Low-Port Tree Representations*

Shiri Chechik and David Peleg

Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science, Rehovot, 76100 Israel
{shiri.chechik,david.peleg}@weizmann.ac.il

Abstract. Consider an n-node undirected graph $G(V, E)$ with a pre-assigned port numbering for the outgoing edges of each node. The port numbers assigned to a node u of degree $\mathtt{deg}(u)$ are $\{0, 1, \ldots, \mathtt{deg}(u) - 1\}$. In certain contexts it is necessary to maintain a directed spanning tree of G, in which case each node needs to remember the port number leading to its parent. Hence the cost of a spanning tree T is the total number of bits the nodes need to store in order to remember T. This paper addresses the question of asymptotically bounding the cost of the optimal tree, as a function of the graph size. A tight upper bound of $O(n)$ is established on this cost, thus improving on the best previously known bound of $O(n \log \log n)$ [6] and proving the conjecture raised therein. This is achieved by presenting a polynomial time algorithm for constructing a spanning tree T of cost $O(n)$ for a given general graph G with an arbitrary port labeling.

1 Introduction

Many distributed applications make use of pre-defined network representations for guaranteeing efficient performance. These representations are often designed to be as compact as possible (cf. [16]) in terms of their memory requirements. Two notable examples are compact routing schemes [1,3,4,7,17,18,19,20], which commonly rely on sparse network representations such as partitions, covers and decompositions, and informative labeling schemes for a variety of applications, e.g., [5,6,8,9,10,11,12,14,15].

An important special case of a pre-defined network representation is that of a spanning tree. A number of well-known algorithms for basic distributed operations, such as broadcast, convergecast and graph exploration (cf. [2,13,16]), are based on maintaining a spanning tree for the network and using it for efficient communication.

The problem of compact port-based representations for spanning trees was introduced in [6], which focused on the problem of providing, during a pre-processing stage, a compact local encoding of a spanning tree for a given graph. In particular, it raised the question of the existence of encodings in which the average number of bits stored at each node is constant.

* Supported by a grant from the Israel Science Foundation.

C. Paul and M. Habib (Eds.): WG 2009, LNCS 5911, pp. 66–76, 2010.

More precisely, the following problem was considered in [6]. Consider an n-node undirected graph $G(V,E)$. Each node u has a pre-assigned port number for each of its outgoing edges, with the port number of the edge connecting it to the neighbor v denoted $\texttt{Port}(u,v)$. Denoting the degree of the node u by $\texttt{deg}(u)$, the port numbers assigned to u's ports are $\{0,\ldots,\texttt{deg}(u)-1\}$, or more formally, the port number $\texttt{Port}(u,v)$ is in $\{0,\ldots,\texttt{deg}(u)-1\}$, where $\texttt{Port}(u,v_1) \neq \texttt{Port}(u,v_2)$ for every two distinct neighbors v_1 and v_2 of u. The port numbers are not necessarily symmetric, i.e., it could be that $\texttt{Port}(u,v) \neq \texttt{Port}(v,u)$. The network nodes are required to maintain a directed spanning tree of G, with each node required to remember the port number leading to its parent. For a port number p, denote by $\omega(p)$ the number of bits required to encode p using the standard binary representation for integers. Formally,

$$\omega(p) = \begin{cases} 1, & \text{if } p = 0 \ , \\ \lfloor \log p \rfloor + 1, & \text{if } p \geq 1 \ . \end{cases}$$

The cost of a tree T is the total number of bits the nodes need to remember, denoted $\texttt{Cost}(T,G)$. Formally,

$$\texttt{Cost}(T,G) \ = \sum_{v \in V,\ v \neq r(T)} \omega(\texttt{Port}(v, parent(v,T))) \ ,$$

where $r(T)$ is the root of the tree T and $parent(v,T)$ is the parent of v in the tree T. Define $\texttt{Cost}(G)$ to be $\min\{\texttt{Cost}(T,G)\}$, where the minimum is taken over all spanning trees T of G. The question of constructing a spanning tree T minimizing $\texttt{Cost}(T,G)$ for a given graph G, hence also determining $\texttt{Cost}(G)$, was shown in [6] to enjoy a polynomial time algorithm.

In this paper, we are interested in bounding the asymptotic behavior of $\texttt{Cost}(G)$ as a function of the graph size. Define $\texttt{Cost}(n)$ to be $\max\{\texttt{Cost}(G)\}$, where the maximum is taken over all n-node graphs G. An upper bound of $O(n \log\log n)$ on $\texttt{Cost}(n)$ was shown in [6], by presenting a polynomial time algorithm constructing a spanning tree T of cost $O(n \log\log n)$ for a given n-node graph G. It was conjectured in [6] that the actual bound is $\texttt{Cost}(n) = \Theta(n)$. In fact, a tight upper bound of $O(n)$ is proved therein for the special cases of complete graphs with arbitrary labeling and of arbitrary graphs with symmetric port labeling. In what follows, we confirm the above conjecture for arbitrary graphs and arbitrary assignments of port numbers, by establishing a tight upper bound of $O(n)$ on $\texttt{Cost}(n)$. This is achieved by presenting a polynomial time algorithm for constructing a spanning tree T of cost $O(n)$ for a given general graph G with an arbitrary port labeling.

2 Construction of a Low Port Tree with Cost $O(n)$

In this section we present an algorithm for constructing a spanning tree with cost $O(n)$ for a given graph $G(V,E)$.

2.1 Escape-Paths

We start by defining some basic notions. For a subtree T of G, we say that the edge (u, w) is an *exit edge* of T if $u \in V(T)$ and $w \notin V(T)$, and the node u is an *exit node* of T if it has an exit edge of T. A tree T rooted at $r(T)$ is *well-oriented* if $r(T)$ is an exit node of T.

For a directed subtree T of G rooted at $r(T)$, a path $P = (v_1, \ldots, v_k)$ is said to be a *escape-path* of T if

1. $v_1 = r(T)$,
2. $v_1, \ldots, v_k \in V(T)$,
3. $(v_i, v_{i+1}) \in E$ for every $1 \leq i \leq k-1$,
4. v_k is an exit node (i.e., it has a neighbor $z \notin V(T)$).

Note that the edges of the escape-path P need not be in T.
Define the following order relation on paths in G. Given a path $P_1 = (v_1, \ldots, v_k)$ and a path $P_2 = (z_1, \ldots, z_r)$, we say that the path P_2 is *lighter* than P_1 if either $r < k$ or $r = k$ and $\texttt{Port}(z_{r-1}, z_r) \leq \texttt{Port}(v_{k-1}, v_k)$. Notice that this lightness relation is transitive. For a subtree T of G, a path $\mathcal{P}$ is said to be a *lightest* escape-path of T if it is an escape-path of T and no other escape-path of T is lighter than $\mathcal{P}$.

2.2 Outline of the Algorithm

The algorithm consists of two phases, a *preprocessing phase* and a *tree construction phase*. Our approach in the construction process of the spanning tree is based on starting with individual nodes and merging them gradually by taking a small well-oriented tree T and hanging it on another subtree $\hat{T}$ by adding an exit edge from $r(T)$ to $\hat{T}$. Whenever this process succeeds, it leads to a low cost spanning tree. The complications arise once the process encounters some small subtree T that is not well-oriented, i.e., whose root has no exit edge. In this case, the algorithm tries to re-orient the subtree T by finding a lightest escape-path of T and reversing the relevant edges in that path.

The construction process proceeds in iterations, where in each iteration i the algorithm chooses a subtree T_i and applies this process on it. For the sake of a clearer description of the algorithm and its analysis, let us keep a record of this merging process by remembering, for each iteration i, the basic small subtree T_i from which we started and the escape-path from the original root $r(T_i)$ to the selected exit node (which acts as the new root of T_i after the reversal). The preprocessing stage deals with selecting such escape-paths.

The preprocessing phase. Preprocessing consists of two stages. In the first, Stage S1, the algorithm chooses for each T_i a lightest escape-path $\mathcal{P}_i$ in a naive way. By the analysis given in [6], these paths yields a tree of cost $O(n \log \log n)$. In the second stage of the preprocessing phase, Stage S2, the algorithm builds cheaper paths that lead to the desired cost of $O(n)$.

Stage S1 of the preprocessing phase consists of $n-1$ iterations. The algorithm maintains a *forest* F, namely, a collection of (vertex disjoint) subtrees whose union spans V. Initially, the forest F contains n subtrees, each consisting of a single vertex. In each iteration i, the algorithm chooses the smallest subtree T_i in the collection and merges it with another subtree in the collection. At the end of this stage, the forest F consists of a single tree spanning the entire graph. In addition, the algorithm also keeps record of the trees $T_1, \ldots, T_{n-1}$ chosen during the $n-1$ iterations. Note that these trees are not necessarily disjoint; it is possible that $V(T_i) \subset V(T_j)$ for some $i < j$, although partial overlaps may not occur.

Let us now describe in detail the process of transforming and merging the subtree T_i, in iteration i of Stage S1. The process consists of two main steps. In the first step, the algorithm identifies a lightest escape-path $\mathcal{P}_i$ of T_i, ending at an exit node v_k. (If the root $r(T_i)$ itself is an exit node, then $\mathcal{P}_i$ consists of the single node $v_1 = r(T_i)$.) The algorithm then transforms T_i into a well-oriented tree T_i' on the same set of vertices. (If $r(T_i)$ is an exit node then no change is needed, i.e., $T_i' = T_i$ and $r(T_i') = r(T_i)$.) In the second step, the algorithm looks at the set of exit edges of $r(T_i')$, selects the exit edge $(r(T_i'), z)$ of minimum $\texttt{Port}(r(T_i'), z)$, sets $Out_i \leftarrow z$, and lets $\hat{T} \in F$ be the subtree containing z. The algorithm then merges the subtrees T_i' and $\hat{T}$ into a subtree $\tilde{T}$ by adding the edge $(r(T_i'), z)$, removes T_i and $\hat{T}$ from F and adds the merged tree $\tilde{T}$ instead. For convenience, the formal description of the above process is broken into two procedures (presented in Figure 1): Procedure Transform, performing the first step, and Procedure Merge, performing the second.

Procedure Transform$(\boldsymbol{G, T_i})$

1. $T_i' \leftarrow T_i$
2. Find a lightest escape-path of T_i, $\mathcal{P}_i = (v_1, \ldots, v_k)$.
3. For every $1 \leq r \leq k-1$, add the edge (v_r, v_{r+1}) of $\mathcal{P}_i$ to T_i'.
   In turn, for $2 \leq r \leq k$, remove from T_i' the (unique) outgoing edge of v_r in T_i', (v_r, w_r).
4. Return the transformed tree and the escape-path, $(T_i', \mathcal{P}_i)$.

Procedure Merge$(\boldsymbol{G, T_i, v_k, T_i', F})$

1. $Out_i \leftarrow$ node z outside T_i' with edge to v_k of minimal $\texttt{Port}(v_k, z)$.
2. Let $\hat{T} \in F$ be the tree in F that contains Out_i.
3. $\tilde{T} \leftarrow (V(T_i') \cup V(\hat{T}), E(T_i') \cup E(\hat{T}))$.
4. Connect v_k to Out_i in $\tilde{T}$.
5. Remove T_i and $\hat{T}$ from F and add the merged $\tilde{T}$ instead.
6. Return (Out_i, F).

Fig. 1. The procedure for merging the tree T_i with the tree $\hat{T}$

The transformation process, performed by Procedure Transform, operates as follows. Let $\mathcal{P}_i = (v_1, v_2, \ldots, v_k)$ be a lightest escape-path of T_i. Set $T_i' \leftarrow T_i$. For every $1 \leq j \leq k-1$, add the edge (v_j, v_{j+1}) of $\mathcal{P}_i$ to T_i'. In turn, for $2 \leq j \leq k$, remove from T_i' the (unique) outgoing edge of v_j in T_i', (v_j, w_j). Figure 2 illustrates this transformation process.

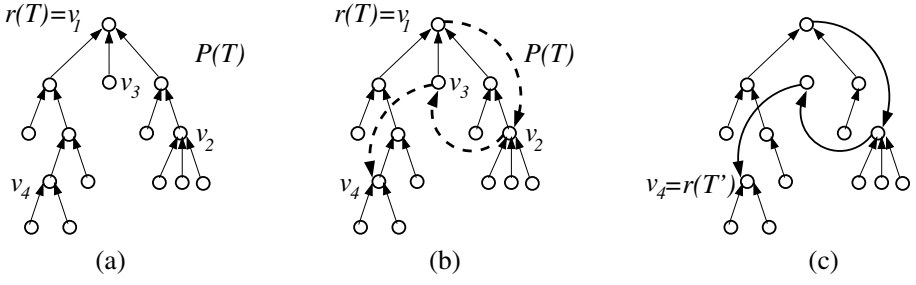

Fig. 2. (a) The tree T. (b) The lightest escape-path $\mathcal{P} = (v_1, v_2, v_3, v_4)$ of T (dashed). The node v_4 has an exit edge (not shown in the figure). (c) The tree T', with v_4 as its root. T' is obtained from T by erasing the edges that connected v_2, v_3 and v_4 to their parents in T, and replacing them by the edges of $\mathcal{P}$.

In the second stage of the preprocessing phase, Stage S2, the algorithm examines the chosen trees T_i and their escape-paths $\mathcal{P}_i$ in reverse order, from T_{n-1} to T_1, and tries to find *shortcut* paths $\mathcal{P}_i^*$ using the nodes in the previously selected shortcut paths $\mathcal{P}_j^*$ for $j > i$.

Loosely speaking, when considering a tree T_i, the algorithm treats all nodes in the tree T_i that participate in some shortcut path $\mathcal{P}_j^*$ for some $j > i$ as nodes outside the tree T_i. It then tries to use these new outer nodes in finding a shortcut path $\mathcal{P}_i^*$ of lower cost than the escape-path $\mathcal{P}_i$. The definition of the *shortcut* path is similar to that of the escape-path, with a small modification: instead of the requirement that the last node in the path has a neighbor outside of $V(T_i)$, we require that the last mode has an edge to a node that participates in $\mathcal{P}_j^*$ for some $j > i$. The *lightest shortcut path* is the analogue of the lightest escape-path, i.e., a path $\mathcal{P}^*$ is said to be a *lightest* shortcut-path of T if it is a shortcut-path of T and no other shortcut-path of T is lighter than $\mathcal{P}^*$. If the algorithm finds a shortcut-path lighter than the original escape-path $\mathcal{P}_i$, then it replaces the escape-path $\mathcal{P}_i$ with the lightest shortcut-path $\mathcal{P}_i^*$.

Hence iteration i of Stage S2 of the preprocessing phase proceeds as follows. Consider the subtree T_i for $1 \leq i \leq n-1$. The algorithm examines all shortcut-paths $(v_1, \ldots, v_k)$ of the tree T_i such that v_k has a neighbor z that belongs to some shortcut path $\mathcal{P}_j^*$ for $j > i$. If the algorithm finds a shortcut-path lighter than $\mathcal{P}_i$ then it sets $\mathcal{P}_i^*$ to be the lightest such path $(v_1, \ldots, v_k)$ and Out_i to the node z that belongs to some shortcut path $\mathcal{P}_j^*$ for $j > i$ with minimal $\texttt{Port}(v_k, z)$.

The preprocessing phase is described formally in Figure 3. After the preprocessing phase, the algorithm keeps a set of subtrees $T_1, \ldots, T_{n-1}$, a set of shortcut paths $\mathcal{P}_1^*, \ldots, \mathcal{P}_{n-1}^*$ and a set of out nodes $Out_1, \ldots, Out_{n-1}$.

Procedure Find_Paths(G)

1. $F \leftarrow \{(\{v\}, \emptyset) \mid v \in V\}$
2. For $i = 1, \ldots, n-1$ do: /* Stage S1 */
   (a) Let T_i be the smallest-size tree in F.
   (b) Invoke $(T'_i, \mathcal{P}_i) \leftarrow \mathsf{Transform}(G, T_i)$.
   (c) Invoke $(Out_i, F) \leftarrow \mathsf{Merge}(G, T_i, v_k, T'_i, F)$.
3. $S \leftarrow \emptyset$
4. For $i = n-1, \ldots, 1$ do: /* Stage S2 */
   (a) Consider $\mathcal{P}_i = (v_1, \ldots, v_k)$.
   (b) If there exists a shortcut-path $\mathcal{P}^*_i$ (using the nodes in S) lighter than $\mathcal{P}_i$ then do:
      i. Set $\mathcal{P}^*_i$ to be the lightest shortcut path $(z_1, \ldots, z_r)$ of T_i using the nodes in S.
      ii. Set Out_i to be the node z in S with an edge (z_r, z) of minimal $\mathtt{Port}(z_r, z)$.
   (c) Else, set $\mathcal{P}^*_i \leftarrow \mathcal{P}_i$.
   (d) $S \leftarrow S \cup V(\mathcal{P}^*_i)$.
5. Return $(T_1, \ldots, T_{n-1}, \mathcal{P}^*_1, \ldots, \mathcal{P}^*_{n-1}, Out_1, \ldots, Out_{n-1})$

Fig. 3. The procedure for finding the transformation paths for all subtrees

Procedure Tree_Cons($G, T_1, \ldots, T_{n-1}, \mathcal{P}^*_1, \ldots, \mathcal{P}^*_{n-1}, Out_1, \ldots, Out_{n-1}$)

1. $\mathcal{T} \leftarrow (V, \emptyset)$
2. For $i = 1, \ldots, n-1$ do:
   (a) Consider $\mathcal{P}^*_i = (v_1, \ldots, v_k)$.
   (b) For every $1 \le j \le k-1$, add the edge (v_j, v_{j+1}) to $\mathcal{T}$.
   In turn, for $2 \le j \le k$, remove from $\mathcal{T}$ the (unique) outgoing edge of v_j in $\mathcal{T}$, (v_j, w_j).
   (c) Add to $\mathcal{T}$ an edge from v_k to Out_i.
3. Return $\mathcal{T}$.

Fig. 4. The procedure for constructing a tree with total cost $O(n)$

Main(G)

1. Invoke $(T_1, \ldots, T_{n-1}, \mathcal{P}^*_1, \ldots, \mathcal{P}^*_{n-1}, Out_1, \ldots, Out_{n-1}) \leftarrow \mathsf{Find\_Paths}(G)$.
2. Invoke $\mathcal{T} \leftarrow \mathsf{Tree\_Cons}(G, T_1, \ldots, T_{n-1}, \mathcal{P}^*_1, \ldots, \mathcal{P}^*_{n-1}, Out_1, \ldots, Out_{n-1})$.

Fig. 5. Constructing a spanning tree with cost $O(n)$

The tree construction phase. The tree construction phase consists of $n-1$ iterations. The algorithm maintains a subgraph $\mathcal{T}$ which is first initialized to be $\mathcal{T} \leftarrow (V, \emptyset)$. In each iteration i, the algorithm does the following. Let $\mathcal{P}_i^* = (v_1, \ldots, v_k)$. For every $1 \leq j \leq k-1$, the algorithm adds the edge (v_j, v_{j+1}) to $\mathcal{T}$. In turn, for $2 \leq j \leq k$, it removes from $\mathcal{T}$ the (unique) outgoing edge of v_j in $\mathcal{T}$, (v_j, w_j). It then adds to $\mathcal{T}$ an edge from v_k to Out_i. In the end, it returns $\mathcal{T}$.

The tree construction phase is presented formally in Figure 4. The main algorithm is given in Figure 5.

3 Analysis

Let $\mathcal{T}$ be the final tree returned by the algorithm. Denote by $\mathcal{S}$ the set of all edges added to $\mathcal{T}$ at some iteration in procedure Tree_Cons. Notice that $E(\mathcal{T}) \subseteq \mathcal{S}$, where $E(\mathcal{T})$ is the set of edges in the final tree $\mathcal{T}$. It could be that $E(\mathcal{T}) \neq \mathcal{S}$, as some of the edges in $\mathcal{S}$ might be removed in later iterations. We show that the total cost of all edges in $\mathcal{S}$, denoted by $\texttt{Cost}(\mathcal{S})$, is $O(n)$, which implies that $\texttt{Cost}(\mathcal{T}, G) = O(n)$. For the analysis, we partition the edges that were added to $\mathcal{S}$ into two subsets, E_{out} and E_{esc}, and bound separately the total cost of edges in each subset by $O(n)$, thus the total cost of all edges in $\mathcal{S}$ is also $O(n)$.

For a subtree T_r with $\mathcal{P}_r^* = (v_1, \ldots, v_k)$, all edges (v_i, v_{i+1}) for $1 \leq i \leq k-1$ belong to the subset E_{esc} and are referred to as *escape-edges*. The edge from the node v_k to the node Out_r is called an *out-edge* and it belongs to the subset E_{out}.

Consider an edge (x, y), and let $c = \texttt{Port}(x, y)$. Notice that there are exactly c neighbors w of x that are "cheaper" than y, i.e., such that $\texttt{Port}(x, w) < \texttt{Port}(x, y)$. For the sake of the analysis, we employ the following *charging rule* on escape-edges. If the algorithm selects the escape-edge (x, y) to the constructed tree $\mathcal{T}$ at some iteration of Procedure Tree_Cons, then each such cheaper node w incurs a charge of 1 upon adding (x, y) to $\mathcal{T}$.

For each node w, we are interested in identifying the subtrees T_i that cause w to incur a charge. Formally, we say that T_i is a *charging subtree* of w if w incurs a charge upon adding the shortcut path $\mathcal{P}_i^* = (v_1, \ldots, v_k)$, namely, there exists some $1 \leq j \leq k-1$ such that the escape-edge (v_j, v_{j+1}) is more expensive than (v_j, w), or in other words, $\texttt{Port}(v_j, v_{j+1}) > \texttt{Port}(v_j, w)$. Denote by $M(w)$ the set of charging subtrees of w. In the analysis we prove that the algorithm guarantees that every node w has at most one charging subtree, namely, $|M(w)| \leq 1$.

For each node $w \in V$, denote by $C(w)$ the *charge count* of w, namely, the number of escape-edges for which w incurred a charge. Formally,

$$C(w) = \#\{v \mid \texttt{Port}(v, u) > \texttt{Port}(v, w) \text{ and } (v, u) \in E_{esc}\}.$$

We now show that the shortcut-paths chosen by the algorithm are disjoint.

Lemma 1. *The paths $\mathcal{P}_i^*$ for $1 \leq i \leq n-1$ are disjoint.*

Proof: The proof is straightforward from the construction of $\mathcal{P}_i^*$. Consider some T_i and T_j where $1 \leq j < i \leq n-1$. Assume, towards contradiction, that the

paths $\mathcal{P}_i^*$ and $\mathcal{P}_j^*$ intersect, i.e., there exists a node v such that $v \in \mathcal{P}_i^*$ and $v \in \mathcal{P}_j^*$. Letting $\mathcal{P}_j^* = (z_1, \ldots, z_r, v, \ldots)$, notice that the path $P = (z_1, \ldots, z_r)$ is shorter than $\mathcal{P}_j^*$. Moreover, after iteration i of step 4 of Procedure Find_Paths, the node v was added to S. Hence when looking for the lightest shortcut path for the tree T_j in iteration j of step 4 of Procedure Find_Paths, the algorithm should have chosen P as $\mathcal{P}_j^*$; contradiction. ∎

The following two lemmas establish that the resulting subgraph $\mathcal{T}$ is a tree.

Lemma 2. *After iteration i of step 2 of Procedure* Tree_Cons*, all cycles in $\mathcal{T}$ contain an edge (z, w) such that $z \in \mathcal{P}_r^*$ for some $r > i$.*

Proof: By induction on i. For $i = 1$, after the first iteration of Procedure Tree_Cons $\mathcal{T}$ contains only one edge and the claim is trivial. Assume the claim holds for every $j < i$, and consider iteration i of step 2 of Procedure Tree_Cons. By the inductive hypothesis, in the beginning of iteration i all cycles in $\mathcal{T}$ contain a node z such that $z \in \mathcal{P}_r^*$ for some $r \geq i$.

Consider $\mathcal{P}_i^* = (v_1, \ldots, v_k)$ and consider a cycle C that contains an edge (z, w) such that $z \in \mathcal{P}_i^*$. Note that during iteration i of step 2(b) of Procedure Tree_Cons, the outgoing edge (z, w) is removed, and therefore the subgraph $\mathcal{T}$ no longer contains the cycle C. In addition, for all edges (v_j, v_{j+1}) that are added to $\mathcal{T}$ in iteration i of step 2 of Procedure Tree_Cons, the outgoing edge of v_{j+1} is removed, so clearly no cycle is created. The only exception is for the edge from v_k to Out_i. If $Out_i \notin T_i$, then again no cycle is closed. If $Out_i \in T_i$ then it must be that $Out_i \in \mathcal{P}_r^*$ for some $r > i$. It follows that the claim also holds for iteration i. ∎

Corollary 1. *The final tree $\mathcal{T}$ does not contain cycles.*

Lemma 3. *The number of edges in $\mathcal{T}$ in the end of procedure* Tree_Cons *is $n-1$.*

Proof: In step 2(b) of Procedure Tree_Cons, the number of edges that are added to $\mathcal{T}$ is equal to the number of edges that are removed from $\mathcal{T}$. In step 2(c) of Procedure Tree_Cons, one edge is added to $\mathcal{T}$. Therefore, one edge is added to $\mathcal{T}$ in each iteration of procedure Tree_Cons. As the procedure has $n-1$ iterations overall, the number of edges in $\mathcal{T}$ in the end of procedure Tree_Cons is $n-1$. ∎

Corollary 1 and Lemma 3 directly yield the following.

Lemma 4. *The subgraph $\mathcal{T}$ is a tree.*

We now show that each node z has at most one charging subtree.

Lemma 5. *The charging trees of every $z \in V$ satisfy $|M(z)| \leq 1$.*

Proof: Consider a node $z \in V$. Assume, towards contradiction, that both $T_{r_1}, T_{r_2} \in M(z)$, and without loss of generality assume that $r_1 > r_2$. Let $\mathcal{P}_{r_1}^* = (v_1, \ldots, v_{k_1})$ and $\mathcal{P}_{r_2}^* = (z_1, \ldots, z_{k_2})$. As $T_{r_1}, T_{r_2} \in M(z)$, it must be that z has an edge to both v_i and z_j, and moreover, $\texttt{Port}(v_i, z) < \texttt{Port}(v_i, v_{i+1})$

and $\texttt{Port}(z_j, z) < \texttt{Port}(z_j, z_{j+1})$ for some $1 \leq i \leq k_1 - 1$ and $1 \leq j \leq k_2 - 1$. Notice that the path $P = (z_1, \ldots, z_j, z)$ is lighter than the path $\mathcal{P}^*_{r_2}$ and also z has an edge to $v_i \in \mathcal{P}^*_{r_1}$. Moreover, after iteration r_1 in step 4 of Procedure Find_Paths, the node v_i was added to S. When looking for a shortcut for the tree T_{r_2} in iteration r_2 in step 4 of Procedure Find_Paths, the algorithm was supposed to choose P (or some other path lighter than P) as $\mathcal{P}^*_{r_2}$; contradiction. ∎

We now turn to the cost analysis of the resulting tree $\mathcal{T}$.

Lemma 6. *The charging count of every $z \in V$ satisfies $0 \leq C(z) \leq 3$.*

Proof: Consider some node $z \in V$. By Lemma 5, $|M(z)| \leq 1$. This means that z incurs a charge only on one subtree. It remains to show that when z incurs a charge on some subtree T_i, that charge is at most 3. Assume $T_i \in M(z)$. Let $\mathcal{P}_i = (v_1, \ldots, v_k)$ be the lightest escape-path generated by procedure Transform in the transformation process of T_i. Since $\mathcal{P}_i$ is a shortest path from v_1 to v_k in $G(T_i)$, we have that z has at most three neighbors in $\mathcal{P}_i$, otherwise the procedure could have used z to get a shorter path between v_1 and v_k (this is due to the fact that if z is adjacent to nodes $v_{l_1}, \ldots, v_{l_t}$ on $\mathcal{P}_i$, then $(v_1, \ldots, v_{l_1}, z, v_{l_t}, \ldots, v_k)$ is an alternate path between v_1 and v_k, and if $t \geq 4$ then this alternate path is necessarily shorter than the original.). When updating $\mathcal{P}_i$ to a lighter path $\mathcal{P}^*_i = (z_1, \ldots, z_r)$, again there can be at most three nodes among $z_1, \ldots, z_r$ that have an edge to z. Thus $C(z) \leq 3$. ∎

For the analysis, we partition the overall cost of $\mathcal{S}$ into

$$\texttt{Cost}(\mathcal{S}) \;=\; C_{out} + C_{esc} \;,$$

where

$$C_{out} \;=\; \sum_{(x,y) \in E_{out}} \omega(\texttt{Port}(x, y))$$

and

$$C_{esc} \;=\; \sum_{(x,y) \in E_{esc}} \omega(\texttt{Port}(x, y)) \;.$$

Lemma 7. *Consider some subtree T_i for $1 \leq i \leq n-1$. Let $\mathcal{P}^*_i = (v_1, \ldots, v_k)$ and $z = Out_i$. Then $\texttt{Port}(v_k, z) < |T_i|$.*

Proof: The proof is straightforward from the definition of Out_i. By definition, Out_i is a node z outside T_i (or a node in T_i that participate in some shortcut path $\mathcal{P}^*_j$ for some $j > i$) with minimal $\texttt{Port}(v_k, z)$. So in the worst case, $\texttt{Port}(v_k, z) = |T_i| - 1$. ∎

Lemma 8. $C_{out} = O(n)$.

Proof: For each subtree T_i, the algorithm adds at most one edge to E_{out}, and by Lemma 7 the cost of this edge is at most $\lfloor \log |T_i| \rfloor + 1$. There are $n - 1$ such subtrees T_i. In each iteration in step 2 of Procedure Find_Paths, the algorithm

chooses the smallest subtree in the forest F and merges it with another subtree. As initially F contains n subtrees of size 1, there are at least $n/2$ iterations with subtrees of size 1, and at least $n/4$ subsequent iterations with subtrees of size at most 2, and so on. It follows that the total cost is

$$C_{out} \;\leq\; \sum_{i=1}^{\log n} \frac{n}{2^i} \cdot (i+1) \;=\; O(n)\ . \qquad \blacksquare$$

Lemma 9. $C_{esc} = O(n)$.

Proof: Notice that exactly $\texttt{Port}(x,y)$ nodes incur a charge for each edge $e = (x,y) \in E_{esc}$, and therefore

$$\sum_{(x,y)\in E_{esc}} \texttt{Port}(x,y) \;=\; \sum_{v\in V} C(v) \;\leq\; 3n\ ,$$

where the last inequality follows by Lemma 6. It follows that

$$\begin{aligned} C_{esc} &= \sum_{\substack{(x,y)\in E_{esc}\\ \texttt{Port}(x,y)=0}} 1 + \sum_{\substack{(x,y)\in E_{esc}\\ \texttt{Port}(x,y)>0}} (\lfloor \log Port(x,y) \rfloor + 1) \leq n + \sum_{(x,y)\in E_{esc}} \texttt{Port}(x,y) \\ &\leq 4n \;=\; O(n). \end{aligned} \qquad \blacksquare$$

Finally, Lemmas 8 and 9 yield the desired bound.

Lemma 10. $\texttt{Cost}(\mathcal{T}, G) = O(n)$.

References

1. Abraham, I., Gavoille, C., Malkhi, D., Nisan, N., Thorup, M.: Compact name-independent routing with minimum stretch. In: Proc. 16th ACM Symp. on Parallel Algorithms & Architectures (SPAA), pp. 20–24 (2004)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. McGraw-Hill, New York (1998)
3. Awerbuch, B., Bar-Noy, A., Linial, N., Peleg, D.: Compact distributed data structures for adaptive network routing. In: Proc. 21st ACM Symp. on Theory of Computing, pp. 230–240 (1989)
4. Awerbuch, B., Peleg, D.: Routing with polynomial communication-space trade-off. SIAM J. on Discrete Math. 5, 151–162 (1992)
5. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 335–346. Springer, Heidelberg (2005)
6. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Labeling Schemes for Tree Representation. In: Pal, A., Kshemkalyani, A.D., Kumar, R., Gupta, A. (eds.) IWDC 2005. LNCS, vol. 3741, pp. 13–24. Springer, Heidelberg (2005)
7. Fraigniaud, P., Gavoille, C.: Routing in trees. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 757–772. Springer, Heidelberg (2001)

8. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. In: Proc. 12th ACM Symp. on Discrete Algorithms, pp. 210–219 (2001)
9. Kannan, S., Naor, M., Rudich, S.: Implicit Representation of Graphs. SIAM J. on Descrete Math. 5, 596–603 (1992)
10. Katz, M., Katz, N.A., Korman, A., Peleg, D.: Labeling Schemes for Flow and Connectivity. SIAM J. Computing 34, 23–40 (2004)
11. Korman, A., Kutten, S., Peleg, D.: Proof Labeling Schemes. In: Proc. 24th ACM Symp. on Principles of Distributed Computing, PODC (2005)
12. Korman, A., Peleg, D., Rodeh, Y.: Labeling Schemes for Dynamic Tree Networks. Theory of Computing Systems (Special Issue of STACS 2002) 37, 49–75 (2004)
13. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1995)
14. Peleg, D.: Proximity-preserving labeling schemes and their applications. In: Widmayer, P., Neyer, G., Eidenbenz, S. (eds.) WG 1999. LNCS, vol. 1665, pp. 30–41. Springer, Heidelberg (1999)
15. Peleg, D.: Informative labeling schemes for graphs. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 579–588. Springer, Heidelberg (2000)
16. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM, Philadelphia (2000)
17. Peleg, D., Upfal, E.: A tradeoff between size and efficiency for routing tables. J. ACM 36, 510–530 (1989)
18. Santoro, N., Khatib, R.: Labelling and implicit routing in networks. The Computer Journal 28, 5–8 (1985)
19. Thorup, M., Zwick, U.: Compact routing schemes. In: Proc. 13th ACM Symp. on Parallel Algorithms & Architectures (SPAA), pp. 1–10 (2001)
20. van Leeuwen, J., Tan, R.B.: Routing with compact routing tables. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 259–273. Springer, Heidelberg (1986)