

Using spine decompositions to efficiently solve the length-constrained heaviest path problem for trees[☆]

Bishnu Bhattacharyya, Frank Dehne^{*}

School of Computer Science, Carleton University, Ottawa, Canada

ARTICLE INFO

Article history:

Received 31 October 2007
 Received in revised form 1 March 2008
 Accepted 31 May 2008
 Available online 11 June 2008
 Communicated by S.E. Hambrusch

Keywords:

Algorithms
 Analysis of algorithms
 Data structures

ABSTRACT

The length-constrained heaviest path (LCHP) in a weighted tree T , where each edge is assigned a weight and a length, is the path P in T with maximum total path weight and total path length bounded by a given value B . This paper presents an $O(n \log n)$ time LCHP algorithm which utilizes a data structure constructed from the spine decomposition of the input tree. This is an improvement over the existing algorithm by Wu et al. (1999), which runs in $O(n \log^2 n)$ time. Our method also improves on a previous $O(n \log n)$ time algorithm by Kim (2005) for the special case of finding a longest nonnegative path in a constant degree tree in that we can handle trees of arbitrary degree within the same time bounds.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Consider an undirected tree $T = (V, E)$, and define functions $w(e)$ and $l(e)$ to be the *weight* and *length* of each edge $e \in E$, respectively. For any path $path(u, v)$ between vertices u and v , we define the path weight $w(path(u, v)) = \sum_{e \in path(u, v)} w(e)$ and path length $l(path(u, v)) = \sum_{e \in path(u, v)} l(e)$. The *length-constrained heaviest path* for T is then defined as follows [6]:

Definition 1. Given a tree $T = (V, E)$ with edge weights $w(e)$ and edge lengths $l(e)$, and a real number B , then the *length-constrained heaviest path* (LCHP) for T is the path P such that

$$w(P) = \max_{u, v \in V} \{w(path(u, v)) \mid l(path(u, v)) \leq B\}$$

and $hw(T, w, l, B)$ denotes the weight of the length-constrained heaviest path for T .

LCHP can be used to solve network design problems on tree networks, where the edge weights represent bandwidth and the lengths represent link costs. A special case of LCHP, called the *longest nonnegative path* (LNP), has numerous applications in computational molecular biology and bioinformatics [1].

Definition 2. Given a tree $T = (V, E)$ with edge weights $w(e) = 1$ for all $e \in E$ and (arbitrary) edge lengths $l(e)$, then the *longest nonnegative path* (LNP) for T is the path P such that

$$w(P) = \max_{u, v \in V} \{w(path(u, v)) \mid l(path(u, v)) \geq 0\}.$$

LNP is a special case of LCHP. Note that Definition 1 allows arbitrary values for B , $w(e)$ and $l(e)$. They can be any real number, and path weight can also be minimized or B can be used as a lower bound instead of an upper bound by using negative numbers.

In [6], Wu et al. presented a recursive algorithm that solves LCHP in time $O(n \log^2 n)$. An $O(n \log n)$ -time algorithm for LNP on trees of constant degree was presented in [4].

In this paper, we improve on both of the above results. We present an $O(n \log n)$ time algorithm that solves LCHP on an arbitrary undirected tree. Our method is based

[☆] Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

^{*} Corresponding author.

E-mail addresses: bbhattac@connect.carleton.ca (B. Bhattacharyya), frank@dehne.net (F. Dehne).

URL: <http://www.dehne.net> (F. Dehne).

on a recursive decomposition of the input tree T called the *spine decomposition* [2] and improves on the currently known $O(n \log^2 n)$ time complexity for LCHP [6]. As a corollary, our method can also solve LNP for arbitrary trees in time $O(n \log n)$ which improves on the result in [4] where this time complexity could only be achieved for trees with constant degree.

2. Spine decomposition of trees

2.1. Tree decompositions

Definition 3. A general *decomposition* of a tree T , denoted $D(T)$, is a collection of subtrees of T such that

1. $T \in D(T)$.
2. For all $T_1, T_2 \in D(T)$ either T_1 and T_2 are disjoint, or one is strictly contained in the other.

The *depth* of a decomposition is the maximum cardinality of $H \subseteq D(T)$ such that

$$H = \{T_1, T_2, \dots, T_k \mid T_1 \subset T_2 \subset \dots \subset T_k\}.$$

It is important to define the depth of a tree decomposition, since it directly influences the running time of our algorithm. A common tree decomposition that is used is the *centroid decomposition* [3].

Definition 4. A *centroid* of a tree T is a vertex x whose removal results in a set of subtrees T_1, \dots, T_k such that for all $1 \leq i \leq k$, $|T_i| \leq |T|/2$ (where $|T|$ denotes the number of vertices in T).

Any tree T has at least one centroid. Let $T(v)$ denote the set of subtrees formed by removing vertex v from T . A centroid decomposition $CD(T)$ is formed by starting with $\{T\}$, finding its centroid x , and adding $T(x)$ to the set of components. This procedure is applied on each tree in $CD(T)$ until the components added are single vertices. The depth of $CD(T)$ is $O(\log n)$ [3]. Note that a centroid decomposition can be represented by a (rooted) tree where each node corresponds to a subtree of T . The depth of this tree is equal to the depth of $CD(T)$. We mention this *decomposition tree* because it is utilized in the algorithm presented in Section 3.

A major flaw of centroid decompositions is that centroids in successive subtrees are unrelated; if $T_i \subset T_j$, we can infer no knowledge about the centroid of T_i by knowing the centroid of T_j . Thus, if we are to evaluate an expression at each node of the decomposition tree, it must be recomputed every time.

2.2. Spine decomposition

A spine decomposition (denoted $SD(T)$) [2] is a special case of a general tree decomposition that consists of a collection of *spines*, or paths from the root of a tree to a leaf.

First, without loss of generality assume that T is a rooted binary tree. If T has no root, we can arbitrarily assign one. If T is not binary, we can transform it into a

binary tree by adding $O(n)$ nodes and zero-length, zero-weight edges [5].

Lemma 1. Suppose (T, w, l, B) is an instance of LCHP, where T is an arbitrary tree. Let T' denote the rooted binary transformation of T , and define functions w', l' as follows:

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \text{ is an edge in } T, \\ 0 & \text{otherwise,} \end{cases}$$

$$l'(u, v) = \begin{cases} l(u, v) & \text{if } (u, v) \text{ is an edge in } T, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{Then, } hw(T, w, l, B) = hw(T', w', l', B).$$

Proof. Since all edges in $T' \setminus T$ have zero weight and zero length, for every path in T there exists a corresponding path in T' with identical weight and length, and vice versa. (For a path p' in T' , select the corresponding path in T obtained by removing all edges in $T' \setminus T$ from p' . For a path p in T , select the corresponding path in T' with the same end points as p .) Hence, $hw(T, w, l, B) = hw(T', w', l', B)$. \square

For the remainder of this section, we assume that T is a rooted binary tree with n nodes and root r_T . $T(v)$ is the subtree of T rooted at v . The number of *descended leaves* from v , denoted $N_l(v)$, is the number of leaf nodes in T that have v as an ancestor.

Definition 5. The *spine decomposition* $SD(T)$ for a rooted binary tree T with root r_T is a tree decomposition consisting of a collection of *spine* subtrees defined as follows: First, the spine $\pi(r_T, l)$ is chosen where $\pi(r_T, l)$ is a path $\{v_0 = r_T, v_1, \dots, v_k = l\}$ in T such that l is a leaf and for all $i = 0, \dots, k-1$ the node v_{i+1} is the child node of v_i with the largest number of descended leaves in T . Next, recursively compute the spine decompositions for every subtree $T(u_i)$ rooted at a node u_i adjacent to $\pi(r_T, l)$. In addition, construct for each spine a binary search tree whose leaves are the vertices of the spine.

Note that, in some trees, a spine can be of length $O(n)$. Consider an algorithm that processes $SD(T)$ bottom-up. Gathering information from that many subtrees in one level of the recursion is inefficient. This is addressed by building a binary search tree (BST) on top of every spine. The leaves of the BST are nodes on the spine. To build the BST with root x on spine $\pi = \{v_0, \dots, v_k\}$, denote $\lambda(v_i) = N_l(T(u_i))$, where u_i is the child of v_i that is not in π . If u_i does not exist, $\lambda(v_i) = 1$. Compute m such that $|\sum_{(i=0)}^m \lambda(v_i) - \sum_{(i=m+1)}^k \lambda(v_i)|$ is minimized, and then recursively compute the BSTs for node x_1 with spine $\{v_0, \dots, v_m\}$ and node x_2 with spine $\{v_{m+1}, \dots, v_k\}$. Minimizing this difference balances the search tree by weight. x_1 and x_2 are then assigned as the left and right child of x , respectively. Consequently, a spine node with many descended leaves will be closer to the root of the binary search tree. The spine decomposition of the tree T in Fig. 1 is shown in Fig. 2. It is important to note that $SD(T)$ includes every vertex and edge in T .

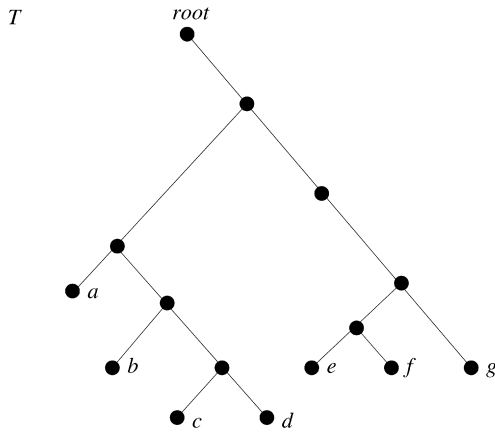


Fig. 1. Tree T .

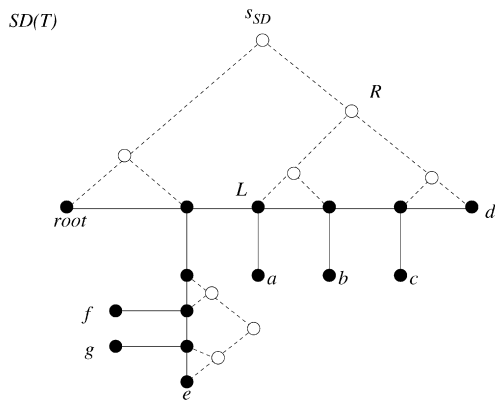


Fig. 2. The spine decomposition $SD(T)$ of the tree T in Fig. 1. Black vertices and solid lines represent nodes and edges of T . White vertices and dashed vertices represent the binary search trees. From this diagram, we see that all nodes and edges in T are also in $SD(T)$.

$SD(T)$ can be computed in $O(n)$ time. The resulting decomposition tree is of height $O(\log n)$ and has $O(n)$ vertices [2]. Note that the height of this tree is independent of the height of T . We denote s_{SD} as the root of the search tree of the first spine in $SD(T)$. s_{SD} is the root of the decomposition tree of T .

3. The algorithm

Our algorithm is presented in three parts. For improved readability, we present only the computation of the weight of the heaviest path. However, it is a simple modification to also compute the path itself.

Initially, *LCHPsolve* (Algorithm 1) pre-processes T by converting it to a rooted binary tree T' and computing the spine decomposition $SD(T')$. In other words, it computes the transformation illustrated in Figs. 1 and 2. It then initiates the recursion by calling *recurseLCHP* (Algorithm 2). However, before we describe *recurseLCHP*, we need some notation:

- If v is a node of a binary search tree, $left(v)$ is the left child of v . $right(v)$ is defined analogously.

Algorithm 1. *LCHPsolve*

- 1: **Input:** Tree T , weight function w , length function l , threshold B
 - 2: **Output:** $soln \leftarrow hw(T, w, l, B)$
 - 3: **if** T is not a rooted binary tree **then**
 - 4: Convert T to a rooted binary tree T' . Create w' and l' accordingly
 - 5: **end if**
 - 6: Construct $SD(T')$
 - 7: Output $recurseLCHP(SD(T'), s_{SD}, w', l', B).soln$
-

Algorithm 2. *recurseLCHP*

- 1: **Input:** Spine decomposition $SD(T)$, node x in $SD(T)$, weight function w , length function l , threshold B
 - 2: **Output:** $soln \leftarrow hw(T, w, l, B)$, $X \leftarrow$ length-sorted array of paths in T ending at $leftmost(x)$, $Y \leftarrow$ length-sorted array of paths in T ending at $rightmost(x)$
 - 3: $X_T := []$
 - 4: $Y_T := []$
 - 5: **if** x is a leaf of T **then**
 - 6: Return $(-\infty, X_T, Y_T)$
 - 7: **else if** x is a spine node **then**
 - 8: **if** $deg(x) = 2$ **then**
 - 9: Return $(-\infty, X_T, Y_T)$
 - 10: **else**
 - 11: $y :=$ The child of x that is in the spine below x in $SD(T)$
 - 12: $z := s_{SD}$ of the spine decomposition of the subtree of T rooted at y
 - 13: $S := recurseLCHP(SD(T), z, w, l, B)$
 - 14: Append edge (x, y) to all paths in $S.X$ and adjust path weight/length accordingly
 - 15: Insert path $P = ((x, y))$ into $S.X$
 - 16: $soln := \max_{i \in S.X} \{S.X[i].weight | S.X[i].length \leq B\}$
 - 17: Return $(\max\{soln, S.soln\}, S.X, S.X)$
 - 18: **end if**
 - 19: **else if** x is a node on a binary search tree **then**
 - 20: Return $BSTnodes(SD(T), x, w, l, B)$
 - 21: **end if**
-

- If v is a node of a binary search tree, $leftmost(v)$ defines the spine node found by repeatedly traversing the left edge from v . $rightmost(v)$ is defined analogously. If v is a spine node, $leftmost(v) = rightmost(v) = v$. In Fig. 2, $leftmost(s_{SD}) = root$ and $rightmost(s_{SD}) = d$. We adopt the convention that leftmost always points towards the head of the spine.
- When discussing *recurseLCHP* (Algorithm 2) and *BSTnode* (Algorithm 3), we may refer to rooted binary tree T' as T . The notation can be simplified since both of these algorithms are oblivious as to whether T was pre-processed or not.

We now outline Algorithms 2 and 3. *recurseLCHP* solves LCHP for the subtree of the decomposition tree of $SD(T)$ that is denoted by a node x in the tree.

When processing $SD(T)$, there are three cases to consider. The first case is when the current node x being processed is a leaf of T . In Fig. 2, these correspond to vertices a, b, c, d, e, f , and g . The second case is when x is not a leaf, yet is still a spine vertex. This corresponds to the remaining black vertices in Fig. 2. The final case is when x is a search node of $SD(T)$, or a white node in Fig. 2.

Algorithm 3. *BSTnode*

```

1: Input: Spine decomposition  $SD(T)$ , binary search tree node  $x$ , weight
   function  $w$ , length function  $l$ , threshold  $B$ 
2: Output:  $soln \leftarrow hw(T, w, l, B)$ ,  $X \leftarrow$  length-sorted array of paths in  $T$ 
   ending at  $leftmost(x)$ ,  $Y \leftarrow$  length-sorted array of paths in  $T$  ending
   at  $rightmost(x)$ 

3:  $P := []$ 
4:  $X_T, Y_T := []$ 
5:  $L := recurseLCHP(SD(T), left(x), w, l, B)$ 
6:  $R := recurseLCHP(SD(T), right(x), w, l, B)$ 
7: Let  $lt$  and  $rt$  denote the left and right children of  $x$ , respectively
8:  $e := edge(rightmost(lt), leftmost(rt))$ 
9: Append edge  $e$  to all paths in  $R.X$ , and merge  $L.Y$  and  $R.X$  into list  $P$ 
10: Insert the path consisting of the single edge  $e$  into  $P$  such that
    the list remains in sorted order
11:  $soln := \max\{L.soln, R.soln\}$ 

12: Let  $n$  denote the number of elements in  $P$ 
13:  $soln := \{soln \mid \max_{1 \leq i \leq n} \{P[i].weight \mid P[i].length \leq B\}\}$ 
14: for all  $i$  such that  $1 \leq i \leq n$  do
15:    $best[i] := P[\alpha]$  such that  $P[\alpha].weight = \max_{1 \leq j \leq i} \{P[j].weight\}$ 

16:  $otherbest[i] := P[\beta]$  such that  $P[\beta].weight = \max_{1 \leq j \leq i} \{P[j].$ 
    $weight\}$  and path  $P[\beta]$  and  $P[\alpha]$  do not share any edges.
   If no such  $\alpha$  exists,  $otherbest[\beta] := -\infty$ 
17: end for
18:  $j := n$ 
19: for  $i = 1$  to  $n$  do
20:   while  $P[i].length + P[j].length > B$  do
21:      $j := j - 1$ 
22:   end while
23:   if  $j < 1$  then
24:     break
25:   end if
26:   if  $P[i]$  and  $best[j]$  share one or more edges (and thus cannot be
     concatenated) then
27:      $soln := \max\{soln, P[i].weight + best[j].weight\}$ 
28:   else
29:      $soln := \max\{soln, P[i].weight + otherbest[j].weight\}$ 
30:   end if
31: end for
32: Append  $path(leftmost(x), leftmost(rt))$  to all paths in  $R.X$ , and merge
    $R.X$  and  $L.X$  into  $X_T$ .
33: Append  $path(rightmost(lt), rightmost(rt))$  to all paths in  $L.Y$ , and
   merge  $L.Y$  and  $R.Y$  into  $Y_T$ 
34: Return  $(soln, X_T, Y_T)$ 

```

In addition to solving LCHP, *recurseLCHP* also returns two length-sorted lists of paths in the subtree. One list is of all paths that terminate at $leftmost(x)$, the other is of all paths that terminate at $rightmost(x)$. These paths are denoted X and Y , respectively. In the first case, where x is a leaf of T , these lists are empty and the solution to LCHP is $-\infty$ (*recurseLCHP*, line 6).

In the second case, where x is a (non-leaf) spine node, the situation is more complex. If $\deg(x) = 2$ we can treat x as if it is a leaf of T . Otherwise, we must first recurse on the subtree of $SD(T)$ rooted at node y , the child of x that is *not* in the current spine. We take the list of paths returned and append edge (x, y) to all of them, adjusting path weight/length accordingly (the list remains sorted) (*recurseLCHP*, lines 13–15). If any of these new paths are a better solution to LCHP than the one returned by the recursive call, we record that (*recurseLCHP*, line 16). Note that in these cases the left list and the right list will be identical.

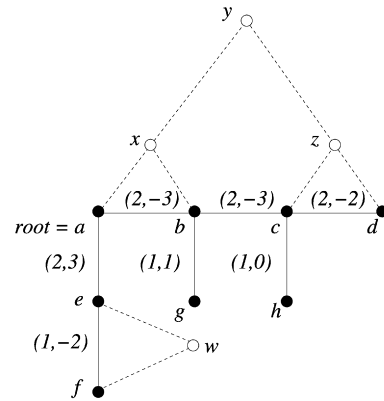


Fig. 3. An example of LCHP. The edges are labeled with (weight, length) pairs, and the solution computed at y is illustrated. The root of the original tree is a , and the root of the top search tree of the spine decomposition is y .

The most complicated case is the third one, when x is a node in a binary search tree above a spine. This is handled by *BSTnode* (Algorithm 3).

Definition 6. If v is a node in a binary search tree in $SD(T)$, the subtree of T that is formed by taking the spine segment from $leftmost(v)$ to $rightmost(v)$ and all spines that hang off it is the subtree of T that is *covered* by v , denoted T_v . In Fig. 2, R covers the spine segment from l to d , as well as leaf nodes a , b , and c .

This is the only case where x is not a node in the original tree. We solve LCHP for the subtree T_x of T . After computing LCHP for $left(x)$ and $right(x)$ (denoted L and R , respectively), we look for the maximum length-constrained path in T_x passing through edge $e = (rightmost(left(x)), leftmost(right(x)))$. We first append e to all the paths in the list $R.X$ and merge with $L.Y$. This results in a list of paths terminating at vertex $w = rightmost(left(x))$.

To compute the best path containing e , we first check the current best solution against all paths in T_x terminating at w (*BSTnode*, line 13). We then check all paths that contain e using the method of [7]. For each path P that terminates at vertex w we first compute the path of maximum weight Q such that $w(Q) \leq w(P)$ for both the left and right subtree of T_x descended from w (*BSTnode*, lines 14–17). Thus, the path starting at some vertex v and passing through e can be quickly calculated by first finding the vertex u such that $path(u, v)$ is the path of greatest length passing through u, v , and w (*BSTnode*, line 20). We then replace the segment $path(u, v)$ with the heaviest path of lesser or equal length in the appropriate subtree (*BSTnode*, lines 23–26). This path is guaranteed to be the heaviest path passing through w and v obeying the length constraint.

Once the solution for the T_x has been computed, we construct a length-sorted list of paths terminating at $leftmost(x)$ and $rightmost(x)$ and pass the solution upwards (*BSTnode*, lines 27–29).

Before analyzing correctness and running-time, we present an example of *LCHPSolve* that illustrates our algorithm in Fig. 3. The root of the original tree is a , and the root of the top search tree of the spine decomposition is y . The edges are labeled with (weight, length) pairs and the threshold B is set to 0. The left path and right path lists for all spine nodes are identical, so when processing them we do not distinguish between them (this is because if v is a spine node $leftmost(v) = rightmost(v) = v$). *LCHPSolve* terminates on nodes e, f, g, h , and d . In these cases, the shortest path is $-\infty$ and the lists of paths are empty. For node b , edge (b, g) is merged with the solution for node g . Since (b, g) has length 1, the length-constrained heaviest path still has weight $-\infty$, but the list of paths terminating at b is now $\{gb\}$. For node c , edge (c, h) is merged with the solution for h . Since it has length 0, the solution for c is 1, and the list of paths is $\{hc\}$.

The next node to be computed is w . This requires us to merge the solutions of e and f . The path appended to the path list for f is fe . This results in a solution of weight 1, and the list of paths is $\{fe\}$ (at both the left and right node).

Now a can be computed. Edge (e, a) is appended to the solution for w . This yields the paths $\{fea, ea\}$. Note that this list is length-sorted. The solution to LCHP at a remains 1.

We now have computed the solution for all the spine nodes, and can start on the search nodes of the top search tree (rooted at y). We begin with node x . The path list at a is $\{fea, ea\}$ and path list at b is $\{gb\}$. Appending edge (a, b) to the solution for b and merging results in the length-sorted path list $\{ba, gba, fea, ea\}$ (for the left) and $\{ab, feab, eab, gb\}$ (for the right). Scanning these lists, we see that the solution to LCHP is formed by concatenating fea with gba , resulting in a path of weight 6 and length -1 . Similarly, at node z , list $\{hc\}$ at c is merged with the empty list at d via edge (c, d) . This results in a solution of 3 (hcd) and a length-sorted path list of $\{dc, hc\}$ (left) and $\{cd, hcd\}$ (right).

In Fig. 3, $s_{SD} = y$, so the solution for y is the solution for the entire tree. The right list for node x and left list for node z are merged via path bc . The resulting list is $\{dcb, ab, hcb, feab, eab, gb\}$. Scanning this list results in path $feabcd$, of weight 9 and length -7 , as our solution to LCHP for the entire tree rooted at a .

Theorem 1. *Algorithm LCHP runs in time $O(n \log n)$, where n is the number of vertices in T .*

Proof. T can be transformed into a binary tree with $O(n)$ nodes and edges in $O(n)$ time [5], and the spine decomposition (of size $O(n)$) can be constructed in $O(n)$ time [2]. Therefore, $T_{LCHP}(n) = O(n) + T_{recurseLCHP}(n)$. For $T_{recurseLCHP}(n)$, we will consider cost per node processed.

Consider vertex x in T . Trivially, when x is processed at a leaf node of $SD(T)$, the cost is $O(1)$. At a spine node of degree 3, an edge is appended to the path from the root to x , and then it is checked against the current solution to LCHP (*recurseLCHP*, lines 11–17). This also costs $O(1)$ time.

At a BST node, x is merged into a combined list, and then checked against the current solution. Depending on

which subtree x is in, the path from x to the root may be extended, but in either case the cost remains the same. While computing *best* and *otherbest* for $1 \leq i \leq n$, we can remember and update the best path found so far, so x is checked a constant number of times (*BSTnode*, lines 14–16). In the nested loops, x is visited exactly twice (when it is indexed by i and j) (*BSTnode*, lines 19–20). Therefore, the total cost for x is again $O(1)$.

Since the depth of a spine decomposition is $O(\log n)$ [2], x appears in $O(\log n)$ subtrees of $SD(T)$. Therefore, with n vertices, the analysis yields $T_{LCHP}(n) = O(n) + T_{recurseLCHP}(n) = O(n) + O(n \log n) = O(n \log n)$. \square

Theorem 2. *Algorithm LCHP correctly computes $hw(T, w, l, B)$.*

Proof. It suffices to show that every path in T is checked by the algorithm. Consider an arbitrary path $P = \{u, \dots, v\}$ in T . Let $Q = \{w, \dots, z\}$ be the segment of P on the highest spine in $SD(T)$. Denote this spine S . For instance, in Fig. 3, if $P = gbch$, $Q = bc$, and $S = abcd$. Let y be the lowest common ancestor of w and z in the binary search tree over S . P is checked by LCHP when y is processed. \square

Corollary 1. *Algorithm LCHP also solves the LNP problem for trees of arbitrary degree in time $O(n \log n)$.*

Proof. LNP is a special case of LCHP. \square

4. Conclusion

In this paper, we presented an $O(n \log n)$ time algorithm for solving the length-constrained heaviest path (LCHP) problem for arbitrary trees. An interesting extension, for future research, would be to consider the dynamic version of this problem where we edit T (e.g., add/delete a leaf, prune/regraft a subtree) and then efficiently update the LCHP or LNP for T . This could be of particular interest for Bioinformatics applications where such tree edits occur frequently.

References

- [1] L. Allison, Longest biased interval and longest nonnegative sum interval, *Bioinformatics* 9 (2003) 1294–1295.
- [2] R. Benkoczi, B. Bhattacharyya, M. Chrobak, L. Larmore, W. Rydder, Faster algorithms for k -median problems in trees, in: 28th International Symposium on Mathematical Foundations of Computer Science vol. 2747, 2003, pp. 218–227.
- [3] R. Cole, U. Vishkin, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica* 3 (1988) 329–346.
- [4] S. Kim, Finding a longest nonnegative path in a constant degree tree, *Information Processing Letters* 93 (2005) 275–279.
- [5] A. Tamir, An $O(pn^2)$ algorithm for the p -median and related problems on tree graphs, *Operations Research Letters* 19 (1996) 59–64.
- [6] B.Y. Wu, K.-M. Chao, C.Y. Tang, An efficient algorithm for the length-constrained heaviest path problem on a tree, *Information Processing Letters* 69 (1999) 63–67.
- [7] B.Y. Wu, C.Y. Tang, An $O(n)$ algorithm for finding an optimal position with relative distances in an evolutionary tree, *Information Processing Letters* 63 (1997) 263–269.