Quickest Paths: Faster Algorithms and Dynamization

(Preliminary Version)*

Dimitrios Kagaris [1] Grammati E. Pantziou [1,2] Spyros Tragoudas [3]
Christos D. Zaroliagis [2,4]

April 18, 1994

(1) Computer Science Program, Dartmouth College, Hanover NH 03755, USA
(2) Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
(3) Computer Science Dept., Southern Illinois University, Carbondale IL 62901, USA
(4) Max-Planck Institut für Informatik, Im Statwald, 66123 Saarbrücken, Germany

Abstract

Given a network $N = (V, E, c, l)$, where $G = (V, E)$, $|V| = n$ and $|E| = m$, is a directed graph, $c(e) > 0$ is the capacity and $l(e) \geq 0$ is the lead time (or delay) for each edge $e \in E$, the quickest path problem is to find a path for a given source–destination pair such that the total lead time plus the inverse of the minimum edge capacity of the path is minimal. The problem has applications to fast data transmissions in communication networks. The best previous algorithm for the single–pair quickest path problem runs in time $O(rm + rn \log n)$, where r is the number of distinct capacities of N [12]. In this paper, we present algorithms for general, sparse and planar networks that have significantly lower running times. For general networks, we show that the time complexity can be reduced to $O(r^*m + r^*n \log n)$, where r^* is at most the number of capacities greater than the capacity of the shortest (with respect to lead time) path in N. For sparse networks, we present an algorithm with time complexity $O(n \log n + r^*n + r^*\tilde{\gamma} \log \tilde{\gamma})$, where $\tilde{\gamma}$ is a topological measure of N. Since for sparse networks $\tilde{\gamma}$ ranges from 1 up to $\Theta(n)$, this constitutes an improvement over the previously known bound of $O(rn \log n)$ in all cases that $\tilde{\gamma} = o(n)$. For planar networks, the complexity becomes $O(n \log n + n \log^3 \tilde{\gamma} + r^*\tilde{\gamma})$. Similar improvements are obtained for the all–pairs quickest path problem. We also give the first algorithm for solving the dynamic quickest path problem.

1 Introduction

Consider a network $N = (V, E, c, l)$, where $G = (V, E)$ is a directed graph, $c : E \to Z^+$ is the capacity function and $l : E \to \Re^*$ is the delay function. The nodes represent transmitters/receivers without data memories and the edges represent communication channels. The capacity $c(e)$ of an edge $e \in E$ represents the amount of data that can be transmitted in a time unit through e. The delay $l(e)$ of an edge $e \in E$ represents the time required for the data to traverse edge e. If σ units of data are to be transmitted from a node u to a node v through edge $e = (u, v)$, then the required

*This work is partially supported by the EEC ESPRIT Basic Research Action No. 7141 (ALCOM II). The work of the second author is also partially supported by the NSF postdoctoral fellowship No. CDA-9211155.

transmission time is $l(e) + \frac{\sigma}{c(e)}$. To see this, observe that since at most $c(e)$ units of data can pass through e in a time unit, all σ units of data will have been pumped out of node u after $\frac{\sigma}{c(e)}$ time units. The last data to leave node u will depart from u at time instant $\frac{\sigma}{c(e)}$ and will arrive at node v at time instant $l(e) + \frac{\sigma}{c(e)}$. Since in the meanwhile all previous data have already been transmitted from u to v in a pipelined fashion, the total transmission time is $l(e) + \frac{\sigma}{c(e)}$. Due to the pipelined nature of the transmission, the delay time $l(e)$ is also characterized as the *lead* time of e.

Let $p = (v_1, v_2, ..., v_k)$ be a path from node v_1 to node v_k. The *capacity of path* p is defined as the minimum capacity of the path edges and is denoted by $c(p) = \min_{1 \leq i \leq k-1} c(u_i, u_{i+1})$. The definition of the path capacity is motivated by the fact that, since the nodes have no data memories, all data that are received by a node in a time unit must be pumped out of the node in the next time unit. The *lead time of path* p is defined as the sum of the lead times of the path edges and is denoted by $l(p) = \sum_{i=1}^{k-1} l(u_i, u_{i+1})$. The transmission time to send σ units of data from v_1 to v_k using path p is $T(\sigma, p) = l(p) + \frac{\sigma}{c(p)}$. Given a source node $s \in V$, a destination node $t \in V$ and an amount σ of data to be transmitted, our problem is to find a path of minimum transmission time to transmit the σ units of data from s to t. This problem is known as the *quickest path* problem [2], or, to be more precise, as the single-pair quickest path problem. Similarly, the all-pairs quickest paths problem asks for the quickest path between any two nodes of the network. The problem has obvious applications to fast data transmissions in communication networks. (For other applications see [2].)

The quickest path problem is quite different from the traditional shortest path one. First, the latter problem is defined on a network where each edge (v, u) owns only a single attribute, namely, the distance from v to u. However, such a kind of network is not applicable in many practical situations, e.g. in a communication network where the transmission time between two nodes depends not only on the distance but also on the capacity of the edges in the network. Moreover, in the quickest path problem the selection of the path depends also on the size of the data to be transmitted. In one extreme case, if the amount of data is huge, then the quickest path should be the path with largest capacity. If on the other hand, the amount of data is quite small, then the quickest path is the shortest path with respect to lead time. An additional singularity of the quickest path problem is that if $(s, ..., v, ..., t)$ is the quickest path from s to t, then its subpath $(s, ..., v)$ is not necessarily the quickest one from s to v, to transmit the same amount of data. For example, in Fig. 1, the quickest path to transmit $\sigma = 100$ units of data from a to d is (a, b, d) with transmission time $36 + \frac{100}{5} = 56$. However, subpath (a, b) is not a quickest path to transmit $\sigma = 100$ units of data from a to b, since path (a, c, b) has smaller transmission time. The fact that a subpath of a quickest path is not necessarily itself a quickest path suggests that a Dijkstra-like labeling algorithm could not be used to solve the quickest path problem and makes interesting the study of different approaches towards the design of efficient algorithms.

Previously known results are as follows. The single pair quickest path problem has been solved in time $O(rm + rn \log n)$ [2, 12], where r is the number of distinct edge capacities. For sparse networks (i.e., $m = O(n)$), this complexity becomes $O(rn \log n)$. The all–pairs quickest path problem has been solved in $O(\min(rnm + rn^2 \log n,\ mn^2))$ [11]. For sparse graphs, the later complexity becomes $O(\min(rn^2 \log n, n^3))$.

In this paper, we show that the above time complexities can be significantly improved. We first present (Section 2) an improved algorithm for the single–pair problem with worst–case time complexity $O(r^*m + r^*n \log n)$, where r^* never exceeds the number of distinct capacities greater than the capacity of the shortest with respect to (wrt) lead time path in N. A more precise definition

of r^* is given in Section 2. The benefit of the proposed algorithm is that it takes advantage of the distribution of the capacity values on the network edges, which existing algorithms ignore. Depending on the capacity of the shortest wrt lead time path on the original network and, moreover, on the capacity of the shortest wrt lead time paths in appropriately defined subnetworks, parameter r^* can be as low as 1, or a very small integer, even if $r = m$. Note that in the worst case $r^* = r$, but the proposed algorithm offers a substantial improvement in all cases that $r^* < r$.

In addition, we present (Section 3) improved algorithms for the single-pair quickest path problem on sparse and planar networks. The complexities of the algorithms are expressed in terms of a parameter $\tilde{\gamma}$ which provides a measure of the topological complexity of N and ranges from 1 up to $\Theta(n)$. If N is planar, we give an $O(n \log n + n \log^3 \tilde{\gamma} + r^* \tilde{\gamma})$-time algorithm (Section 3.2). For arbitrary non-planar sparse networks, we obtain an algorithm with time complexity varying from $O(n \log n)$ up to $O(r^* n \log n)$, depending on the particular value of $\tilde{\gamma}$. Also, the all–pairs quickest path problem for planar networks is solved in $O(rn^2)$ time, while for arbitrary sparse networks, it is solved in $O(r\tilde{\gamma}^2 \log \tilde{\gamma} + rn^2)$ time.

Note that our results are clear improvements over the best previous ones. Moreover, our bounds match the best previous bounds only when *both* parameters r^* and $\tilde{\gamma}$ reach their extreme values (of r and $\Theta(n)$, respectively).

All of the above-mentioned results, however, relate to the static version of the problem, i.e. the network, the lead times and the capacities on its edges, as well as the amount of data to be transmitted, do not change over time. We also consider here (Section 4) a dynamic environment, where edges can be deleted and their lead times and capacities can be modified, and also, the amount of data to be transmitted can be changed. More precisely, we investigate the following *dynamic quickest path problem:* Given a network N (as above), build a data structure (i.e. preprocess N) in order to be able to answer very fast on-line queries asking for a path from any node u to any other node z with minimum transmission time for sending an amount of data σ. Also the data structure should be efficiently updated and maintain quickest path information after a modification of N. According to the best of our knowledge, there are no previous algorithms for the above problem. In this paper, we present an efficient dynamic algorithm for the case of planar and sparse networks. In the case of planar (resp., sparse) networks, the algorithm needs $O(r(n + \tilde{\gamma} \log \tilde{\gamma}))$ (resp., $O(r(n + \tilde{\gamma}^2 \log \tilde{\gamma}))$) time for preprocessing and $O(r(\tilde{\gamma} + \log n) + L)$ (resp., $O(r \log n + L)$) time for finding the quickest path from any vertex u to any vertex z, where L is the number of edges of the quickest path. In the case of a modification in the lead time or in the capacity of an edge, $O(r(\log n + \log^3 \tilde{\gamma}))$ (resp., $O(r(\log n + \tilde{\gamma}^2 \log \tilde{\gamma}))$) time is required for updating the data structure.

2 Single–pair quickest paths in general networks

Let $N(V, E, c, l)$ be a network where $G = (V, E)$, $|V| = n$, $|E| = m$, is a directed graph, $c(e) > 0$ is the capacity of an edge $e \in E$ and $l(e) \geq 0$ is the lead time for an edge $e \in E$. Let $C_1 < C_2 < ... < C_r$ be the r distinct capacity values of the edges of network N, $r \leq m$. We define $N^w = (V, E^w, c, l)$ to be a subnetwork of $N(V, E, c, l)$ such that $E^w = \{e : e \in E \wedge c(e) \geq w\}$. In the sequel, we say *shortest lead time path* from u to v to refer to the shortest path from u to v with respect to the lead time. The following observation has been made in [12].

Fact 2.1 *If q is a quickest path, then q is a shortest lead time path in $N^{c(q)}$.*

Proof: Since q is a quickest path with capacity $c(q)$, all of its edges belong to the network $N^{c(q)}$. Suppose that there is another path $p \in N^{c(q)}$ that is shortest (with respect to lead time) than q i.e. $l(p) < l(q)$. Since q is a quickest path, $l(q) + \sigma/c(q) \leq l(p) + \sigma/c(p)$. This implies that $c(p) < c(q)$, a contradiction since we have assumed that $p \in N^{c(q)}$. □

The algorithm of Rosen et.al. [12] for computing the quickest path from s to t in N, computes the shortest lead time path p_i in each network N^{C_i}, $1 \leq i \leq r$, and outputs as the quickest path, the one that minimizes the quantity $l(p_i) + \sigma/c(p_i)$, $1 \leq i \leq r$. Their algorithm follows.

PROCEDURE RSX(N) [12]
1. **for** $i = 1, ..., r$ **do**
Find a shortest lead time path p_i from s to t in N^{C_i};
2. The quickest path is p_k, where index k minimizes $l(p_i) + \sigma/c(p_i)$, $1 \leq i \leq r$.
end RSX

Using the algorithm of [9] that computes a shortest path in time $O(m+n\log n)$, the overall time complexity of algorithm RSX is $O(rm+rn\log n)$. For sparse graphs, the corresponding complexity becomes $O(rn\log n)$.

Algorithm RSX can be viewed as seeding serially for the capacity of the quickest path. If a hypothetical oracle would give us the capacity w_o of the quickest path, then the actual path could be found just in time $O(m+n\log n)$ by applying the shortest path algorithm of [9] on N^{w_o}. Below, we show that the seed for the capacity of the quickest path does not have to be serial. Let s^w denote the shortest lead time path in N^w and q the quickest path in N.

Lemma 2.1 *If the capacity of the quickest path q is $c(q) > C_i$ for some $i < r$, then $c(q) \geq c(s^{C_{i+1}})$.*

Proof: Since $c(q) > C_i$ for some $i < r$, q is in fact a path in subnetwork $N^{C_{i+1}}$. Let $s^{C_{i+1}}$ be the shortest lead time path in $N^{C_{i+1}}$. Since q is the quickest path, $l(q) + \sigma/c(q) \leq l(s^{C_{i+1}}) + \frac{\sigma}{c(s^{C_{i+1}})}$. However, since $s^{C_{i+1}}$ is the shortest lead time path in $N^{C_{i+1}}$, $l(s^{C_{i+1}}) \leq l(q)$. Adding the two inequalities, we get $\sigma/c(q) \leq \sigma/c(s^{C_{i+1}}) \Leftrightarrow c(q) \geq c(s^{C_{i+1}})$. □

Proposition 2.1 *If for some i, $1 \leq i \leq r$, the capacity of the shortest lead time path s^{C_i} is $c(s^{C_i}) = C_r$, then the capacity of the quickest path q is either $c(q) < C_i$ or $c(q) = C_r$.*

Proof: Assume $c(q) \geq C_i$ and $c(q) \neq C_r$. Since $c(q) > C_{i-1}$, then by Lemma 2.1, $c(q) \geq c(s^{C_{i+1}}) \Rightarrow c(q) \geq C_r \Rightarrow c(q) = C_r$, a contradiction. □

We also observe that some subnetworks N^w may not be connected graphs. In the case that there is no path from s to t in network N^{C_i} for some $i < r$, then s and t will remain unconnected in any network N^{C_j}, $j \geq i$, since N^{C_j} is a subnetwork of N^{C_i}. That is,

Lemma 2.2 *If there is no path from s to t in N^{C_i} for some i, $1 \leq i \leq r$, then the capacity $c(q)$ of the quickest path q is $c(q) < C_i$ (if $i = 1$, no path exists).*

By convention, we assume that if there is no path from s to t in some subnetwork, then the shortest path algorithm returns a path of "infinite" length and "infinite" capacity. Initially, all we know for the capacity of the quickest path is that $c(q) \geq C_1$. According to lemma 2.1, each application of the shortest path algorithm on network N^{w_i}, where $w_1 = C_1$ and $w_i = c(s^{w_{i-1}}), i > 1$, can be regarded as a query that provides us successively with more information for the range of

$c(q)$ (namely, $c(q) \geq c(s^{w_i})$). If for the ith such successive query, it happens that $c(s^{w_i}) = C_r$ or $c(s^{w_i}) = \infty$, then by proposition 2.1 or lemma 2.2 respectively, no more than $r^* = i$ queries are required. If, on the other hand, $\forall i, 1 \leq i < r$, $c(s^{w_i}) = w_i \Leftrightarrow c(s^{C_i}) = C_i$, then (and only then) we end up making $r^* = r$ queries. Given that the latter scenario is in every practical respect a very rare case, the improvement that the algorithm yields is important. The algorithm is given below.

PROCEDURE General_Single_Pair_Quickest_Path(N)
1. $w = C_1$;
2. $r^* = 0$;
3. **while** $w < C_r$ **do**
4. $\quad r^* = r^* + 1$;
5. $\quad$ Find a shortest lead time path p_{r^*} in N^w;
6. $\quad$ **if** $\exists i < r$ such that $c(p_{r^*}) = C_i$ **then** $w = C_{i+1}$ **else** $w = \infty$;
7. The quickest path is p_k, where index k minimizes $l(p_i) + \sigma/c(p_i)$, $1 \leq i \leq r^*$.
end General_Single_Pair_Quickest_Path

Lemma 2.3 *Algorithm General_Single_Pair_Quickest_Path correctly finds the quickest path between two given nodes of a general network in $O(r^*m + r^*n \log n)$ time.*

Proof: The **while** loop in Step 3 terminates whenever $w = C_r$ or $w = \infty$. This is justified by proposition 2.1 and lemma 2.2. The candidate set of shortest paths from which the quickest path is chosen in Step 7, is justified by Lemma 2.1. Since there are r^* applications of the shortest path algorithm in Step 5, we obtain, by using the algorithm of [9], an overall time complexity of $O(r^*m + r^*n \log n)$. □

An example where r^* is significantly smaller than r is given in Fig. 1. Algorithm General_Single_Pair_Quickest_Path is applied on a network N to transmit $\sigma = 100$ units of data from a to h. The algorithm is applied successively on subnetworks $N^{(4)} = N$ (finding shortest lead time path (a, e, h) with capacity 10), $N^{(15)}$ (finding shortest lead time path (a, f, h) with capacity 20) and $N^{(25)}$ (finding no path). It thus terminates in $r^* = 3$ iterations, yielding (a, f, h) as the quickest path. In contrast, algorithm RSX would require $r = |E| = 13$ iterations to find the quickest path. We also observe that Algorithm General_Single_Pair_Quickest_Path can be further enhanced in practice by making sure that among the potentially many paths in N^w with the same shortest lead time, the shortest such path with the largest minimum capacity is always chosen. This can be easily done in the same time complexity by modifying slightly the shortest path algorithm used. That is, the comparison for each derived subpath is now based on a vector $(l', -c')$, where l' is the subpath's lead time, c' is the minimum capacity of the subpath so far, and all comparisons are performed in lexicographic order.

3 Computing quickest paths in sparse networks

In the previous section, we saw that there is a kind of "information redundancy" in the approach of [12] in that not all queries that are asked may in fact be necessary. However, another potential source of redundancy may be found in the repeated applications of the shortest path algorithm on network versions that, loosely speaking, do not differ much. That is, subnetworks N^{C_i} and $N^{C_{i+1}}$ differ only in that $N^{C_{i+1}}$ has some fewer edges than N^{C_i} and therefore, the information obtained by

computing the shortest lead time path in $N^{C_{i+1}}$ may be useful in the computation of the shortest lead time path in N^{C_i}. This suggests the use of a dynamic algorithm for computing shortest paths that allows updates/deletions of edges.

Below, we show that dynamic shortest path algorithms can be used advantageously in the quickest path context. We give algorithms for both the single pair and the all pairs quickest paths problems on planar and on sparse networks, that compare favorably with the existing ones. Before proceeding to the description of the algorithms, we give some preliminaries.

3.1 Preliminaries

A closed surface is *orientable* if it can be constructed by attaching handles to a sphere. The *genus* $g(G)$ of a graph G is the minimum number of handles of an orientable surface in which G can be embedded.

A *hammock decomposition* is a decomposition of an n-vertex graph G into certain outerplanar digraphs called *hammocks* [8]. Hammocks satisfy certain separator conditions and the decomposition has the following properties: (i) each hammock has at most *four* vertices in common with any other hammock (and therefore with the rest of the graph), called the *attachment vertices*; (ii) each edge of the graph belongs to exactly one hammock; and (iii) the number of hammocks produced is order of the minimum possible among all decompositions and is bounded by a function involving certain topological measures of G (e.g., genus). More concretely, the number $\tilde{\gamma}$ of hammocks can range from 1 up to $\Theta(m)$ (m is the number of edges of G), depending on the graph, and is proportional to $g(G) + q$, where G is embedded on an orientable surface with $g(G)$ handles so as to minimize the number q of faces that cover all vertices [8]. If G is sparse, then $\tilde{\gamma}$ ranges from 1 up to $\Theta(n)$. In the case that G is planar, $g(G) = 0$ and the number of hammocks is at most a constant factor times the minimum number of faces that cover all vertices of G among all the possible embeddings of G on the plane. Also, if G is outerplanar then $\tilde{\gamma} = 1$. As it has been proved in [8], the hammock decomposition can be obtained in time linear to the size of G and an embedding of G into some topological surface does not need to be provided by the input.

In the case of general digraphs, the best algorithm for the dynamic shortest paths problem is due to Even and Gazit [4]. Their algorithm supports insertions and/or deletions of edges. If the digraph is planar, Feuerstein and Spaccamela have proposed a much more efficient algorithm that supports edge cost modification and/or edge deletion [6]. Their algorithm makes an $O(n \log n)$ time and space preprocessing of the graph and then answers queries in $O(n)$ time. An interesting feature of their algorithm is that $O(n^2)$ queries can be answered in $O(n^2)$ time. Their data structures can be updated in $O(\log^3 n)$ time. Also, their algorithm has the same complexity in the case of digraphs that have an $O(n^t)$-*separator decomposition*, where $0 < t < 1$ [5]. (A separator of a digraph $G = (V, E)$ is a subset of vertices $S \subset V$ such that the subgraph induced by $V \backslash S$ is not connected and the number of vertices in each connected component is at most a fixed fraction of the number of vertices in G. An $f(n)$-separator decomposition is a recursive decomposition of G using separators, where subgraphs of size n have separators of size $O(f(n))$.)

A dynamic shortest path algorithm that works efficiently for sparse digraphs with $\tilde{\gamma} = o(n)$, and supports edge cost modification and/or edge deletion is recently presented in [3]. If the input digraph is planar then the algorithm needs $O(n+\tilde{\gamma} \log \tilde{\gamma})$ preprocessing time and space; $O(\tilde{\gamma}+\log n)$ single–pair distance query time; $O(\tilde{\gamma} + \log n + L)$ single–pair shortest path query time, where L is the length of the shortest path; $O(\log n + \log^3 \tilde{\gamma})$ update time after an edge cost modification or edge deletion. If the input digraph is not planar, but has an $O(n^t)$-separator decomposition

$0 < t < 1$, then the bounds are the same with those for the planar case. Otherwise, the algorithm needs $O(n + \tilde{\gamma}^2 \log \tilde{\gamma})$ preprocessing time and space; $O(\log n)$ single–pair distance query time; $O(\log n + L)$ single–pair shortest path query time, where L is the length of the shortest path; and, $O(\log n + \tilde{\gamma}^2 \log \tilde{\gamma})$ update time after an edge cost modification or edge deletion. Since the algorithm of [3] is mostly used here, for the sake of completeness we briefly describe it in appendix A.

3.2 Single-pair quickest paths

We denote by L_i the length of the shortest lead time path from the source s to the destination t in subnetwork N^{C_i}. Let also E_i be the set of edges with capacity C_i, $1 \leq i \leq r$. The algorithm for finding the quickest path between two nodes s and t in a planar network is as follows:

PROCEDURE Planar_Single_Pair_Quickest_Path(N)
1. Use the preprocessing algorithm of [3] for planar digraphs, to build the appropriate data structures.
2. **for** $i = 1$ to r **do**
3. Use the query algorithm of [3] for planar digraphs, to find the length L_i of the shortest lead time path from s to t in N^{C_i}.
4. **for** each edge e in E_i **do**
5. Use the update algorithm for planar digraphs of [3] to delete e from N^{C_i}.
6. Find index k that minimizes $L_i + \sigma/C_i, 1 \leq i \leq r$.
7. Obtain the quickest path by applying a single source shortest path algorithm on N^{C_k}.

end Planar_Single_Pair_Quickest_Path

Lemma 3.1 *Algorithm Planar_Single_Pair_Quickest_Path correctly finds the quickest path between two given nodes of a planar network in $O(n \cdot \log n + n \cdot \log^3 \tilde{\gamma} + r^* \cdot \tilde{\gamma})$ time.*

Proof : First notice that the r iterations in the loop of Step 2 can be reduced to r^* as it was discussed in the previous section. From fact 2.1, the quickest path q is a shortest lead time path in $N^{c(q)}$. It is enough to compute the length L_i of the shortest lead time path in N^{C_i}, $1 \leq i \leq r$, and then compute the minimum of $L_i + \sigma/C_i, 1 \leq i \leq r$. Since network $N^{C_{i+1}} = (V, E^{C_{i+1}})$ differs from $N^{C_i} = (V, E^{C_i})$ in that $E^{C_{i+1}} = E^{C_i} - E_i$, and since we have already computed L_i in N^{C_i}, it is clear that L_{i+1} can be computed by deleting the edges in E_i from N^{C_i} and calling the query algorithm of [3] on the resulting network $N^{C_{i+1}}$. The overall time complexity is $O(n + \tilde{\gamma} \cdot \log \tilde{\gamma})$ for the preprocessing, plus $O(m \cdot (\log n + \log^3 \tilde{\gamma}))$ for the m updates, plus $O(r^* \cdot (\tilde{\gamma} + \log n))$ for the r^* single-pair distance queries [3]. Since in planar graphs $m = O(n)$, we have a total time of $O(n \cdot \log n + n \cdot \log^3 \tilde{\gamma} + r^* \cdot \tilde{\gamma})$. □

Note that the above algorithm applies also to sparse networks that have an $O(n^t)$-separator decomposition, $0 < t < 1$. But not all sparse networks may have this property. Below we give an algorithm for this latter case. The algorithm is based on the decomposition of the original network into a number of hammocks. The basic idea of the algorithm is the following. Consider the case that nodes s and t belong to two different hammocks H_s and H_t respectively. (The other case is similar.) Suppose that we have already computed L_i and we have to compute L_{i+1}. For each edge $e \in E_i$, we delete e from the current network and use the outerplanar update algorithm of [3] to update the hammock where e belongs to. We then, compute the distance from s to each attachment vertex of H_s, and also, the distance from each attachment vertex of H_t to t, using the outerplanar query

algorithm of [3]. Then, each hammock H is substituted by a constant sized subgraph that keeps the shortest path information among the attachment vertices of H. This constant size subgraph is called *sparse representative* of H and can be constructed in time linear in the size of the graph [3]. In the resulting network, we compute four (at most) shortest lead time path trees rooted at the four attachment vertices of H_s. We use this information to compute the quickest path from s to t. The details of the algorithm follow. (In the sequel, if v is not an attachement vertex, we will denote by H_v the hammock in which v belongs to. We also denote by $d_G(u, v)$ the length of the shortest lead time path from u to v subject to the constraint that the path lies totally within subgraph G.)

PROCEDURE Sparse_Single_Pair_Quickest_Path(N)
1. Find a decomposition of the network N into hammocks.
2. Apply the preprocessing algorithm of [3] on each one of the hammocks.
3. **for** $i = 1$ to r **do**
4. **if** s is not an attachement node **then**
Find the length $d_{H_s}(s, a_k^s)$ of the shortest lead time path in H_s from s to the attachment node a_k^s $(k = 1, \cdots, 4)$ of H_s using the algorithm of [3].
5. **if** t is not an attachement node **then**
Find the length $d_{H_t}(a_k^t, t)$ of the shortest lead time path in H_t from the attachment node a_k $(k = 1, \cdots, 4)$ of H_t to t.
6. For each hammock H, find its sparse representative $SR(H)$ using the algorithm of [3] and substitute H by $SR(H)$.
Let $N_{\tilde{\gamma}}$ be the resulting network and $V_{\tilde{\gamma}}$ be the set of nodes of $N_{\tilde{\gamma}}$.
7. Find shortest lead time paths trees in $N_{\tilde{\gamma}}$ rooted at the four attachment nodes a_k $(k = 1, \cdots, 4)$, of H_s using the algorithm of [9].
8. **if** s and t do not belong to the same hammock (i.e., $H_s \neq H_t$) **then**
Compute the length L_i of the shortest lead time path from s to t as the minimum of $d_{k,l}(s, t) = d_{H_s}(s, a_k^s) + d_{N_{\tilde{\gamma}}}(a_k^s, a_l^t) + d_{H_t}(a_l^t, t)$ for all possible combinations of a_k^s, a_l^t where a_k^s (a_l^t) is an attachment node of H_s (H_t).
else (s and t belong to the same hammock say H_s, i.e., $H_s = H_t$)
Compute the length L_i of the shortest lead time path from s to t as the minimum of $d_{k,l}(s, t)$, for all possible combinations of a_k^s, a_l^s, and $d_{H_s}(s, t)$.
9. **for** each edge e in E_i **do**
10. Delete e from the current network and apply the update algorithm of [3] on the hammock H^e containing e.
11. Find index k that minimizes $L_i + \sigma/C_i, 1 \leq i \leq r$.
12. Obtain the quickest path by applying a single source shortest path algorithm on N^{C_k}.
end Sparse_Single_Pair_Quickest_Path

Lemma 3.2 *Algorithm Sparse_Single_Pair_Quickest_Path correctly computes the quickest path between a given pair of nodes in a sparse network in* $O(n \cdot \log n + r^* \cdot (n + \tilde{\gamma} \cdot \log \tilde{\gamma}))$ *time.*

Proof : First notice again that the r iterations in the loop of line 3 can be reduced to r^* as discussed in the previous section. Since each edge of the network belongs to exactly one hammock, once an edge e is deleted from the current network, exactly one hammock H^e needs to be modified and the shortest lead time path information among its attachment vertices to be updated. The

updated hammock H^e is then substituted by its sparse representative that keeps the new shortest lead time path information among its four attachment vertices, and in this way, the network $N_{\tilde{\gamma}}$ is updated. At the query time, i.e., when the distance from s to t is computed in the current network, the single source algorithm of [9] finds the updated shortest lead time paths from the attachment vertices of H_s to the attachment vertices of H_t. Steps 1,2,4,5,6 take time $O(n)$. Step 7 takes time $O(\tilde{\gamma} \cdot \log \tilde{\gamma})$, since network $N_{\tilde{\gamma}}$ has $O(\tilde{\gamma})$ nodes and is sparse. The overall time of the main loop (step 3), excluding the time for the loop in step 9, is thus $O(r^* \cdot (n + \tilde{\gamma} \cdot \log \tilde{\gamma}))$. The overall number of times that step 10 is executed is $m = O(n)$. The updating procedure after an edge deletion is applied only in one hammock, and this takes time $O(\log n)$ [3]. So the total time complexity is $O(n \log n + r^*(n + \tilde{\gamma} \log \tilde{\gamma}))$. □

3.3 All–pairs quickest paths for sparse networks

The algorithm for computing all pairs quickest paths in a planar or a sparse network is the following.

PROCEDURE Sparse/Planar_All_Pairs_Quickest_Paths(N)
1. Find a decomposition of the planar network N into hammocks.
2. Apply the preprocessing routine of [3] on each one of the hammocks.
3. **for** $i = 1$ to r **do**
4. For each hammock H and for each node $v \in H$, find a shortest lead time path tree rooted at v.
5. For each hammock H, find its sparse representative $SR(H)$ using the algorithm of [3] and substitute H by $SR(H)$. Let $N_{\tilde{\gamma}}$ be the resulting network. of $N_{\tilde{\gamma}}$.
6. Compute all-pairs shortest lead time paths in $N_{\tilde{\gamma}}$ as follows. **if** $N_{\tilde{\gamma}}$ is planar or has a separator decomposition **then** apply the algorithm of [6] **else** apply the algorithm of [9].
7. **for** each pair of nodes u, v **do**
8. **if** u and v belong to the same hammock H **then** Compute $d_H(u, v)$ and $d_{k,l}(u, v) = \min_{k,l}\{d_H(u, a_k) + d_{N_{\tilde{\gamma}}}(a_k, a_l) + d_H(a_l, v)\}$. Compute the length $L_i^{u,v}$ of the shortest lead time path from u to v as the minimum of $d_H(u, v)$ and $d_{k,l}(u, v)$. **else** $L_i^{u,v} = \min_{k,l}\{d_{H_u}(u, a_k) + d_{N_{\tilde{\gamma}}}(a_k, a_l) + d_{H_l}(a_l, v)\}$, where H_u (H_v) is the hammock containing u (v).
9. Compute the quantity $L_i^{u,v} + \sigma/C_i$.
10. **for** each edge e in E_i **do**
11. Delete e from the current network and update the hammock containing e, as well as $N_{\tilde{\gamma}}$ using the algorithm of [3].

end Sparse/Planar_All_Pairs_Quickest_Paths

Lemma 3.3 *All pairs quickest paths in a sparse network that has an $O(n^t)$-separator decomposition, $0 < t < 1$ (e.g. a planar one), can be computed in $O(rn^2)$ time. Otherwise, all pairs quickest paths can be computed in $O(r\tilde{\gamma}^2 \log \tilde{\gamma} + rn^2)$ time.*

Proof : Correctness is clear from discussions in subsection 3.2. For the resource bounds note that each iteration of the for-loop (step 3) needs $O(n^2)$ time if N is planar or has an $O(n^t)$-separator

decomposition ($0 < t < 1$). The dominating step is step 7, since step 6 needs $O(\tilde{\gamma}^2)$ time by [6]. Otherwise, step 6 needs $O(\tilde{\gamma}^2) \log \tilde{\gamma})$ time by [9]. Also, note that each iteration of steps 10,11 needs $O(\log n + \log^3 \tilde{\gamma})$ (if N is planar or has a separator decomposition), or $O(\log n + \tilde{\gamma}^2 \log \tilde{\gamma})$ time by [3], and therefore does not dominate the running time of the algorithm. The bounds follow. □

4 Dynamic quickest paths

In this section we present a solution to the dynamic quickest path problem for the case of planar and sparse networks. As it is mentioned in the introduction, the amount of data to be transmitted is an important factor to determine the quickest path from a node u to a node z. While in the static statement of the problem the amount of data to be transmitted is fixed, in the dynamic one we allow these data to be changed. Thus, the preprocessing procedure of our algorithms makes a preprocessing of the network and creates the appropriate data structures in order to answer as fast as possible on-line queries asking for a path with minimum transmission time to send a given amount of data from any node u to any other node z of the network. Update procedures are also given which update the data structures in the case that the lead time and/or the capacity of one or more edges is modified.

Before describing the dynamic algorithm, we define a variation of the input network as follows. Let $N(V, E, c, l)$ be the input network and $C_1 < C_2 < ... < C_r$ be the r distinct capacity values of the edges of N. Let $N_\infty^w = (V, E, c, l^w)$ be a variation of N such that for each $e \in E$, $l^w(e) = l(e)$ if $c(e) \geq w$ and $l^w(e) = \infty$ otherwise. We shall first describe the case that N is planar. Our dynamic algorithm consists of three procedures, namely preprocessing, query and update ones.

Preprocessing procedure: Call the preprocessing algorithm of [3] in each one of the networks $N_\infty^{C_i}$ for $i = 1, \cdots, r$ and create the appropriate data structures.

Query procedure: Call the distance query algorithm of [3] in each one of the networks $N_\infty^{C_i}$, $i = 1, \cdots, r$, and compute the length $L_i^{u,z}$ of the shortest lead time path from u to z in the network $N_\infty^{c_i}$, $i = 1, \cdots, r$. In each network $N_\infty^{C_i}$, $i = 1, \cdots, r$, the quantity $L_i^{u,z} + \sigma/C_i$, is computed. If k is the index that minimizes $L_i^{u,z} + \sigma/C_i$, $i = 1, \cdots, r$, then, using the shortest path query algorithm of [3], the shortest lead time path from u to z is computed in the network $N_\infty^{C_k}$.

Update procedure: There are two update procedures. The lead time update and the capacity update procedure. The first one updates the data structure in the case of a modification to the lead time of an edge, while the second one in the case of a modification to the capacity of an edge. Note that deletion of an edge e corresponds to updating the lead time of the edge with an ∞ lead time.

Lead time update: Let e be the edge whose lead time l_1 is to be modified and let $c(e)$ be its capacity. Let also l_2 be the new lead time of e. Then, the lead time update algorithm uses the update algorithm of [3] to change the lead time of e in each network N_∞^w with $w \leq c(e)$, from l_1 to l_2.

Capacity update: Let e be the edge whose capacity C_i is to be modified and let $l(e)$ be its lead time. Suppose also that C_j is the new capacity of e. Then, the capacity update algorithm proceeds as follows: If $C_i < C_j$, it uses the update algorithm of [3] to change the lead time of e in each network N_∞^w with $C_i < w \leq C_j$, from ∞ to $l(e)$. Otherwise ($C_j < C_i$), it uses the update algorithm of [3] to change the lead time of e in each network N_∞^w with $C_j \leq w < C_i$, from $l(e)$ to ∞.

The following lemma discusses the correctness of the algorithm and gives the bounds in the case that the network is planar.

Lemma 4.1 *Given an n-node planar network N, there exists an algorithm for the dynamic quickest path problem on N that supports edge lead time modification, edge capacity modification and edge deletion, with the following characteristics: (i) preprocessing time $O(r(n+\tilde{\gamma}\log\tilde{\gamma})$; (ii) single–pair quickest path query time $O(r(\tilde{\gamma}+\log n)+L)$, where L is the number of edges of the quickest path; and (iii) update time $O(r(\log n+\log^3\tilde{\gamma}))$, after any modification and/or edge deletion.*

Proof : From the definition of the network N^w_∞, each path in N^w_∞ that has lead time not equal to ∞, has capacity greater than or equal to w. It is not difficult to see that if q is a quickest path in N, then q is a shortest lead time path in $N^{c(q)}_\infty$. Thus, the query procedure correctly computes the quickest path from a node u to a node z. Suppose now that the lead time of an edge e is changed. Then, we have to update all the networks where e belongs to, namely, all the networks N^w_∞ with $w \leq c(e)$. If the capacity of an edge e is changed, then we have to change the lead time of e in each network N^w_∞ with w between the old and the new capacity of e. The time complexity of the algorithm comes from the complexity of the algorithm of [3], in the case of planar digraphs. □

As previously discussed, the above bounds hold also if the sparse network is not planar but has a separator decomposition. Otherwise, we have the following.

Lemma 4.2 *Given an n-node sparse network N, there exists an algorithm for the dynamic quickest path problem on N that supports edge lead time modification, edge capacity modification and edge deletion, with the following characteristics: (i) preprocessing time $O(r(n+\tilde{\gamma}^2\log\tilde{\gamma})$; (ii) single–pair quickest path query time $O(r\log n+L)$, where L is the number of edges of the quickest path; and (iii) update time $O(r(\log n+\tilde{\gamma}^2\log\tilde{\gamma}))$, after any modification and/or edge deletion.*

Proof: Follows by the discussion in the proof of the previous lemma and the algorithm of [3] for this type of networks. □

Consider now the following *modified* version of the dynamic quickest path problem: The source and destination nodes of the network remain unchanged, and the amount of data to be transmitted from the source to the destination change more frequently than the network itself changes. Then, we can modify our dynamic algorithm in such a way that the computation of the shortest lead time distance is incorporated in the preprocessing and the update procedures instead of the query one. This improves considerably the query time without increasing the preprocessing time and by adding a factor of $r\tilde{\gamma}$ to the update time. More precisely we have the following lemma.

Lemma 4.3 *Given an n-node planar (resp., sparse) network N with a source s and a destination t, there exists an algorithm for the modified dynamic quickest path problem on N that supports edge lead time modification, edge capacity modification and edge deletion, with the following characteristics: (i) preprocessing time $O(r(n+\tilde{\gamma}\log\tilde{\gamma})$ (resp., $O(r(n+\tilde{\gamma}^2\log\tilde{\gamma}))$; (ii) $s-t$ quickest path query time $O(r+L)$, where L is the number of edges of the quickest path; and (iii) update time $O(r(\tilde{\gamma}+\log n+\log^3\tilde{\gamma}))$ (resp., $O(r(\log n+\tilde{\gamma}^2\log\tilde{\gamma}))$, after any modification and/or edge deletion.*

Proof: Correctness is easy. The resource bounds are as follows. Preprocessing takes the time stated in the lemma, because we run the preprocessing algorithm of [3] in the r subnetworks N^{C_i}. In the case of edge lead time or capacity modification and/or edge deletion, we run the update procedure of [3] in each subnetwork. We also find the quickest $s-t$ path in each N^{C_i}. This is done in order to achieve faster query time for our problem (as the data change more frequently than the network itself). Hence, the update bounds are clear. Answering an $s-t$ quickest path query takes

now $O(r+L)$ time, since all we have to do is to find the minimum among r transmission times and to output a path of L edges. □

Acknowledgements. We are grateful to Esteban Feuerstein for many helpful discussions.

References

[1] J. A. Bondy, U. S. R. Murty, *Graph Theory with Applications*, North Holland, New York, 1976.

[2] Y. L. Chen and Y. H. Chin, "The quickest path problem", *Computers and Operations Research*, vol. 17, pp. 153–161, 1990.

[3] H. N. Djidjev, G. E. Pantziou and C. D. Zaroliagis, "On-line and Dynamic Shortest Paths through Graph Decompositions," Dartmouth Technical Report PCS-TR93-200, November 1993. Submitted.

[4] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Operations Research*, Vol. 49, pp. 371–387, 1985.

[5] E. Feuerstein, Personal communication, January 1994.

[6] E. Feuerstein and A. M. Spaccamela, "Dynamic Algorithms for Shortest Paths in Planar Graphs," *Theor. Computer Science*, 116 (1993), pp.359-371.

[7] G. N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths," *J. ACM*, Vol.38, No. 1, pp.162–204, 1991;

[8] G. N. Frederickson, "Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems", *Proc. 30th Annual IEEE Symp. on FOCS*, 1989, pp.448-453; also CSD–TR-897, Purdue University, August 1989.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network oprimization algorithms," *J. ACM*, vol. 34, pp. 596–615, 1987.

[10] M.R. Garey, and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, New York, NY, 1979.

[11] Y.-C. Hung and G.-H. Chen, "On the quickest path problem," *Proc. ICCI'91*, LNCS 497, Springer-Verlag, pp. 44–46, 1991.

[12] J. B. Rosen, S.-Z. Sun and G.-L. Xue, "Algorithms for the quickest path problem and the enumeration of quickest paths," *Computers and Operations Research*, vol. 18, pp. 579–584, 1991.

[13] D. D. Sleator and R. E. Tarjan, "A Data Structure for Dynamic Trees", *JCSC*, Vol.26, pp.362-391, 1983.

APPENDIX A

In this appendix we give a brief description of the dynamic shortest path algorithm in [3]. We shall first give the algorithm in the case of planar digraphs and then discuss the extension to any non-planar sparse digraph. The algorithmic scheme for solving the dynamic shortest path problem on a planar digraph G is based on the solution of the same problem when the input digraph is outerplanar and on the hammock decomposition technique. Suppose that we have an n-vertex outerplanar digraph G_o with real edge costs (but no negative cycles). A *compressed version* $C(G_o)$ of G_o, with respect to a constant number of distinguished vertices a_i, is an outerplanar digraph of $O(1)$ size and can be generated in $O(n)$ time [7]. Furthermore, for every pair a_i, a_i, $i \neq j$, in G_o if a shortest path from a_i to a_j exists in G_o, then there is a corresponding (compressed) one in $C(G_o)$ of equal cost. Let S be a separator of G_o (i.e. a set of vertices whose removal disconnects G_o). Let also $SR(G_o)$ be the graph obtained as follows: remove S from G_o, substitute each subgraph produced by its compressed version and then join again the subgraphs. The graph $SR(G_o)$ is called the *sparse representative* of G_o. Clearly, $SR(G_o)$ has size $O(|S|)$. The preprocessing algorithm for G_o is based on a recursive separator decomposition of G_o. Note that in G_o we can always find a separation pair (i.e. $|S| = 2$) that splits G_o into two subgraphs G_1 and G_2 such that G_1 is at most (2/3) of the size of G_o. If we recursively apply this to G_1 and G_2, we have a separator decomposition of G_o. The interesting feature of this decomposition is that it can be represented as a binary tree $T(G_o)$ such that: the root node corresponds to G_o, its left and right child correspond to G_1 and G_2 respectively, and so on. Clearly the depth of $T(G_o)$ is $O(\log n)$. With each node v of $T(G_o)$ (except for the leaves) we also associate the sparse representative $SR(G)$ of the subgraph G corresponding to v. Each leaf of $T(G_o)$ is associated only with the original subgraph G, since in this case G is of $O(1)$ size.

The preprocessing algorithm for G_o proceeds in three steps. First, find shortest path information in G_o using the algorithm of [7]. (This takes $O(n)$ time.) Second, construct the separator tree $T(G_o)$. Third, compute sparse representatives for each node of $T(G_o)$ (except for the leaves). This step can be implemented in a recursive way starting from the root of $T(G_o)$. It is not hard to see that step 3 can be executed in $O(n)$ time. Step 2 can also be executed in $O(n)$ time using the dynamic trees of [13]. Therefore, the preprocessing algorithm runs in $O(n)$ time. The update algorithm proceeds as follows. Let e be the edge that its cost has been modified. Then e will belong to at most $O(\log n)$ subgraphs as they are determined by the preprocessing algorithm. Hence, it suffices to update (in a bottom-up fashion) those sparse representatives corresponding to the nodes of $T(G_o)$, that are in the tree path from the leaf node associated with the subgraph containing e until the root of $T(G_o)$. It is not hard to see that this algorithm takes $O(\log n)$ time. Now, as far as it concerns answering a single-pair query between any two vertices v and z, we proceed as follows. Using $T(G_o)$ find the first separation pair (p_1, p_2) that separates v from z in G_o. Note that this pair corresponds to a node x in $T(G_o)$. Then, $d(v, z) = \min\{d(v, p_1) + d(p_1, z), d(v, p_2) + d(p_2, z)\}$. The four distances can be computed by traversing the two tree paths from the leaves corresponding to the subgraphs containing v and z respectively, up to x. (The computation of the corresponding shortest paths is done in a similar way.) Therefore, a single pair shortest path or distance query can be answered in $O(\log n + L)$ time (where L is the number of edges in the path), or in $O(\log n)$ time respectively.

The preprocessing algorithm for a planar digraph G proceeds as follows. First, find a hammock decomposition of G into $\tilde{\gamma}$ hammocks. (This takes $O(n)$ time by [8].) Second, run the outerplanar preprocessing algorithm in each hammock separately. Third, replace each hammock H with its

sparse representative. This results into a new planar digraph $G_{\tilde{\gamma}}$ which is of size $O(\tilde{\gamma})$. Fourth, run the preprocessing algorithm of [6] on $G_{\tilde{\gamma}}$. This algorithm runs in $O(n + \tilde{\gamma} \log \tilde{\gamma})$ time. The update algorithm is straightforward. Let e be the edge that its cost has been modified. We have two data structures that should be updated. The first one concerns the hammock H where e belongs to. This is done by the outerplanar update algorithm. The second data structure is that of $G_{\tilde{\gamma}}$ and can be updated in $O(\log^3 \tilde{\gamma})$ time by [6]. Hence, we need in total $O(\log n + \log^3 \tilde{\gamma})$ for the updating of G. A single-pair query between any two vertices v and z can be answered as follows. If v and z do not belong to the same hammock, then their distance $d(v, z) = \min_{i,j}\{d(v, a_i) + d(a_i, a'_j) + d(a'_j, z)\}$ where a_i and a'_i respectively are the attachment vertices of the hammocks in which v and z belong to. If both v and z belong to the same hammock H, then note that the shortest path between them does not necessarily have to stay in H. Therefore, first compute (using the outerplanar query algorithm) their distance $d_H(v, z)$ inside H. After that, compute $d_{ij}(v, z) = \min_{i,j}\{d(v, a_i) + d(a_i, a_j) + d(a_j, z)\}$. Clearly, $d(v, z) = \min\{d_H(v, z), d_{ij}(v, z)\}$. Note that we need $O(\tilde{\gamma})$ time for querying in $G_{\tilde{\gamma}}$'s data structure [6] and $O(\log n)$ (or $O(\log n + L)$) time for querying in each hammock.

The above results can be extended to hold for any sparse digraph G as follows. If G is provided with an $O(n^t)$-separation decomposition $(0 < t < 1)$, then the bounds hold as they are for the planar case [5]. Otherwise, we use the dynamic algorithm scheme of [4] for $G_{\tilde{\gamma}}$ instead of the one in [6]. Note that by [8] the hammock decomposition technique applies to any digraph G and a hammock decomposition can be computed in time linear to the size of G. We summarize with the following.

Lemma 4.4 *Given an n-vertex planar (resp., sparse) digraph G with real-valued edge costs but no negative cycles, there exists an algorithm for the on-line and dynamic shortest path problem on G that supports edge cost modification and edge deletion with the following performance characteristics: (i) preprocessing time and space $O(n + \tilde{\gamma} \log \tilde{\gamma})$ (resp., $O(n + \tilde{\gamma}^2 \log \tilde{\gamma})$). (ii) single-pair distance query time $O(\tilde{\gamma} + \log n)$ (resp., $O(\log n)$); (iii) single-pair shortest path query time $O(\tilde{\gamma} + \log n + L)$ (resp., $O(L + \log n)$) (where L is the number of edges of the path); (v) update time (after an edge cost modification or edge deletion) $O(\log n + \log^3 \tilde{\gamma})$ (resp., $O(\log n + \tilde{\gamma}^2 \log \tilde{\gamma})$).*