Succinct tree coding for greedy navigation

Zoltán Király*

Department of Computer Science and
Egerváry Research Group (MTA-ELTE)
Eötvös University
Pázmány Péter sétány 1/C, Budapest, Hungary
kiraly@cs.elte.hu

Sándor Kisfaludi-Bak

Department of Computer Science
Eötvös University
Pázmány Péter sétány 1/C, Budapest, Hungary
kbsandor@cs.elte.hu

Abstract: We present a succinct tree coding that enables *greedy routing* in arbitrary connected graphs. The worst-case length of vertex addresses is at most $3\log n + \log\log n + 3.59$ bits for n vertex graphs. We define a distance function with which greedy message forwarding is guaranteed to deliver messages between any pair of vertices in the graph.

Keywords: Navigation, Greedy routing

1 Introduction

The need to assign codes to the vertices of a tree emerges naturally in many applications. One of these applications is the navigation or routing problem of graphs. The task is to deliver messages between any pair of vertices, using only local information and the label of the recipient. The most simple such algorithm is the greedy routing algorithm, where the labels correspond to points in some space which is equipped with a distance function f, and each vertex forwards the message to the neighbour that is the closest to the recipient according to f.

Finding a space in which to embed our graph such that the greedy routing algorithm is guaranteed to work is a non-trivial task. Messages may get stranded if there is no neighbour of the current vertex v that is closer to the recipient than vertex v itself.

Several embedding methods were created ([6], [8]) that solved the issue of stranded messages. The downside of these methods is that they use $\Theta(n \log n)$ bits to encode labels, which is no better than prescribing a forwarding place for each possible recipient. Reducing the label length is a main concern for applications as well.

The paper of Eppstein and Goodrich [2] was one of the first attempts to give a *succinct* greedy embedding, i.e., an embedding requiring only $O(\log n)$ bits for each vertex, using abstract coordinates, but these coordinates could still be assigned to points on a hyperbolic plane. Due to a flaw in the method the delivery is not guaranteed here.

Greedy embeddings do not need to rely on metric spaces. The paper of Zhang and Govindaiah [9] uses a semi-metric embedding (a metric space without triangle inequality), and guarantee $O(\log n)$ length address for 3-connected planar graphs, or any graph that has a spanning tree with bounded degree. Note that the final code length is more than $\Delta \log n$ because some separators need to be encoded. (Here Δ denotes the maximum degree of the chosen spanning tree.) When trying to use this embedding it is unclear whether there is a spanning tree with low maximal degree in an arbitrary connected graph, so this method still falls short of achieving the $O(\log n)$ address length in the most general sense.

Independently of our research, a paper by Althofer et al. [1] proves that there is a nearest common ancestor labelling for any tree with n vertices. It is straightforward to prove that this labelling can be used for greedy routing in our sense. They construct a method for representing the addresses using

*Research was supported by grants (CNK 77780 and no. K 109240) from the National Development Agency of Hungary, based on a source from the Research and Technology Innovation Fund.

$3\lfloor \log n \rfloor$ bits. (Note that they also show a labelling with shorter addresses, but the coding and decoding process has no known polynomial algorithm.) While this bound is superior to ours in terms of maximum address length (we have a $\log \log n + 3$ additive term after $3 \log n$), this embedding uses close to $3\lfloor \log n \rfloor$ bits in practice, as it is demonstrated empirically in [5].

Our method is inspired by the embedding given by Eppstein and Goodrich [2], but our 'distance' function will not be a semi-metric: it will not be a symmetric function. It can be argued that the this is not a distance function, nevertheless we use this term because it describes the usage of the function. The main focus of the construction is to reduce code length as much as possible.

2 Preliminaries

Let G be an arbitrary simple and connected graph. We denote by $N_G(v)$ the set of neighbours of $v \in V(G)$. Let f be an *embedding* into a set S, i.e., an injective function $f : V(G) \to S$, and let *dist* be a *distance* function: $dist : S \times S \to \mathbb{R}_{\geq 0}$. (Note that it is not required for *dist* to be a metric on S.) Using f, we can define the distance from u to v as $d(u, v) = dist(f(u), f(v))$.

Messages that are sent over the network have a specific sender $s \in V(G)$ and target $t \in V(G)$, and the message header contains $f(t) \in S$, that is, the *address* of t. The *greedy routing scheme* sends the message from v to the vertex $w \in N(v)$ for which $d(w, t)$ is minimal. (If v has multiple neighbours at the same distance to t, we choose a vertex among these arbitrarily.) The pair $(f, dist)$ defines a *guaranteed greedy embedding* if the greedy routing scheme can deliver any message.

We are now ready to formulate the main theorem.

Theorem 1 *For a connected graph G there is an assignment $f : V(G) \to \{0, 1\}^t = S$ and a function $d : S \times S \to \mathbb{R}_{\geq 0}$ such that (f, d) is a guaranteed greedy embedding with code length $t \leq \lfloor 3 \log n \rfloor + \lceil \log \lfloor 3 \log n \rfloor \rceil + 1 \leq 3 \log n + \log \log n + 3.59$.*

An embedding of a graph G which satisfies that for any $v, t \in V(G)$, $v \neq t$ there is a vertex $w \in N(v)$ such that $d(w, t) < d(v, t)$ is called a *distance decreasing* embedding. It is straightforward to prove that any distance decreasing embedding is a guaranteed greedy embedding. The following claim is also easy to verify.

Claim 2 *Let G' be a connected spanning subgraph of G. Then any distance decreasing greedy embedding of G' is a distance decreasing greedy embedding of G.*

Note that Claim 2 greatly simplifies our task: it is sufficient to give a distance decreasing greedy embedding of a spanning tree T of G. If we tried to deliver messages only in T, then the message would make very big detours compared to the shortest path in G. Fortunately, the routing considers all edges of G, so the number of hops taken by the routing is comparable to the shortest path in G.

To construct our embedding, we use various trees. Our trees are usually rooted and ordered, that is, for every vertex v there is an ordering defined on its children. Occasionally, we use partially ordered trees, where we only specify partial orderings on the children of each vertex. For binary trees, we call the first child as left child and the other as right child. If a vertex has only one child, then it will be regarded as a left child. For each vertex we define its *depth* (notation: $\mathrm{depth}(v)$) as its distance from the root. We define the *subtree* of a vertex v as the tree spanned by v and all its descendants.

For the rest of this section we discuss binary trees. We associate a bit sequence and a number with a vertex v of a rooted and ordered binary tree. (From now on we use the abbreviation ROBT for rooted ordered binary trees.) The bit sequence $\mathrm{code}(v)$ is defined recursively: for the root r let $\mathrm{code}(r)$ be the empty bit sequence, and for any other vertex v with parent p let $\mathrm{code}(v)$ be $\mathrm{code}(p)^\frown 0$ if v is the left child or $\mathrm{code}(p)^\frown 1$ if v is the left child. (The notation '$^\frown$' is used for the concatenation of two bit sequences.)

Let $\mathcal{S}$ be the set of finite $0-1$ sequences. The code function defines a bijection between the vertices of the infinite binary tree B_∞ and $\mathcal{S}$. Vertices of a ROBT are referred to by their code, and vice versa.

Another number that we associate with the vertex of a ROBT will be denoted by $\mathrm{diad}(v)$, and we use the same definition that Eppstein and Goodrich [2] does: for the root r let $\mathrm{diad}(r) = \frac{1}{2}$ and for a vertex

v with parent p let $\mathrm{diad}(v) = \mathrm{diad}(p) \pm 2^{-\mathrm{depth}(v)-1}$, where we add $2^{-\mathrm{depth}(v)-1}$ if v is a right child and subtract if v is a left child. Observe that for any vertex v, $0 < \mathrm{diad}(v) < 1$ and the bit sequence $\mathrm{code}(v)^\frown 1$ is the sequence of binary fractions of $\mathrm{diad}(v)$. Note that diad defines an ordering on the vertices of the binary tree. In the rest of the paper saying that u is on the left of v is equivalent to $\mathrm{diad}(u) < \mathrm{diad}(v)$.

3 Embedding method

Our first goal is to embed a spanning tree into a binary tree, and use the encodings described in Section 2 to make binary sequences.

We begin by choosing an arbitrary spanning tree T of G, and we also a fix a root $r_T \in V(T)$. Let $\mathrm{des}(v)$ be the number of descendants of v: the number of vertices in the subtree defined by v, including v itself. (E.g., $\mathrm{des}(v) = 1$ for leaves and $\mathrm{des}(r_T) = |V(T)|$.) Take an edge uv where u is the parent of v. The edge uv is *heavy* if $\mathrm{des}(v) \geq \mathrm{des}(u)/2$. Note that each vertex has at most one outgoing heavy edge. All other edges are *light*. We say that a child is heavy or light accordingly.

We introduce a partial ordering on the children of every vertex in T that has a heavy child: we simply require that the heavy edge is the first (leftmost) edge. The ordering of the light children is not fixed. The heavy edges together form vertex disjoint *heavy paths*. For a vertex v on a heavy path P we define its child group as the set of light children: $\{w \in V(T) \mid vw \in E(T),\ w \notin V(P)\}$. By contracting the heavy paths we get the new tree Z, and we denote the effect on the vertices by $\varphi : V(T) \to V(Z)$.

Next, we produce a binary tree B with root r_B that is essential in our embedding. This binary tree is an aggregate of smaller binary trees. It is constructed in a bottom-up manner, creating small binary trees for each heavy path (called *heavy path tree*s, or in short HPTs), and a small binary tree for the light children of every vertex (called *light children tree*s, or in short LCTs). Due to space constraints, we only give an outline of the method. A more detailed write-up is available in [4].

In each HPT, we require that the leafs are assigned to the vertices on the heavy path, with the same left-to-right ordering as in the heavy path. In case of single vertex heavy paths, the tree is required to be the single vertex tree. In an LCT made for the children of $v_T \in V(T)$, the leaves correspond to the light children of v_T. The method for building these small ROBTs will be described in Subsection 3.1.

These LCTs and HPTs are glued together in a natural way: a vertex v in $V(T)$ corresponds to a leaf of the HPT of its heavy path, and to the root of its LCT. We glue these trees together: the root of the LCT will be the leaf of the HPT.

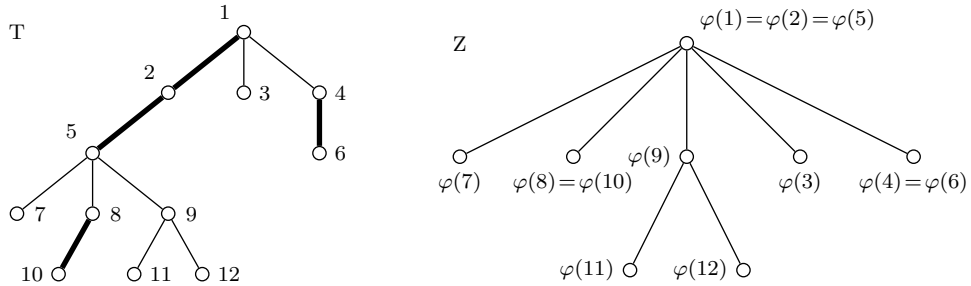


Figure 1: The first two steps of our embedding: finding the heavy paths in the spanning tree, and the tree Z obtained by contracting the heavy paths.

By gluing all the HPTs and LCTs together, we arrive at the final binary tree B. We will construct the tree in a bottom-up manner, so we can balance the small trees with the information about the height of their leaves in the final tree B. For a vertex $v \in G$, let $c_1(v)$ be the vertex of B that is the root of its HPT, and let $c_2(v)$ be the leaf of the HPT that corresponds to v. See Figure2 for the values of c_1 and c_2 in our previous example.

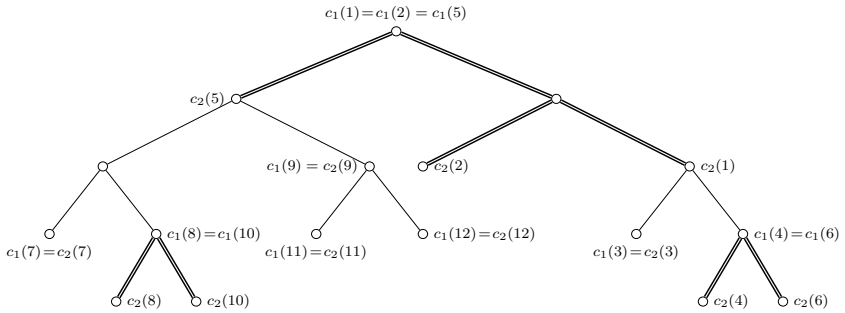


Figure 2: The values of c_1 and c_2. The double and single edges represent HPT and LCT edges respectively.

Notice that $c_1(v)$ is always an ancestor of $c_2(v)$, and for single-vertex heavy paths, $c_1(v) = c_2(v)$ holds. Moreover, the function c_1 can be thought of as a function from $V(Z)$ to B, and this function keeps ancestor-descendant relations between vertices of Z, and for $v, w \in V(T)$ who are on the same heavy path, $\mathrm{diad}(c_2(v)) < \mathrm{diad}(c_2(w))$ if and only if v is a descendant of w.

Finally, we need to encode the vertices $c_1(v)$ and $c_2(v)$ in a single bit sequence. The bit sequences $\mathrm{code}(c_1(v))$ and $\mathrm{code}(c_2(v))$ are easy to encode, we use the following standard procedure:

Lemma 3 *Let s_2 be a bit sequence of length k with a starting slice s_1. Then s_1 and s_2 can be encoded in a machine word of length at least $k + \lceil \log k \rceil + 1$.*

Proof: Given the bit sequence s_1 and s_2, we use the lower $k + \lceil \log k \rceil + 1$ bits of the machine word. We set the first of these bits to 1, then follow it with $\lceil \log k \rceil$ bits on which we write the length of s_1. Note that the length of s_1 is at most k, so its length can be represented on $\lceil \log k \rceil$ bits. Finally, we write s_2 on the remaining k bits. To encode such a sequence, we need to determine k. First, we find the highest bit that is set, that gives us the value $f(k) = k + \lceil \log k \rceil + 1$. It is straightforward to check that for any $k \geq 1$ the value $\lceil \log f(k) \rceil$ is either $\lceil \log k \rceil$ or $\lceil \log k \rceil + 1$. This also gives us the value of k. □

3.1 Optimal depth binary trees

We are looking for an embedding that provides a minimum depth binary tree B, given our definition of heavy edges, our ordering restrictions and our requirements on the heavy path trees (HPTs) and light children trees (LCTs).

First, we define the LCT building method. Let v_T be a vertex of T that has at least one light child. We define z_i to be $\mathrm{height}_B(w_i)$, where $w_i \in B$ is the root of the HPT that corresponds to the heavy path of the i-th light child of v_T. These depths are available to us by our construction in 3. The LCTs will be constructed by a slight modification of Huffman's coding algorithm [3]. For the unordered sequence $\mathcal{Z} = \{z_1, z_2, \ldots, z_k\}$, we choose the two smallest values (z_i and z_j), and build a tree of 3 vertices, where the two children of the root correspond to z_i and z_j. In $\mathcal{Z}$, we delete z_i and z_j, and add $1 + \max\{z_i, z_j\}$. We continue taking the two smallest value from $\mathcal{Z}$ and adding the new element (their maximum plus 1), until there is only one element left in $\mathcal{Z}$, and its value is the depth of the optimal subtree. In the meantime we also gave the construction for the optimal ROBT.

We now turn to the HPT building method. We use a dynamic programming algorithm which is a modification of the naïve implementation of Knuth's algorithm for binary search trees [7]. We require an initial depth for each heavy path vertex: let z_i be the the $\mathrm{height}_B(w_i)$ where $w_i \in B$ is the root of the LCT that corresponds to the i-th vertex of the heavy path. Again, these depth are available to us by our construction in Section 3.

We use dynamic programming to determine the optimal depth. If we are given a sequence $z_1, z_2, \ldots, z_k$, we are looking for a ROBT that has each z_i assigned to a leaf v_i, and

$$\max_{i \in \{1,\ldots,k\}} \{\mathrm{depth}(v_i) + z_i\} \qquad (*)$$

(the depth of the whole subtree) is minimal, while the leaves $v_1, v_2, \ldots, v_k$ are ordered from left to right. Let Odt_i^j denote the optimal depth tree for the sequence $z_i, z_{i+1}, \ldots, z_j$, and let od_i^j be the optimal subtree depth (we take the maximum between i and j in equation $*$), for each $1 \le i \le j \le k$. We can set Odt_i^i to be the 1-vertex tree and $\mathrm{od}_i^i = 0$. The dynamic programming will compute the values od_i^j in increasing order of $j - i$ using the formula $\mathrm{od}_i^j = \min_{i \le t < j} \max\{1 + \mathrm{od}_i^t, 1 + \mathrm{od}_{t+1}^j\}$. Let m be an index for which this formula takes its minimum value, and let r be the root of Odt_i^j, to which we add two children. The tree Odt_i^j can be constructed by attaching the tree Odt_i^m by its root to the left child of r and attaching the tree Odt_{m+1}^j by its root to the right child of r. It is routine to prove that this construction gives a minimum depth tree.

4 Computing the 'distance'

Recall that the function named code defined in Section 2 maps the vertices of the rooted binary tree B into $0-1$ sequences. Let $d_B : V(B) \times V(B) \to \mathbb{N}$ denote the (hop) distance of two vertices in B. Furthermore, let $a \wedge_W b$ denote the lowest common ancestor of a and b in the rooted tree W.

Our asymmetric distance function $d : V(T) \times V(T) \to \mathbb{R}_{\ge 0}$ is the sum of two functions: $d = d_1 + d_2$, where the summands are defined in the following way:

$$d_1(a,b) = \begin{cases} d_B(c_1(a), c_1(b)) & \text{if } c_1(a) \wedge_B c_1(b) \in \{c_1(a), c_1(b)\} \\ d_B(c_1(a), c_1(b)) + d_B(r_B, c_1(a) \wedge_B c_1(b)) & \text{otherwise} \end{cases}$$

$$d_2(a,b) = \begin{cases} \dfrac{\mathrm{diad}(c_2(a))}{2} & \text{if } a \wedge_T b = a \\ 1 - \dfrac{\mathrm{diad}(c_2(a))}{2} & \text{otherwise} \end{cases}$$

In order to make this distance function suitable for greedy routing, we need to be able to compute it using only the values of c_1 and c_2. It is easy to see that all cases of both d_1 and d_2 are defined in terms of c_1 and c_2, so the problem lies in distinguishing between the cases. This issue is resolved in [4].

Theorem 4 *Our embedding is greedy, that is, using the distance function $d(v,t)$ for greedy forwarding algorithm (where the recipient is t) will always deliver the message in a finite number of steps.*

The proof of this theorem relies on the following claim:

Claim 5 *Let $H = (v_1, v_2, \ldots, v_k)$ be a heavy path in T with topmost vertex v_k, and let b be a descendant of v_t in T (where $1 \le t \le k$). Then the following inequalities hold:*

$$d_2(v_1,b) > d_2(v_2,b) > \cdots > d_2(v_t,b) \text{ and } d_2(v_t,b) < d_2(v_{t+1},b) < \cdots < d_2(v_k,b).$$

The proof of this claim and the detailed proof of Theorem 4 can be found in [4].

5 Alternative ROBT building method

We describe a second ROBT building procedure. Although this procedure yields a binary tree of suboptimal depth, it is easier to upper bound the depth of its resulting tree, and it has the same worst-case performance as the optimal building method.

For both the HPTs and LCTs we define a weight sequence. Each individual weight in this sequence will be assigned to a leaf of the HPT or LCT. We fix an ordering on the light children for every vertex (recall that this was not fixed previously): let the left-to right ordering of the children be decreasing in the number of descendants, i.e., for child u and v if $\mathrm{des}(u) > \mathrm{des}(v)$ then u is on the left of v. (We break ties arbitrarily.) When building our LCTs and HPTs, we preserve the ordering, that is, the ordering of

the assigned weights of the leaves from left to right will be the same as the ordering of the original weight sequence.

We can assign a weight sequence to each vertex of an ROBT: the sequence assigned to vertex v consists of the weights assigned to the descendant leaves of v, and the ordering is the same as the left-to-right ordering on those leaves. This assignment maps the root of the ROBT to the full initial weight sequence.

Let $v_0, v_1, \dots v_k$ be the vertices of a heavy path traversed upward, so the topmost vertex is v_k. Let the weight sequence for the the root of the HPT be $(\mathrm{des}(v_0), \mathrm{des}(v_1) - \mathrm{des}(v_0), \mathrm{des}(v_2) - \mathrm{des}(v_1), \dots, \mathrm{des}(v_k) - \mathrm{des}(v_{k-1}))$, or in other words, the i-th weight is the total sum of its child group's descendants plus 1. For a vertex with light children $\{u_1, u_2, \dots u_k\}$ let the weights of the LCT root be $(\mathrm{des}(u_1), \mathrm{des}(u_2), \dots, \mathrm{des}(u_k))$ (so in this case, the weights are a non-increasing sequence of integers).

Let $\underline{w} = (w_1, w_2, \dots, w_k)$ be a weight sequence ($w_i \in \mathbb{N}^+$). Let $s_i = \sum_{j=1}^{i} w_j$ (for $i = 1, 2, \dots, k$), and let $|\underline{w}| = s_k$ be the total weight of the weight sequence. These s_i values are called *separators*. A *cut* of the weight sequence $\underline{w}$ along separator s_i ($1 \le i \le k-1$) divides the weight sequence into two child sequences. The left child sequence is $(w_1, w_2, \dots w_i)$ and the right child sequence is $(w_{i+1}, w_{i+2}, \dots, w_k)$.

We define our ROBT building method as a sequence of cuts. For each weight sequence, we need to find a separator at which we perform the cut. This will be defined depending on the weight sequence class (these classes will be defined later). The left and right child sequences will be assigned to the two children of the current vertex. For each child, we classify the weight sequence and make a cut accordingly. Continuing this cutting method recursively leads to a well-defined ROBT.

5.1 Cuts and weight sequence classes

Let $\underline{w} = (w_1, w_2, \dots, w_k)$ be a weight sequence with total weight $W = |\underline{w}|$. We denote by b the index of the biggest weight. (If there are multiple biggest weights, we set b to the least index among them.) Let α be the ratio of the biggest weight to the sum of weights, that is, $w_b = \alpha W$. Furthermore, we denote by ζ the constant $2^{-1/3} \approx 0.794$.

We use one of the following methods to determine the separator for a cut.

Balanced cut: Let s_t be the separator for which $|W/2 - s_t|$ is minimal (the separator closest to the 'middle' of the sequence).

Left-heavy cut: Let s_t be the least separator not smaller than $W/2$ (so $s_t \ge W/2$). If $s_t > \zeta W$ and $t = 1$, then we cut at s_t. If $s_t > \zeta W$ and $t > 1$, then we cut at s_{t-1}. If $s_t \le \zeta W$ then let j be the index of the biggest weight in $(w_{t+1}, w_{t+2}, \dots w_k)$. If $s_{j-1} > \zeta W$ then we cut at s_t, otherwise we cut at s_{j-1}. Notice that this method is designed to make the cut in a way that the the second weight sequence has its biggest weight in its first element as long as the left child is not too large.

Right-heavy cut: We perform the 'reflection' of left-heavy cut: we cut at $W - x$ where x is the separator for left-heavy cut on the reverse weight sequence $(w_k, w_{k-1}, \dots, w_1)$.

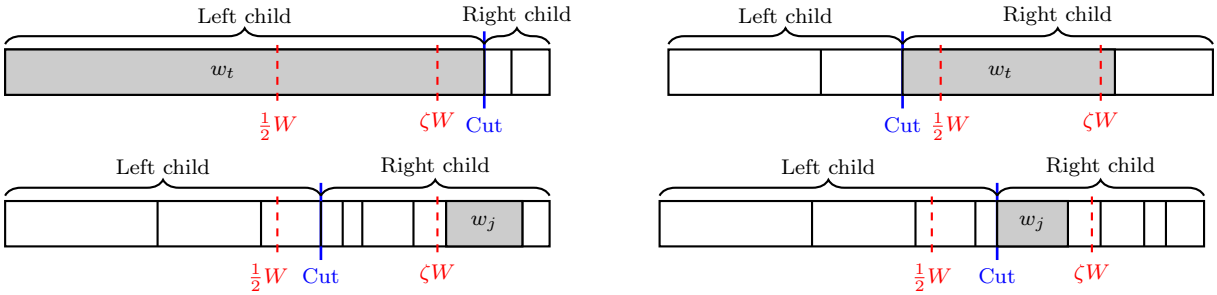


Figure 3: The four possible cases of a left-heavy cut

Next, we describe our weight classes. The classes are organized into class types. Most class types have a real parameter α, the ratio of the biggest weight to the sum of the weights in the weight sequence. The weight sequence classes themselves are just class types with the parameter value fixed.

In case of HPT building, we need to classify only those weight sequences that are assigned to non-leaf vertices of the HPT, since the weights assigned to leafs will be treated in the LCT attached to this leaf, or in case of weight 1, the vertex will be a leaf of the large binary tree. Notice that in case of HPTs every non-leaf vertex has a weight sequence consisting of at least two weights.

We approach the LCTs similarly: we only classify non-leaf vertices. Contrary to HPTs though, it is possible that a non-leaf vertex has a singleton weight sequence assigned to it: if the HPT corresponds to a vertex v_T that is a non-ending point of its heavy path, and v_T has a single light descendant, then the root of the LCT will be assigned a single weight.

Weight sequence classes arising when creating an LCT:

- $\mathbf{D}_\alpha$ $(0 < \alpha \leq 1)$. Sequence $\underline{w}$ is in $\mathbf{D}_\alpha$ (*'Decreasing'*) if $w_1 = \alpha|\underline{w}|$. (Recall that for LCTs, $w_1 \geq w_2 \geq \cdots \geq w_k$, so the child sequences of a cut are also in a decreasing order.) We use balanced cut to cut class $\mathbf{D}_\alpha$.

Weight sequence classes arising when creating an HPT:

- $\mathbf{H}$. Sequence $\underline{w}$ is in $\mathbf{H}$ (*'Heavy'*) if $\underline{w}$ contains the first weight of the weight sequence assigned to the root of the HPT. When cutting weight sequences of class $\mathbf{H}$, we are going to use left-heavy cuts.
- $\mathbf{FL}_\alpha$ $(0 < \alpha < 1)$. Sequence $\underline{w}$ is in FL_α (*'First-Last'*) if it is not in $\mathbf{H}$, and its biggest weight (or one of its biggest weights) is the first weight or the last weight, so $w_1 = \alpha|\underline{w}|$ or $w_k = \alpha|\underline{w}|$. If $\alpha > \frac{1}{2}$, then we use a balanced cut. If $\alpha \leq \frac{1}{2}$ and $w_1 = \alpha|\underline{w}|$, then we will use a left-heavy cut, otherwise (if $\alpha \leq \frac{1}{2}$ and $w_k = \alpha|\underline{w}|$) we use right-heavy cut.
- $\mathbf{M}_\alpha$ $(0 < \alpha < 1)$. Sequence $\underline{w}$ is in M_α (*'Middle'*) if the biggest weight is some weight $w_j = \alpha|\underline{w}|$, such that $1 < j < k$ and $w_1 \neq w_j \neq w_k$. We apply balanced cut for this class.

Now we have fully described our HPT and LCT building method.

6 Bounding the height of the large binary tree

We define a constant for each weight sequence class. These constants are denoted by the non-bold version of the name of the class, e.g., the class constant of $\mathbf{D}_\alpha$ is D_α.

For a function $f(x)$ let $(f(x))^+ = \max(0, f(x))$. We define our class constants the following way:

$$\begin{aligned} D_\alpha &= -1 + (2 + 3\log(\alpha))^+ \\ H &= 0 \\ FL_\alpha &= 1 + (1 + 3\log(\alpha))^+ \\ M_\alpha &= 2 + (1 + 3\log(\alpha))^+ \end{aligned}$$

Note that the functions $(2 + 3\log(\alpha))^+$ and $(1 + 3\log(\alpha))^+$ are monotone increasing, and their value is 0 on the interval $\left[0, \zeta^2\right]$ and $[0, \zeta]$ respectively.

Usually we are looking for a higher bound on the class constant. We denote by D_1, FL_1 and M_1 the supremum of the constants in each class type. (By our definition, these values are $1, 2$ and 3 respectively.)

Let $\text{height}_B(v)$ be the height of v in the large binary tree B (or equivalently, the height of the subtree of v in B, so $\text{height}_B(r_B)$ is the height of B). Our upper bound for the subtree depth of a vertex v in the binary tree is $3\log W + c$, where c is the class constant of the weight sequence of v, and W is the total weight of v's weight sequence class. We will state this more precisely below. The following two theorems will be the key to prove our main theorem.

Theorem 6 *Let α be the parameter of the weight sequence $\underline{w} = (w_1, w_2, \ldots w_k)$ assigned to the root r of an LCT. Suppose that* $\text{height}_B(l_i) \leq 3\log w_i$*, where l_i is the leaf to which we assigned the weight w_i in the LCT. It follows that* $\text{height}_B(r) \leq 3\log W + D_\alpha$*. (As before, W is the total weight of $\underline{w}$.)*

Theorem 7 *Let r be the root of an HPT with weight sequence $(w_1, w_2, \ldots w_k)$ and total weight W. Suppose that* $\text{height}_B(l_1) \leq \max(3\log w_1 - 1,\ 0)$ *and* $\text{height}_B(l_i) \leq 3\log w_i + 1$ *for $2 \leq i \leq k$, where l_i is the leaf to which we assigned the weight w_i in the HPT. It follows that* $\text{height}_B(r) \leq 3\log W$*.*

Finally, we will prove the following upper bound on the height of the large binary tree B:

Theorem 8 *The height of B using the optimal tree building method is at most $3 \log n$, where n is the number of vertices in the original graph.*

The following lemma will be used multiple times.

Lemma 9 *Let $\underline{w}$ be the weight sequence with at least 3 weights, and let w_m be the weight that covers the midpoint. Suppose that either $w_m \leq s_{m-1}$ or $w_m \leq W - s_m$. After a balanced cut in $\underline{w}$, both the resulting weight sequences have total weight at most $\frac{2}{3}W$.*

PROOF: We prove this lemma by contradiction. Suppose that the cut happens outside the interval $\left[\frac{1}{3}W, \frac{2}{3}W\right]$. It follows that the middle weight covers the whole interval, so its size must be bigger than $W/3$. Thus $s_{m-1} < W/3$ and $W - s_m < W/3$, so we get that $s_{m-1} < w_m$ and $W - s_m < w_m$. This contradicts the condition in our lemma. □

Our aim is to upper bound the total weight of the child sequences with a constant factor of the total weight of the parent sequence (denoted by W). We will also need to give upper bounds for the constants of the new classes. A $(\mathbf{C}, r, p)$-*child* is a child weight sequence which is in class $\mathbf{C}$, has total weight at most rW, and its class parameter is at most p. (For classes without parameters we omit the third term in the brackets.)

6.1 Proof of Theorem 6 and an outline for the proof of Theorem 7

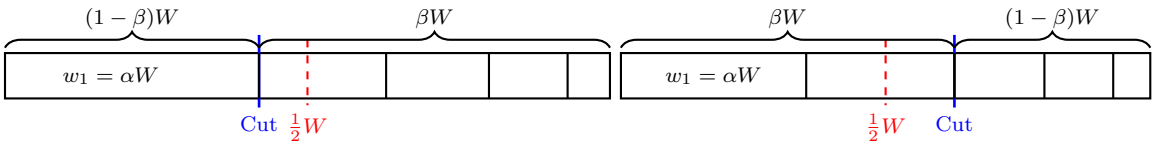
A special kind of a child is a leaf of the LCT. This child sequence always contains a single weight. We denote this child by $(\mathbf{L}, r)$ where r is the weight ratio as before.

Lemma 10 *The possible children of $\mathbf{D}_\alpha$ after applying our cutting method are $(\mathbf{L}, \alpha)$ and $(\mathbf{D}, \beta, \frac{1-\beta}{\beta})$, where $\frac{1}{2} \leq \beta \leq \frac{2}{3}$.*

PROOF: If there is a single child class, then $\alpha = 1$, and the child is a leaf of weight ratio 1. Otherwise the smaller child class is either a leaf with weight ratio at most α, or it has class type $\mathbf{D}$, has weight ratio at most $\frac{1}{2}$, and its parameter is at most 1, so it is a $(\mathbf{D}, \frac{1}{2}, 1)$ child. This falls under the category of $(\mathbf{D}, \beta, \frac{1-\beta}{\beta})$, where $\beta = \frac{1}{2}$

The larger child class can also be a leaf, with weight at most α. If it is not a leaf, then it has class type $\mathbf{D}$. Let β be the weight ratio of the larger class. By Lemma 9, $\beta \leq \frac{2}{3}$.

The parameter of the child is the ratio of its first weight to its total weight. Its largest weight has size at most αW. If this larger child class is the right child, then it does not contain the first weight $w_1 = \alpha W$, so $\beta \leq 1 - \alpha$ follows. If this is a left child class, then by the definition of balanced cuts the separator at βW is closer to $\frac{1}{2}$ than the αW separator. It follows that $\alpha \leq \frac{1}{2}$ and $\beta - \frac{1}{2} \leq \frac{1}{2} - \alpha$, or equivalently, $\alpha \leq 1 - \beta$. So in both cases, the parameter of this child is at most $\frac{\alpha}{\beta} \leq \frac{1-\beta}{\beta}$, making it a $(\mathbf{D}, \beta, \frac{1-\beta}{\beta})$ child. □



PROOF: (of Theorem 6) To prove the theorem, we will prove the stronger proposition that for any non-leaf vertex v of the LCT with weight sequence $\underline{w}$, the inequality $\text{height}_B(v) \leq 3 \log W + D_\alpha$ holds, where W is the total weight of $\underline{w}$, and $\mathbf{D}_\alpha$ is its weight sequence class.

We prove the statement by induction on W. For $W = 1$, we know that $\alpha = 1$, and the LCT has only two vertices, its root r and the leaf l of weight one, that is a leaf of the big binary tree. Since $\text{height}_B(l) = 0$, we get $\text{height}_B(r) = 1 = 3 \log 1 + (-1) + (2 + 3 \log 1)^+ = 1$.

Now suppose that $W \geq 2$. In the special case of a single weight class, the child is always a leaf, so by the theorem's condition on leaves ($\text{height}_B(l) \leq 3 \log W$ for any leaf l), we get

$$\text{height}_B(r) = 1 + \text{height}_B(l) \leq 1 + 3 \log W = 3 \log W + D_1.$$

In all other cases, we can bound $\text{height}_B(v)$ the following way:

$$\text{height}_B(v) \leq \max\left(1 + \text{height}_B(v_1), 1 + \text{height}_B(v_2)\right),$$

where v_1 and v_2 are the children of v in the LCT. Since we already have an upper bound for both $\text{height}_B(v_1)$ and $\text{height}_B(v_1)$, we can apply induction because their weights are strictly smaller than W, so it is sufficient to prove that $3 \log W + D_\alpha$ is not smaller than this bound. Using Lemma 10 we arrive at the following inequalities:

$$3 \log W + D_\alpha \geq 1 + 3 \log(\alpha W) \tag{1}$$

$$3 \log W + D_\alpha \geq 1 + 3 \log(\beta W) + D_{\frac{1-\beta}{\beta}} \qquad \text{where } \beta \in \left[\frac{1}{2}, \frac{2}{3}\right]. \tag{2}$$

We subtract $3 \log W$, and substitute the class constants. In the second inequality, we decrease the left side by taking the minimum possible value of D_α, which is -1.

$$-1 + (2 + 3 \log \alpha)^+ \geq 1 + 3 \log \alpha \tag{3}$$

$$-1 \geq 1 + 3 \log \beta + -1 + \left(2 + 3 \log \frac{1-\beta}{\beta}\right)^+ \qquad \text{where } \beta \in \left[\frac{1}{2}, \frac{2}{3}\right] \tag{4}$$

Inequality (3) is straightforward. For inequality (4) we distinguish two cases. If $\frac{1-\beta}{\beta} \leq \zeta^2$, then $0 \geq 1 + 3 \log \beta \Leftrightarrow \beta \leq \zeta$, which is satisfied since $\beta \leq \frac{2}{3} < \zeta$. If $\frac{1-\beta}{\beta} > \zeta^2$:

$$3 \log \beta + 2 + 3 \log \frac{1-\beta}{\beta} = 2 + 3 \log(1 - \beta) \leq -1,$$

where the last inequality holds because $\beta \geq \frac{1}{2}$. $\square$

The proof of Theorem 7 uses the same technique. We mention the necessary definitions and the main steps of its proof, and point the reader to [4] for the detailed proof.

We will need two special classes for the leaves of an HPT. The leftmost leaf will be denoted by **SL** (*starting leaf*), and the rest of the leaves will be denoted by **OL** (*ordinary leaf*).

Lemma 11 *The possible children of* $\mathbf{H}$ *after applying the left-heavy cut are* $(\mathbf{SL}, 1)$, $(\mathbf{OL}, \frac{1}{2})$, $(\mathbf{H}, \zeta)$, $(\mathbf{FL}, 1-\zeta, 1)$, $(\mathbf{FL}, \beta, \frac{1-\beta}{\beta})$ *where* $\frac{1}{2} \leq \beta \leq 1 - \frac{\zeta}{2}$, *and* $(\mathbf{M}, \beta, \frac{1-\zeta}{\beta})$ *where* $1 - \zeta \leq \beta \leq \frac{1}{2}$.

Lemma 12 *The possible children of* $\mathbf{FL}_\alpha$ *are* $(\mathbf{OL}, \alpha, 1)$, $(\mathbf{M}, \frac{1}{2}, 1)$ $(\mathbf{FL}, \beta, \frac{\alpha}{\beta})$ *where* $\frac{1}{2} \leq \beta \leq \zeta; \alpha \leq \frac{1}{2}$, *and* $(\mathbf{FL}, \beta, \frac{1-\beta}{\beta})$ *where* $\frac{1}{2} \leq \beta \leq 1 - \frac{\zeta}{2}; \alpha \leq \frac{1}{2}$.

Lemma 13 *The possible children of* $\mathbf{M}_\alpha$ *are:* $(\mathbf{OL}, \alpha, 1)$, $(\mathbf{M}, \frac{1}{2}, 1)$, $(\mathbf{FL}, \frac{1}{2}, 1)$, $(\mathbf{FL}, \beta, \frac{\alpha}{\beta})$ *where* $\alpha \leq \beta \leq \frac{1+\alpha}{2}$, *and* $(\mathbf{M}, \beta, \frac{\alpha}{\beta})$ *where* $\frac{1}{2} < \beta \leq \frac{2}{3}; \alpha \leq \frac{1}{2}$.

It is now easy to produce the system of inequalities similar to (1) and (2) for each lemma above, and all those inequalities can be proven separately.

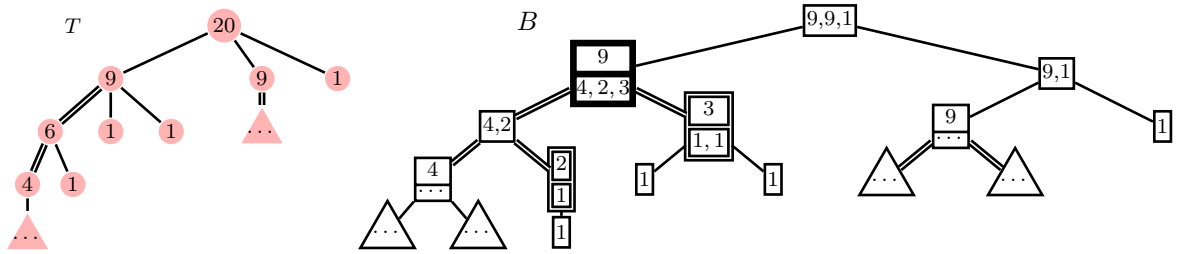


Figure 4: Left: original spanning tree with double lines indicating heavy edges. Unimportant subtrees are indicated with a small triangle. Right: the resulting binary tree, with the assigned weight sequences to each vertex. At HPT-LCT attaching points the top number corresponds to the leaf, the bottom sequence is the weight sequence assigned to the root.

6.2 Proof of Theorem 8 and Theorem 1

PROOF: (of Theorem 8) First, we observe that the binary tree B_0 that is obtained using the optimal tree building method of Subsection 3.1 has equal or less height as the tree B obtained from the weight based cutting method described in section 5. Thus it is sufficient to prove that B has height at most $3 \log n$.

Let $v_B \in B$ be a root of a HPT subtree $\mathcal{H}$ that is also a leaf of an LCT (we denote the LCT subtree by $\mathcal{L}$). Notice that the weight assigned to v_B in $\mathcal{L}$ is $\text{des}(u_T)$ for the light child $u_T \in V(T)$ corresponding to that leaf. In $\mathcal{H}$, the total weight of the weight sequence assigned to v_B is $\text{des}(u_0) + (\text{des}(u_1) - \text{des}(u_0)) + (\text{des}(u_2) - \text{des}(u_1)) + \cdots + (\text{des}(u_k) - \text{des}(u_{k-1})) = \text{des}(u_k)$ where $u_0, u_1, \ldots u_k$ are the vertices of the heavy path traversed upward, so the topmost vertex is $u_k = u_T$. Thus the total weight of the root weight sequence in $\mathcal{H}$ is equal to the weight assigned to the leaf in $\mathcal{L}$ (e.g. the double contour vertices in Figure 4).

Now let v_B be a root of an LCT subtree $\mathcal{L}$ which is also a leaf of a HPT denoted by $\mathcal{H}$. (E.g. the double contour vertices in Figure 4.) The weight assigned to v_B in $\mathcal{H}$ is either $\text{des}(v_T)$ if the corresponding vertex $v_T \in T$ is the bottom vertex of a heavy path, or $\text{des}(v_T) - \text{des}(v'_T)$ if its heavy child is v'_T. The total weight assigned to v_B in $\mathcal{L}$ is the sum of the descendants of the light children of v_T, which is exactly $\text{des}(v_T) - 1$ if v_T is the bottom vertex of its heavy path, or $\text{des}(v_T) - \text{des}(v'_T) - 1$ if its heavy child is v'_T.

To prove Theorem 8, we can invoke Theorems 6 and 7 going bottom up in the tree B. The conditions are straightforward to check for the HPT subtrees that have no LCT children in B.

For a HPT subtree $\mathcal{H}$, we need that $\text{height}_B(l_1) \leq \max(3 \log W - 1, 0)$, where W is the weight assigned to this leaf in $\mathcal{H}$. Let v_T be the corresponding vertex in T. The first leaf corresponds to the bottom of the heavy path. If the leaf has no children, then its weight is 1, so the maximum is 0 and the condition is satisfied. Otherwise it has light children only, so the total weight assigned to the root of $\mathcal{L}$ is $W - 1$ by our previous observation. Also note that the biggest weight has size at most $(W-1)/2$, otherwise there would be a heavy child, so v_T could not be the bottom vertex of its heavy path. By the stronger statement of our theorem, the root of the LCT has class parameter $\leq \frac{1}{2}$, so $\text{height}_B(l_1) \leq 3 \log(W - 1) - 1 < 3 \log W - 1$.

For the rest of the leaves, we need $\text{height}_B(l_i) \leq 3 \log W_i + 1$. This is straightforward since the total weight of l_i in L is $W_i - 1$, so $\text{height}_B(l_i) \leq 3 \log(W_i - 1) + 1 < 3 \log W_i + 1$ by induction.

For a LCT subtree $\mathcal{L}$, we need that $\text{height}_B(l_i) \leq 3 \log W$ where W is the weight assigned to l_i in $\mathcal{L}$. The leaf can be a leaf of B, in which case it has weight 1 and the height is 0, or it has a child HPT, which has total weight W by our previous observation. By induction, we have $\text{height}_B(l_i) \leq 3 \log W$ from the child HPT.

The topmost small ROBT of B is either a HPT or an LCT with parameter at most $\frac{1}{2}$. In the former case we get the upper bound $3 \log n$ on the height of B, and in the latter, we get $3 \log n - 1$. □

PROOF: (of Theorem 1) Choose any spanning tree T of G, perform our embedding. Use $\text{code}(c_1(v))$ and $\text{code}(c_2(v))$ in Lemma 3 to create a single bit sequence $f(v)$. Now f with the distance function d is a guaranteed greedy embedding by Theorem 4. Since the length of $\text{code}(c_2(v))$ is at most the depth of B,

it is at most $\lfloor 3 \log n \rfloor$ by Theorem 8. So by Lemma 3 the length of $f(v)$ is at most

$$\lfloor 3 \log n \rfloor + \lceil \log \lfloor 3 \log n \rfloor \rceil + 1 \leq 3 \log n + \lceil \log 3 + \log \log n \rceil + 1 < 3 \log n + \log \log n + 3.59.$$

This completes our proof. □

References

[1] S. Alstrup, E. B. Halvorsen, and K. G. Larsen. Near-optimal labeling schemes for nearest common ancestors. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 972–982. SIAM, 2014.

[2] D. Eppstein and M. Goodrich. Succinct greedy geometric routing using hyperbolic geometry. *Computers, IEEE Transactions on*, 60(11):1571–1580, 2011.

[3] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[4] Z. Király and S. Kisfaludi-Bak. A succinct tree coding for greedy navigation. Technical Report TR-2015-02, Egerváry Research Group, Budapest, 2015. www.cs.elte.hu/egres/.

[5] S. Kisfaludi-Bak, T. Sebők, Z. Király, and I. Csabai. Guaranteed Milgram routing using near-optimal address lengths. *Submitted manuscript.*

[6] R. Kleinberg. Geographic routing using hyperbolic space. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1902–1909, 2007.

[7] D. E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

[8] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 96–108. ACM, 2003.

[9] H. Zhang and S. Govindaiah. Greedy routing via embedding graphs onto semi-metric spaces. *Theoretical Computer Science*, 508(0):26 – 34, 2013. Frontiers of Algorithmics.