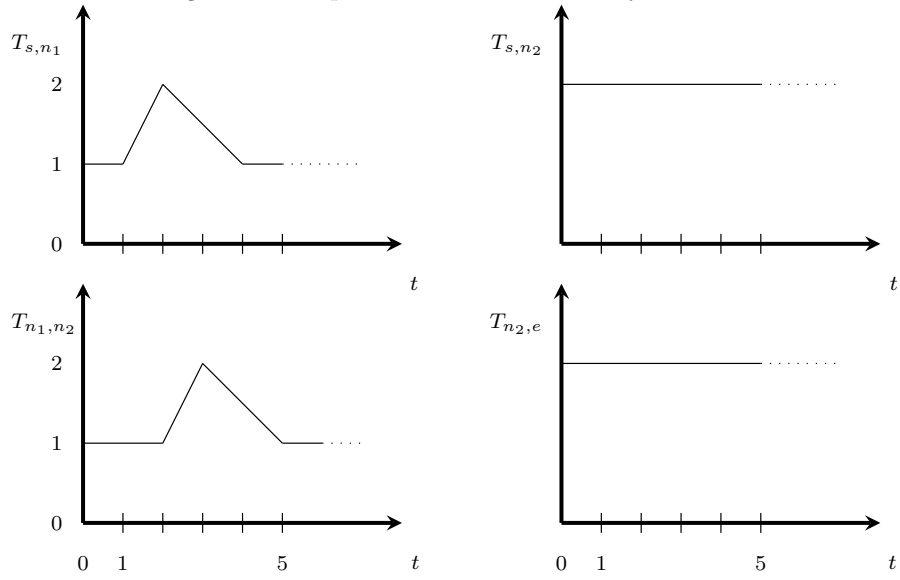


Figure 6: The piecewise linear travel time functions for the edges.

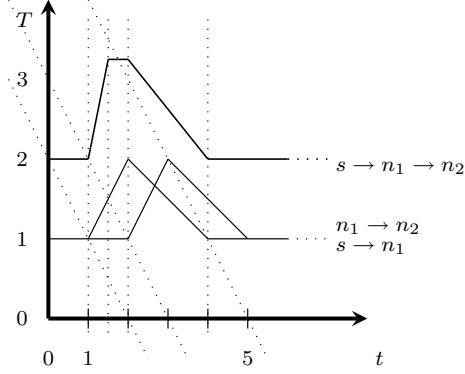


Figure 7: Graphical construction of the travel time function of the path $s \rightarrow n_1 \rightarrow n_2$ from the travel time functions of the path $s \rightarrow n_1$ and the edge (n_1, n_2) . The dashed diagonal lines represent the equations $t_c = t + T(s \rightarrow n_2, t)$ where t_c is a point at which T_{n_1, n_2} changes from one linear piece to another. The vertical dashed lines represent the change points of the new function

6 Implementation and Experiments

Assuming that the network is large and needs to be stored on disk, the connectivity-cluster access method (CCAM) ([2]) can be adopted to store and access the network information.

Another implementation improvement is to use a more sophisticated estimator $T_{est}(n \Rightarrow e)$. An example for this is the *boundary-node distance estimator*. It uses a precomputed estimation of the shortest path between two nodes instead of using the Euclidian distance.

Some early experiments compare the IntAllFastestPaths algorithm to variations of itself and to discrete time algorithms ([1]). In particular, the boundary-node estimator was compared to the naive Euclidian distance estimator and it can be observed that the boundary-node estimator outperforms the naive implementation just as expected. Another comparison is done between the IntAllFastestPaths algorithm and the A* algorithm using a discrete time model. For this, A* is applied repeatedly on the problem with different, single value leaving times. Depending on the degree of discretization, A* is either faster and less accurate or slower and about equally accurate.

Later experiments compare the IntAllFastestPaths algorithm to more advanced algorithms for the singleFP problem ([3]). Those experiments clearly show the exponential nature of the IntAllFastestPaths algorithm which is clearly outperformed by the other tested algorithms on larger graphs.

7 Assessment

In section 2 we introduced the CapeCod patterns, the CapeCod networks and the two queries allFP and singleFP, and in section 4 we saw how to compute the answer to such a query. We did however not discuss whether or not those models are well suited for the applications they were proposed for and whether or not the techniques proposed to solve the queries are appropriate. Discussing

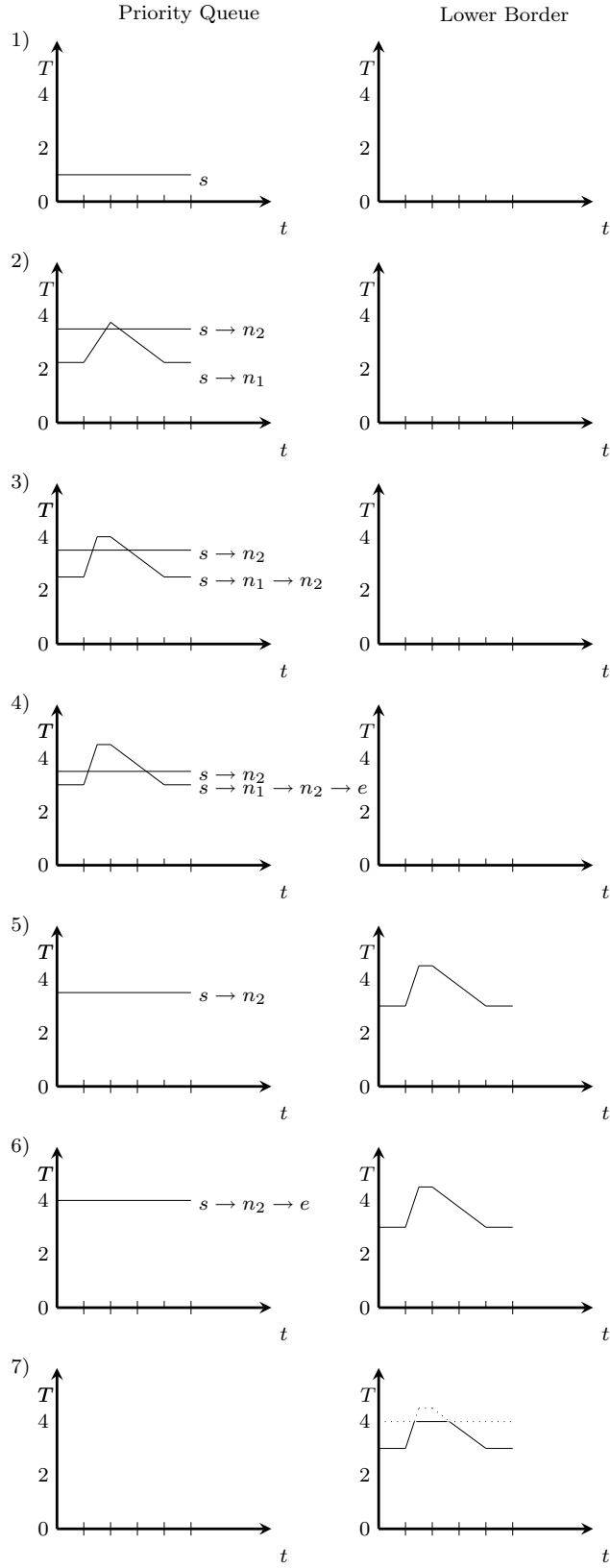


Figure 8: The priority queue and the lower border function after each step.

those questions might give insight in the reasons why those models and methods have been chosen.

Concerning the CapeCod model, it is at least questionable whether piecewise constant speed is not an oversimplification for street traffic. However, the use of piecewise constant speed can be justified by the lack of more precise statistical data, which would make a more complex model useless anyway. Considering this, one could also think, that the ‘complicated’ piecewise linear functions aren’t of much use either - if we define the travel time function as piecewise constant, the resulting data can not be much worse. It is important to notice that such a model would have to introduce the possibility to wait at nodes, since otherwise the assumption that a optimal path consists of optimal subpaths would be violated. Since the application of the model is mainly street traffic, waiting at nodes would obviously not be very realistic. For this reasons, choosing piecewise constant functions for the velocity is probably the simplest approach to model street traffic - we have varying velocities without having to introduce arbitrary waiting at the crossings.

The choice of the proposed queries is not obvious as well. The singleFP query seems to be clear, but the allFP query isn’t. It is hard to think of a case in which one would want to query this information and it adds a lot of complexity to the problem. The authors of [1] don’t argue why this exact query is of interest.

The IntAllFastestPaths algorithm that was described in section 4.2 has several severe disadvantages, for example the need to store whole paths in the queue, not having a ‘closed’ set etc. Considering more recent contributions (as in [3]) which are clearly more advanced, the IntAllFastestPaths algorithm may be regarded as a first, simplistic approach to a newly defined problem.

References

- [1] E. Kanoulas, Y. Du, T. Xia, D. Zhang. Finding Fastest Paths on A Road Network with Speed Patterns. *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, 2006
- [2] S. Shekhar, D.-R. Liu. CCAM: A Connectivity Clustered Access Method for Networks and Network Computations. *TKDE*, 9(1):102-119, 1997
- [3] B. Ding, J. Xu Yu, Lu Qin. Finding Time-Dependent Shortest Paths over Large Graphs. *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, 2008